Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs

Michael Schwarz
Graz University of Technology
michael.schwarz@iaik.tugraz.at

Claudio Canella
Graz University of Technology
claudio.canella@iaik.tugraz.at

Lukas Giner
Graz University of Technology
lukas.giner@iaik.tugraz.at

Daniel Gruss
Graz University of Technology
daniel.gruss@iaik.tugraz.at

ABSTRACT

Meltdown and Spectre exploit microarchitectural changes the CPU makes during transient out-of-order execution. Using side-channel techniques, these attacks enable leaking arbitrary data from memory. As state-of-the-art software mitigations for Meltdown may incur significant performance overheads, they are only seen as a temporary solution. Thus, software mitigations are disabled on more recent processors, which are not susceptible to Meltdown anymore.

In this paper, we show that Meltdown-like attacks are still possible on recent CPUs which are not vulnerable to the original Meltdown attack. We show that the store buffer—a microarchitectural optimization to reduce the latency for data stores—in combination with the TLB enables powerful attacks. We present several ASLR-related attacks, including a KASLR break from unprivileged applications, and breaking ASLR from JavaScript. We can also mount side-channel attacks, breaking the atomicity of TSX, and monitoring control flow of the kernel. Furthermore, when combined with a simple Spectre gadget, we can leak arbitrary data from memory. Our paper shows that Meltdown-like attacks are still possible, and software fixes are still necessary to ensure proper isolation between the kernel and user space.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Systems security**; **Operating systems security**.

KEYWORDS

side channel, side-channel attack, Meltdown, store buffer, store-to-load forwarding, ASLR, KASLR, Spectre, microarchitecture

1 INTRODUCTION

Modern processors have numerous optimizations to achieve the performance and efficiency that customers expect today. Most of these optimizations, e.g., CPU caches, are transparent for software developers and do not require changes in existing software. While the instruction-set architecture (ISA) describes the interface between software and hardware, it is only an abstraction layer for the CPUs microarchitecture. On the microarchitectural level, the CPU can apply any performance optimization as long as it does not violate the guarantees given by the ISA. Such optimizations also include pipelining or speculative execution. As the microarchitectural level is transparent and the optimizations are performed automatically, such optimizations are usually not or only sparsely documented. Furthermore, the main focus of microarchitectural optimizations is performance and efficiency, resulting in fewer security considerations than on the architectural level.

In recent years, we have seen several attacks on the microarchitectural state of CPUs, making the internal state of the CPU visible [13, 25, 62, 64, 85]. With knowledge about the internal CPU state, it is possible to attack cryptographic algorithms [4, 39, 41, 54, 62, 64, 85], spy on user interactions [26, 52, 68], or covertly transmit data [54, 57, 82, 83]. With the recent discovery of Meltdown [53], Foreshadow [77], and Foreshadow-NG [80], microarchitectural attacks advanced to a state where not only metadata but arbitrary data can be leaked. These attacks exploit the property that many CPUs still continue working out-of-order with data even if the data triggered a fault when loading it, e.g., due to a failed privilege check. Although the data is never architecturally visible, it can be encoded into the microarchitectural state and made visible using microarchitectural side-channel attacks.

While protecting against side-channel attacks was often seen as the duty of developers [5, 41], Meltdown and Foreshadow-NG showed that this is not always possible. These vulnerabilities, which are present in most Intel CPUs, break the hardware-enforced isolation between untrusted user applications and the trusted kernel. Hence, these attacks allow an attacker to read arbitrary memory, against which a single application cannot protect itself.

As these CPU vulnerabilities are deeply rooted in the CPU, close to or in the critical path, they cannot be fixed with microcode updates, but the issue is fixed on more recent processors [11, 37, 38]. Due to the severity of these vulnerabilities, and the ease to exploit them, all major operating systems rolled out software mitigations to prevent exploitation of Meltdown [14, 19, 28, 44]. The software mitigations are based on the idea of separating user and kernel space in stricter ways [22]. While such a stricter separation does not only prevent Meltdown, it also prevents other microarchitectural attacks on the kernel [22], e.g., microarchitectural KASLR (kernel address-space layout randomization) breaks [24, 33, 43]. Still, software mitigations may incur significant performance overheads, especially for workloads that require frequent switching between kernel and user space [19]. Thus, CPU manufacturers solved the root issue directly in hardware, making the software mitigations obsolete.

