Transient-Execution Attacks and Defenses

Habilitation by **Daniel Gruss**

June 2020

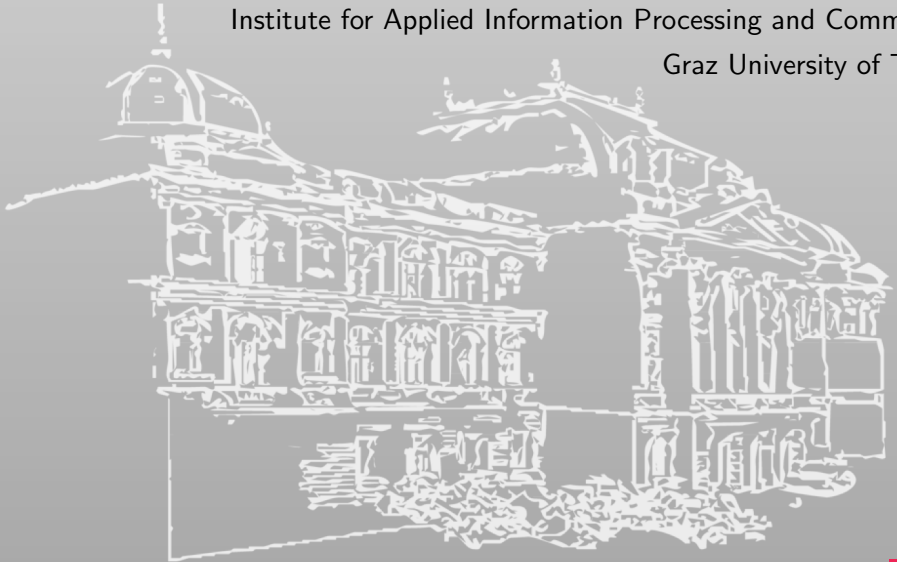
Daniel Gruss

Transient-Execution Attacks and Defenses (Part I only)

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

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.

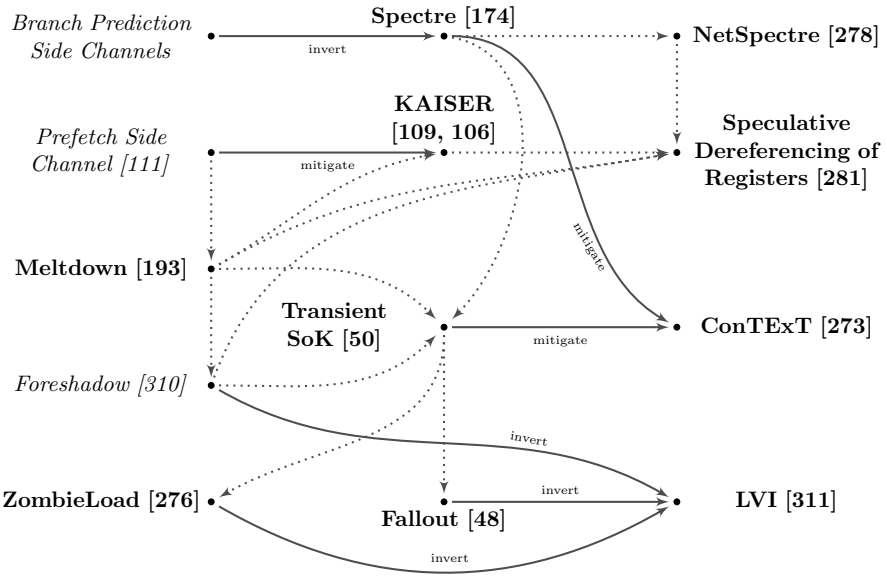


Figure 1.1.: Connection between the papers in this habilitation (highlighted in bold) and some related works. In some cases, previous attacks were inverted such that the victim experiences the former leakage, and by that becomes a confused deputy. In other cases, we developed mitigations for other attacks.

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.

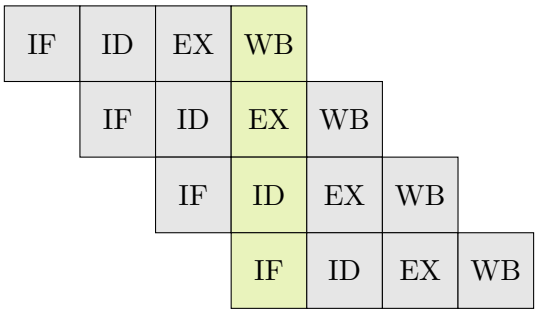


Figure 2.1.: A simple 4-stage pipeline. By interleaving instruction fetch (IF), instruction decoding (ID), instruction execution (EX), and write-back (WB), the processor can improve the instruction throughput substantially.

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.

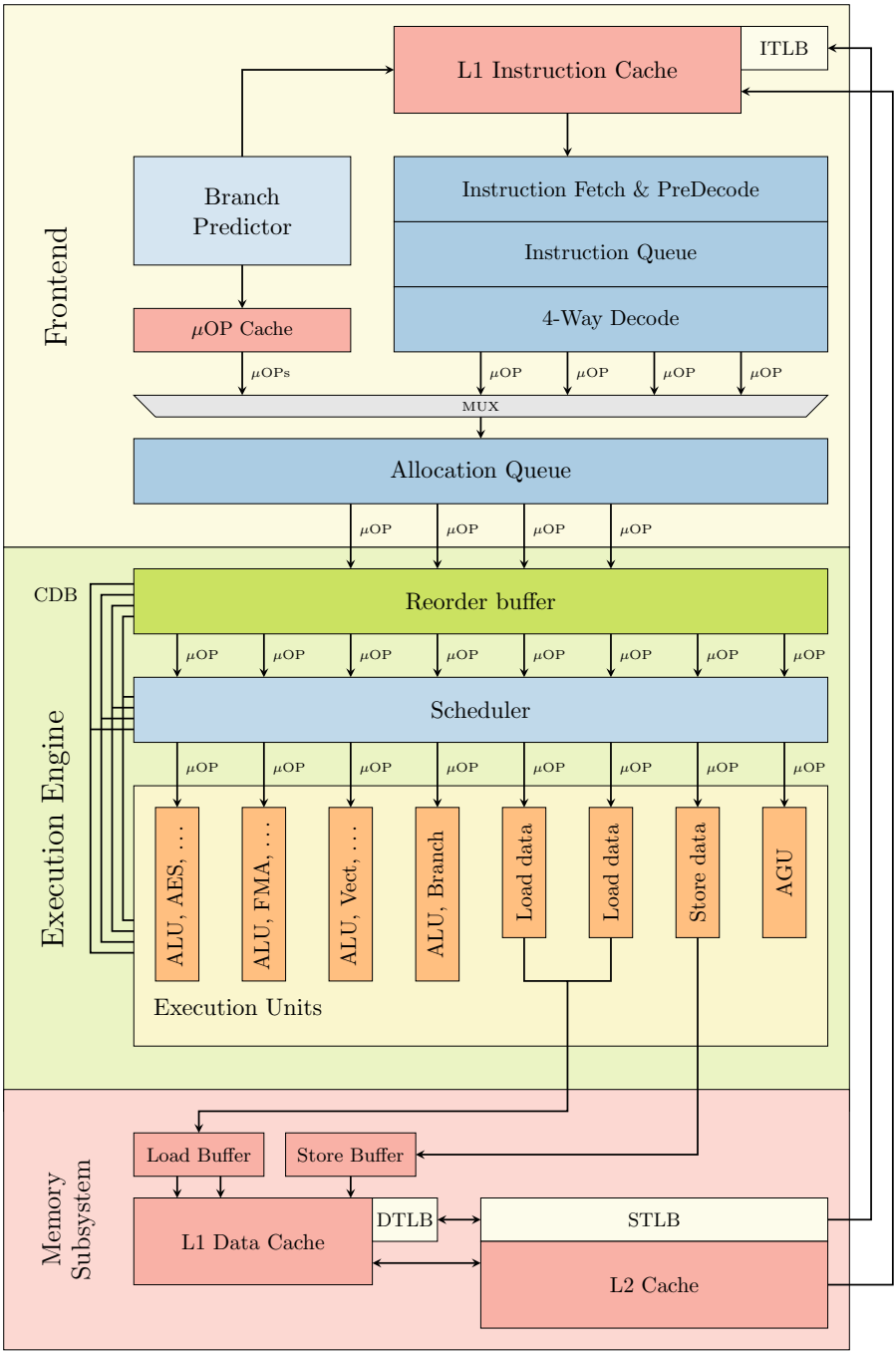


Figure 2.2.: Simplified illustration of a single core of the Intel's Skylake microarchitecture.

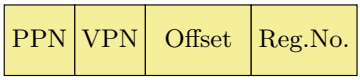


Figure 2.3.: Possible design for a load-buffer entry.

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.

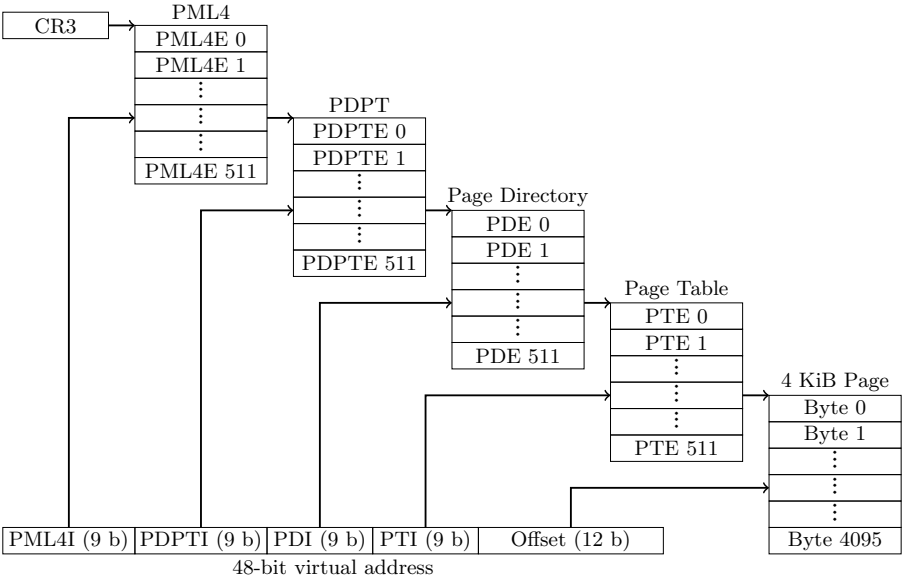


Figure 2.4.: Address translation for 4 KB pages on x86-64 processors. Starting with the PML4 base address from the CR3 register, the processor determines the physical address by using parts of the virtual address to index the different levels.

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.

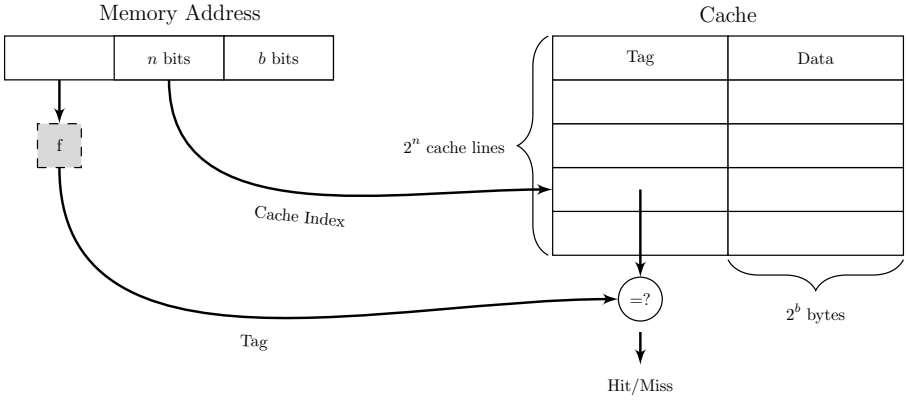


Figure 2.5.: A directly-mapped cache. Based on the middle n bits, the cache index is computed to choose a cache line. The tag is used to check whether an address is cached. If it is cached (cache hit), the 2^b bytes data are returned to the processor.

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-

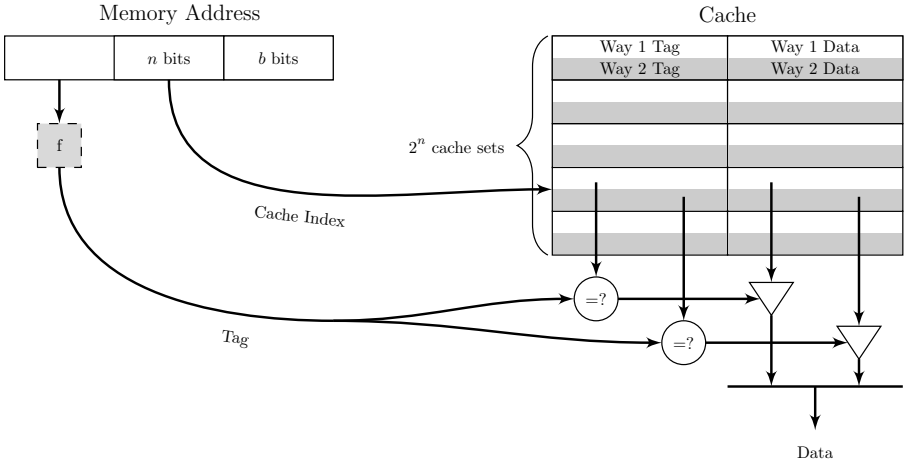


Figure 2.6.: A 2-way set-associative cache. The middle n bits are the cache index, selecting the cache set. The tag is used to check all ways simultaneously. The data in the matching cache way is returned to the execution core.

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

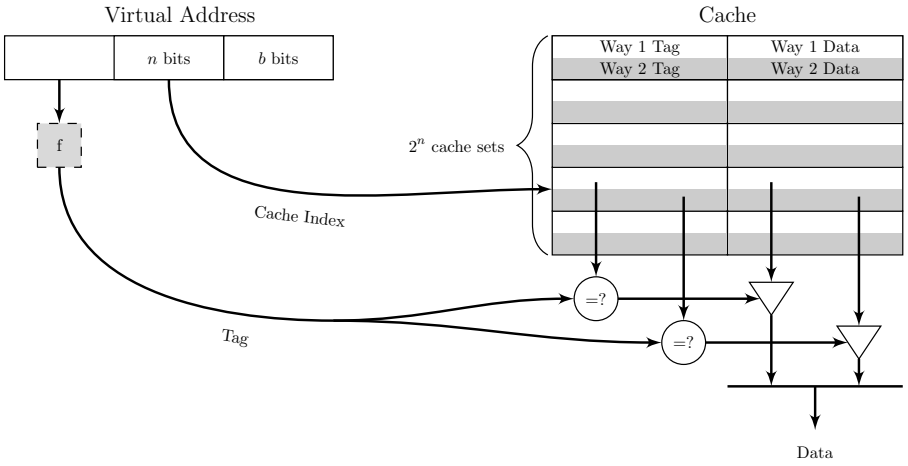


Figure 2.7.: A virtually-indexed virtually-tagged (VIVT) cache. The virtual address is used to compute both index and tag. The processor does not have to translate any addresses.

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

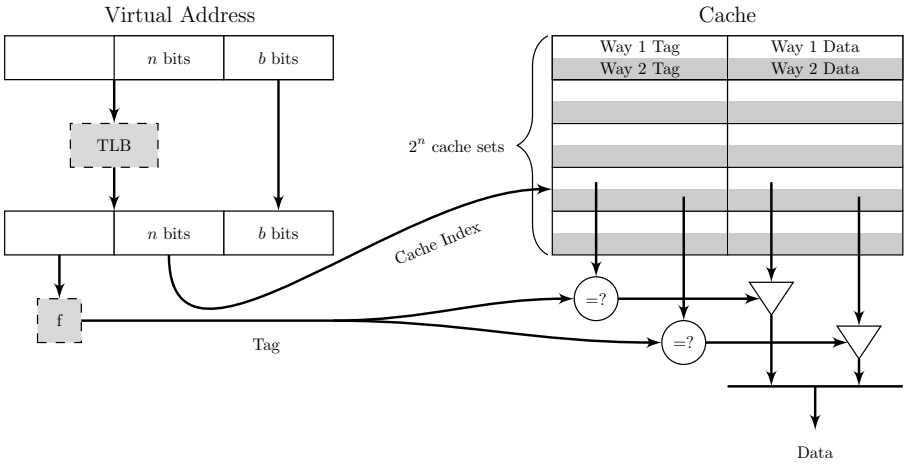


Figure 2.8.: A physically-indexed, physically-tagged (PIPT) cache. The physical address is used to compute both index and tag. The processor has to translate the virtual address before the cache set lookup.

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

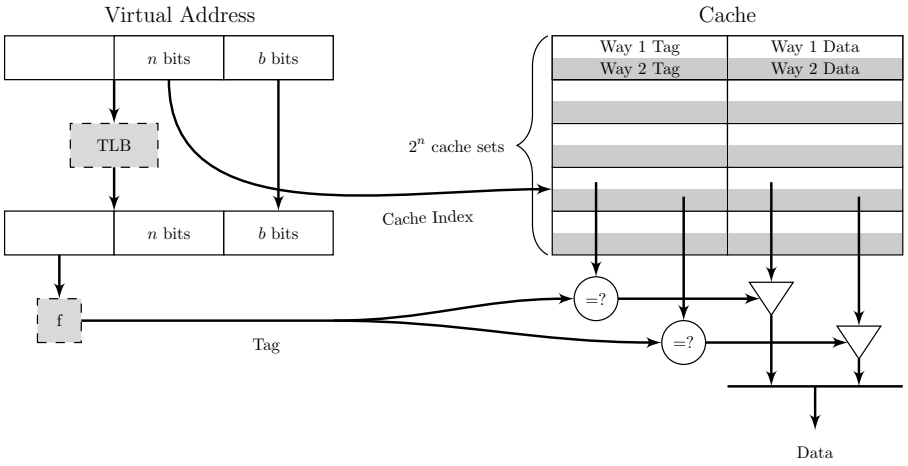


Figure 2.9.: A virtually-indexed, physically-tagged (VIPT) cache. The physical address is used to compute both the tag, but the virtual address is used to compute the index. The cache set lookup is done in parallel to the address translation and tag computation.

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.

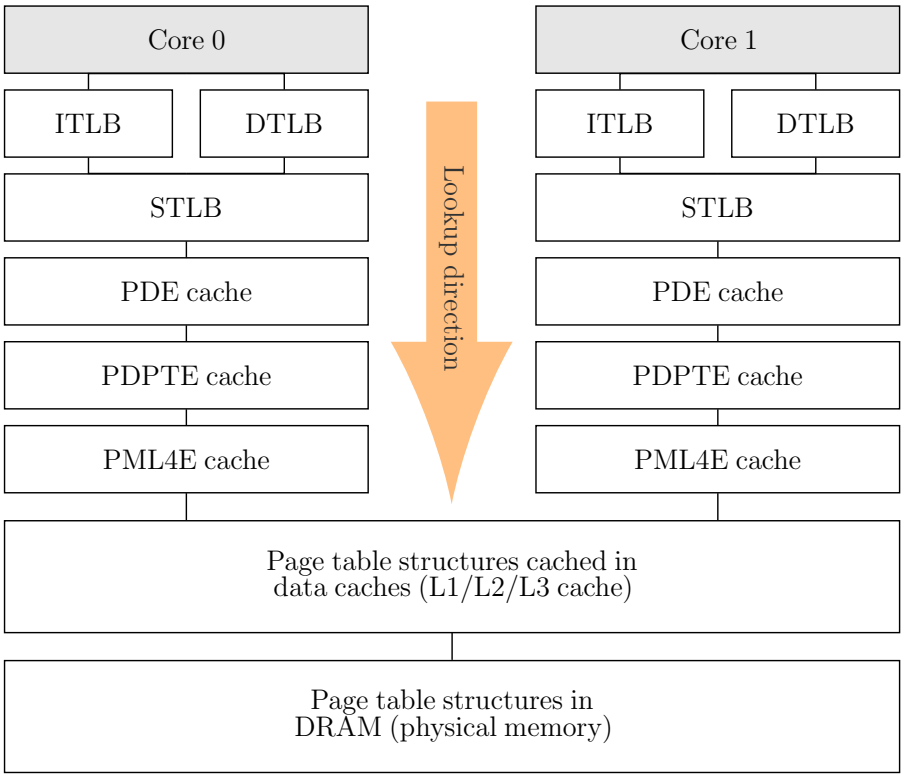


Figure 2.10.: The translation table cache hierarchy consists of multiple TLB levels and caches for each of the page table levels.

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

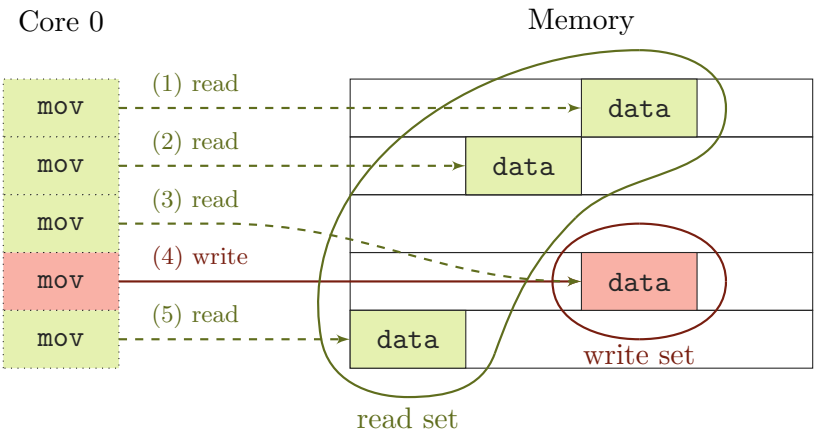


Figure 2.11.: Hardware transactional memory maintains a read set and a write set to be able to detect conflicts and revert transactions. Memory in the read set is unmodified; memory in the write set has been modified during the transaction.

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].

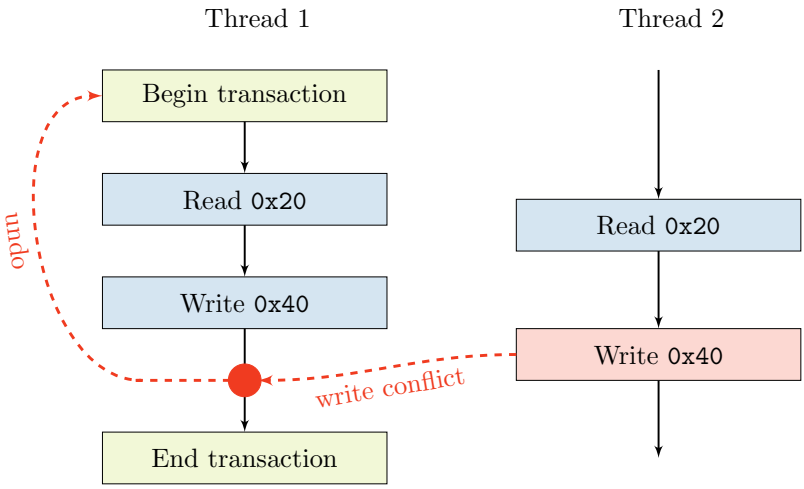


Figure 2.12.: Hardware transactional memory ensures that no concurrent modifications influence the transaction, either by preserving the old value or by aborting and reverting the transaction.

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.

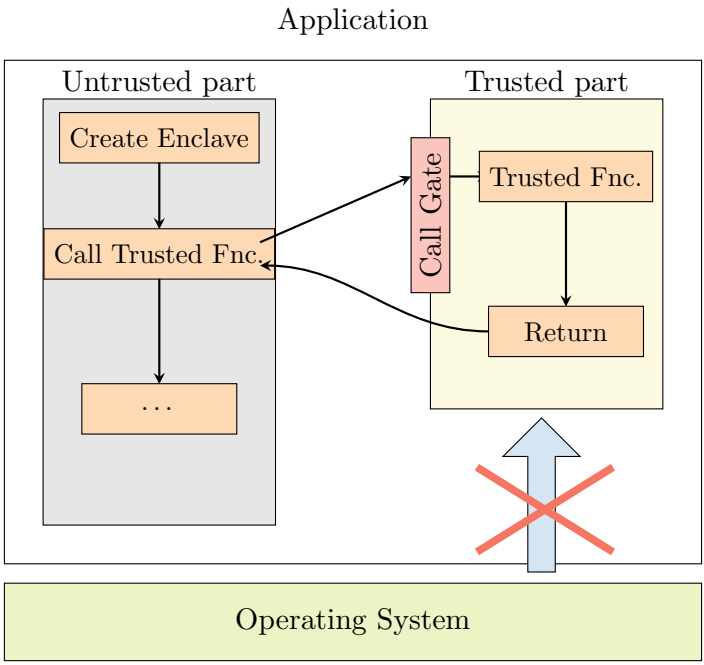


Figure 2.13.: With Intel SGX, applications are split into a trusted (enclave) and an untrusted (host) part. The hardware prevents any access to the trusted part. The only communication between enclave and host uses predefined `ecalls` and `ocalls`.

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

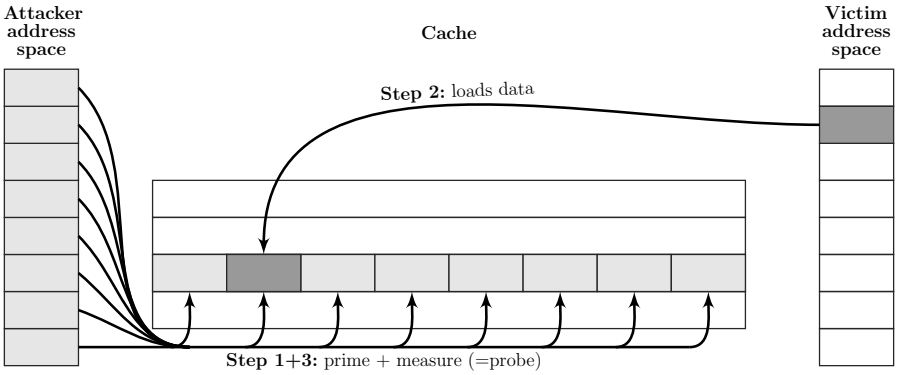


Figure 2.14.: A Prime+Probe attack illustrated in 3 steps [105]. The attacker continuously primes a cache set using its own memory locations and measures the execution time of this step (Step 1 and Step 3). In Step 2, the victim possibly accesses (non-shared) memory locations that map to the same cache 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.

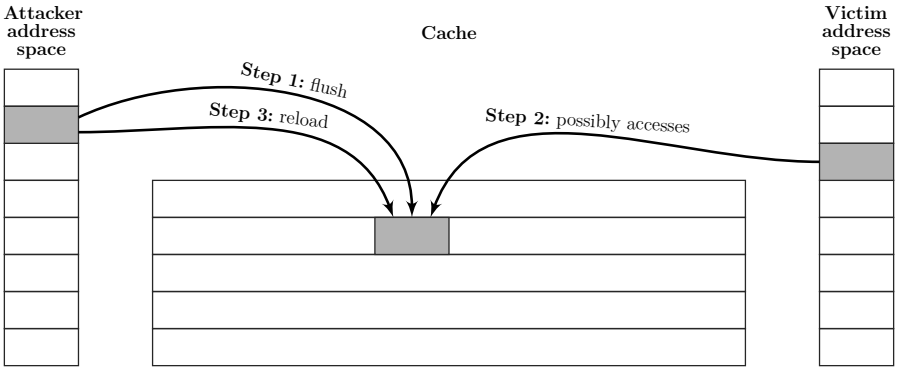


Figure 2.15.: A Flush+Reload attack illustrated in 3 steps [105]. In Step 1, the attacker flushes a shared memory location in the attacker's virtual address space. In Step 2, the victim possibly accesses the same shared memory location in the victim virtual address space. In Step 3, the attacker reloads 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.

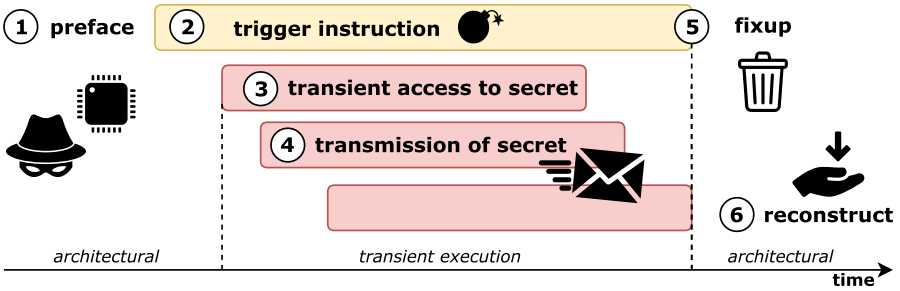


Figure 3.1.: High-level overview of a transient-execution attack in 6 phases. Note that we added an explicit Phase 3 for accessing the secret, compared to Canella et al. [50]: (1) prepare microarchitecture, (2) execute a *trigger instruction*, (3) *transient instructions* access data of interested to 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.

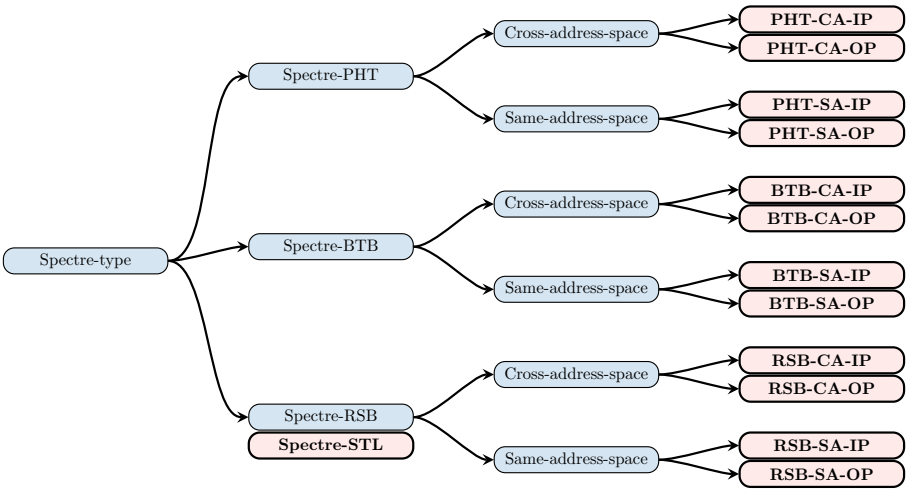


Figure 3.2.: State of the art Spectre classification tree [50].

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

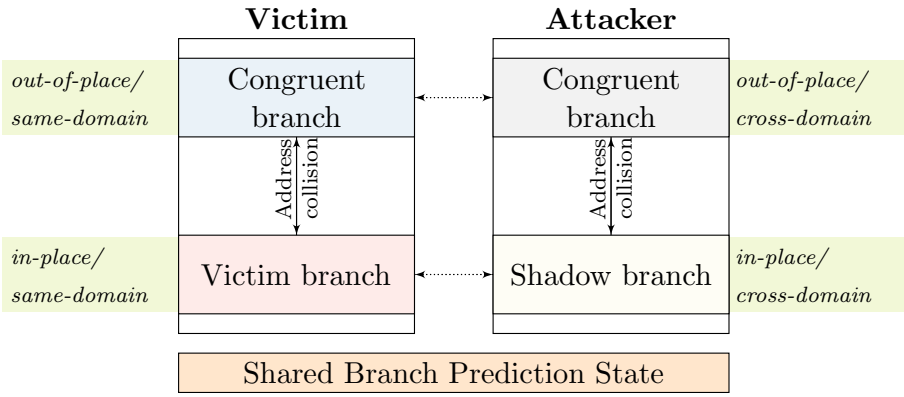


Figure 3.3.: A branch can be mistrained either within the victim domain (*same-domain*), or in an attacker-controlled domain (*cross-domain*); using the vulnerable branch itself (*in-place*), or a branch at a congruent virtual address (*out-of-place*).

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

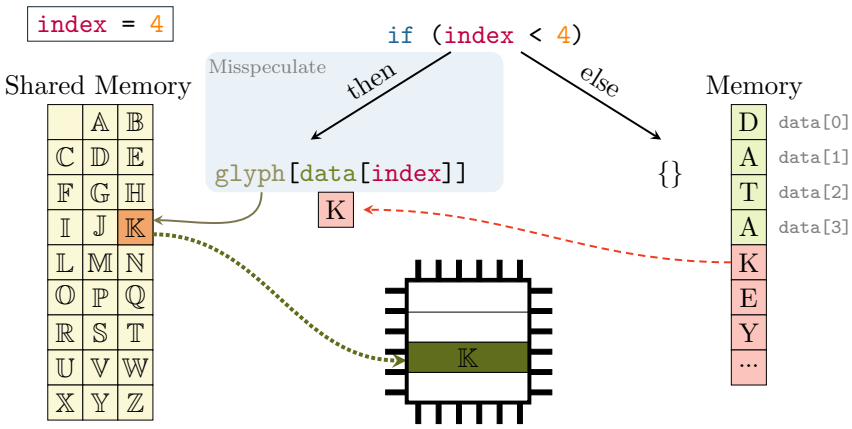


Figure 3.4.: A Spectre-PHT attack works by poisoning the PHT such that the victim misspeculates into a branch. In that branch, the victim accesses a secret and leaks it via a microarchitectural covert channel, e.g., a cache covert channel, as shown in this example. The secret value is encoded into an 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

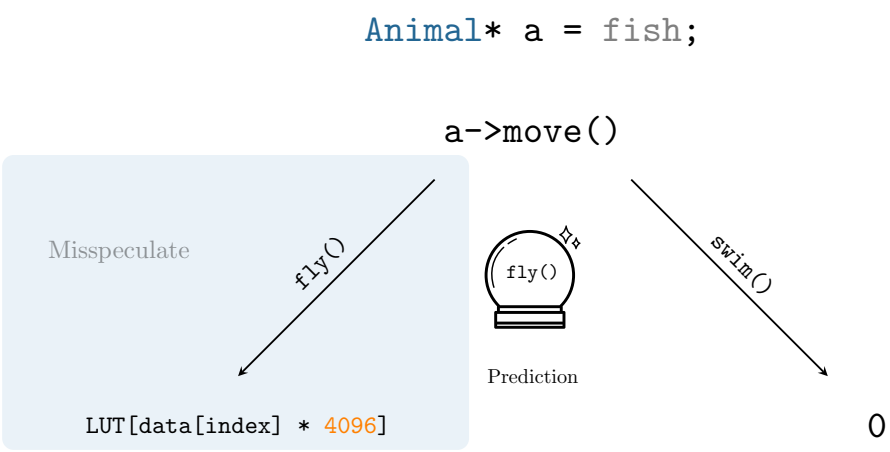


Figure 3.5.: A Spectre-BTB attack works by poisoning the BTB such that a wrong code address is predicted instead of the correct one. At that code location, the victim accesses a secret and leaks it via a microarchitectural covert channel, e.g., a cache covert channel, as shown in this example. The secret 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.

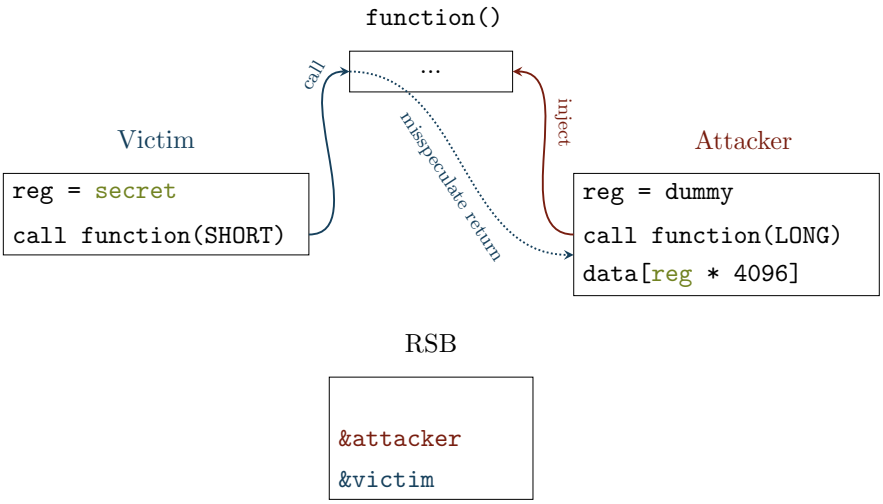


Figure 3.6.: A Spectre-RSB attack works by poisoning the RSB such that a wrong return address is predicted in a victim context. In this example, the attacker performs function calls while the victim is in a function. The victim then mispredicts the return to an attacker-chosen address. There the victim 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).

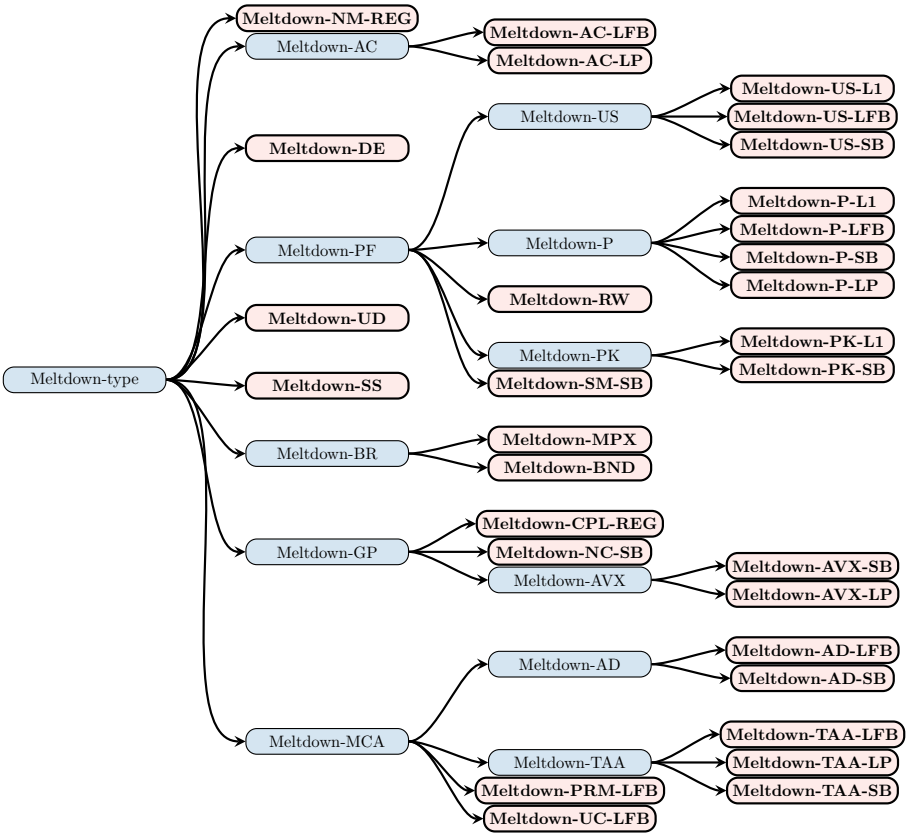


Figure 3.7.: State of the art Meltdown classification tree, extended from [50]. LVI can be viewed as an inverse-Meltdown-type attack and, hence, in principle, would enable the same attack variants regardless of the relevance of LVI in the specific attack scenarios.

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

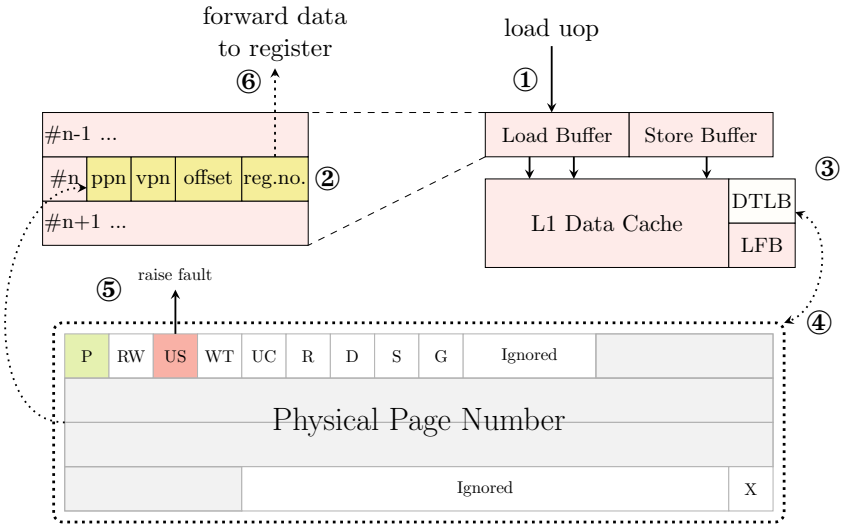


Figure 3.8.: The original Meltdown attack (Meltdown-US) [193] from a microarchitectural perspective. The illustration shows how the deferred permission check allows data to be forwarded to the target register of the load operation.

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

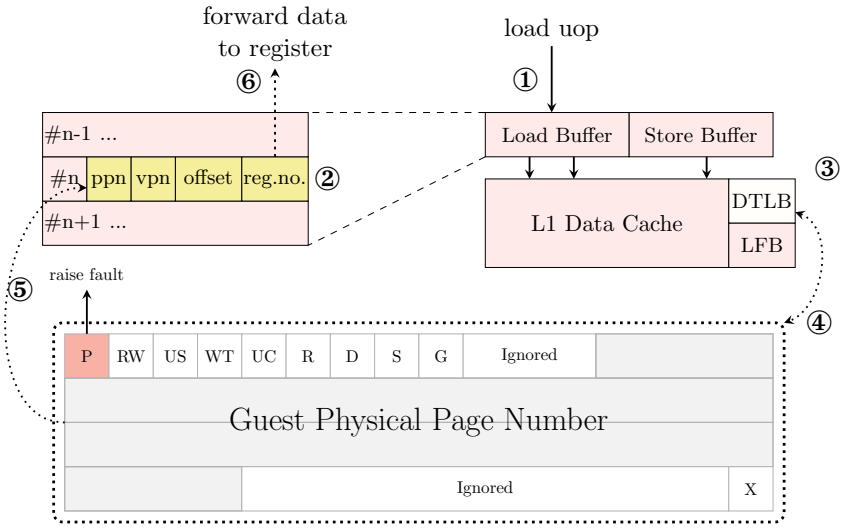


Figure 3.9.: The Foreshadow-VMM attack [310, 334] from a microarchitectural perspective. The illustration shows the steps that incorrectly use intermediate values to forward data to the target register of the load operation.

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

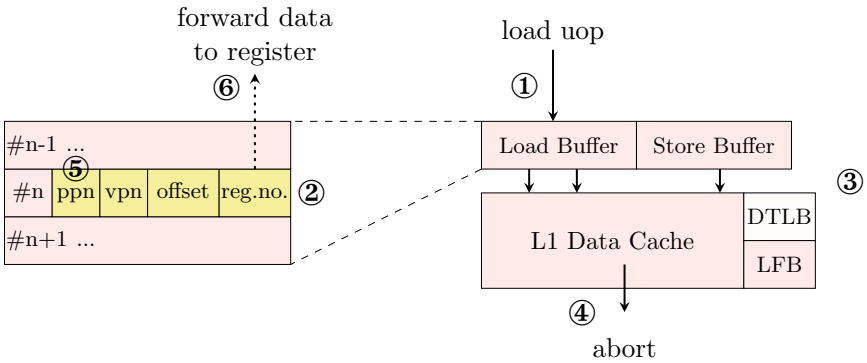


Figure 3.10.: The ZombieLoad v1 attack [276] from a microarchitectural perspective. The illustration shows the steps that lead to a use-after-free in the load buffer and thus wrong data being forwarded to the target register of the load operation.

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

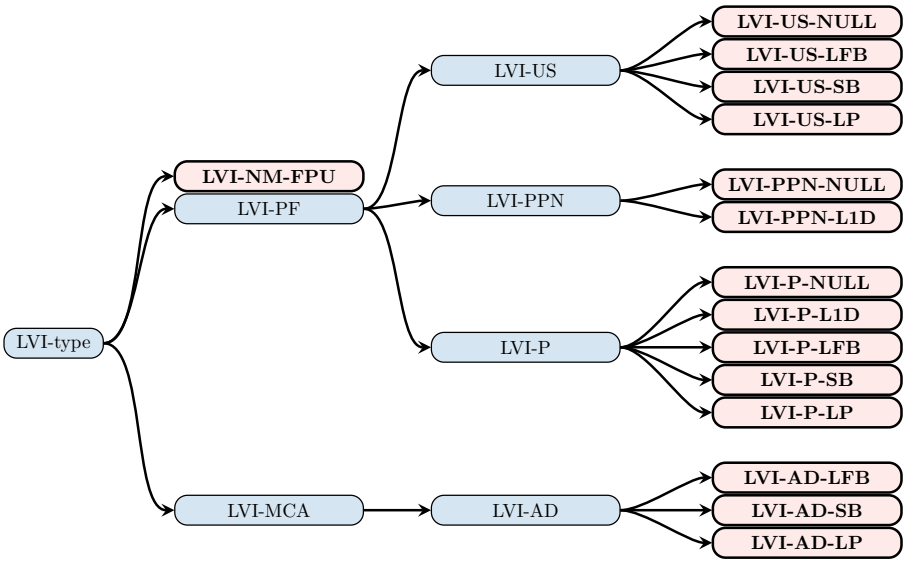


Figure 3.11.: State of the art LVI classification tree [311].

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).

Information on Part II

Note that Part II is not included in this PDF. Please download the full version for Part II.

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