Although new CPUs are not vulnerable to the original Meltdown attack, we show that similar Meltdown-like effects can still be observed on such CPUs. In this paper, we investigate the store buffer and its microarchitectural side effects. The store buffer is a microarchitectural element which serializes the stream of stores and hides the latency when storing values to memory. It works similarly to a queue, completing all memory stores asynchronously while allowing the CPU to continue executing the execution stream out of order. To guarantee the consistency of subsequent load operations, load operations have to first check the store buffer for pending stores to the same address. If there is a store-buffer entry with a matching address, the load is served from the store buffer. This so-called store-to-load forwarding has been exploited in Spectre v4 [32], where the load and store go to different virtual addresses mapping the same memory location. Consequently, the virtual address of the load is not found in the store buffer and a stale value is read from the caches or memory instead. However, due to the asynchronous nature of the store buffer, Meltdown-like effects are visible, as store-to-load forwarding also happens after an illegal memory store.

We focus on correct store-to-load forwarding, *i.e.*, no positive or negative mismatches. We present three basic attack techniques that each leak side-channel information from correct store-to-load forwarding. First, *Data Bounce*, which exploits that stores to memory are forwarded even if the target address of the store is inaccessible to the user, e.g., kernel addresses. With *Data Bounce* we break KASLR, reveal the address space of Intel SGX enclaves, and even break ASLR from JavaScript. Second, Fetch+Bounce, which combines *Data Bounce* with the TLB side channel. With Fetch+Bounce we monitor kernel activity on a page-level granularity. Third, Speculative Fetch+Bounce, which combines Fetch+Bounce with speculative execution, leading to arbitrary data leakage from memory. Speculative Fetch+Bounce does not require shared memory between the user space and the kernel [46], and the leaked data is not encoded in the cache. Hence, Speculative Fetch+Bounce even works with countermeasures in place which only prevent cache covert channels.

We conclude that the hardware fixes for Meltdown are not sufficient on new CPUs. We stress that due to microarchitectural optimizations, security guarantees for isolating the user space from kernel space are not as strong as they should be. Therefore, we highlight the importance of keeping the already deployed additional software-based isolation of user and kernel space [22].

Contributions. The contributions of this work are:

(1) We discover a Meltdown-like effect around the store buffer on Intel CPUs (*Data Bounce*).
(2) We present Fetch+Bounce, a side-channel attack leveraging the store buffer and the TLB.
(3) We present a KASLR break, and an ASLR break from both JavaScript and SGX, and a covert channel.
(4) We show that an attacker can still leak kernel data even on CPUs where Meltdown is fixed (Speculative Fetch+Bounce).

Outline. Section 2 provides background on transient execution attacks. We describe the basic effects and attack primitives in Section 3. We present KASLR, and ASLR breaks with *Data Bounce* in Section 4. We show how control flow can be leaked with Fetch+Bounce in Section 5. We demonstrate how Speculative Fetch+Bounce allows leaking kernel memory on fully patched hardware and software in Section 6. We discuss the context of our attack and related work in Section 7. We conclude in Section 8.

Responsible Disclosure. We responsibly disclosed our initial research to Intel on January 18, 2019. Intel verified our findings. The findings were part of an embargo ending on May 14, 2019.

2 BACKGROUND

In this section, we describe the background required for this paper. We give a brief overview of caches, transient execution and transient execution attacks, store buffers, virtual memory, and Intel SGX.

2.1 Cache Attacks

Processor speeds increased massively over the past decades. While the bandwidth of modern main memory (DRAM) has increased accordingly, the latency has not decreased to the same extent. Consequently, it is essential for the processor to fetch data from DRAM ahead of time and buffer it in faster internal storage. For this purpose, processors contain small memory buffers, called caches, that store frequently or recently accessed data. In modern processors, the cache is organized in a hierarchy of multiple levels, with the lowest level being the smallest but also the fastest. In each subsequent level, the size and access time increases.

As caches are used to hide the latency of memory accesses, they inherently introduce a timing side channel. Many different cache attack techniques have been proposed over the past two decades [5, 25, 47, 62, 85]. Today, the most important techniques are Prime+Probe [62, 64] and Flush+Reload [85]. Variants of these attacks that are used today exploit that the last-level cache is shared and inclusive on many processors. Prime+Probe attacks constantly measure how long it takes to fill an entire cache set. Whenever a victim process accesses a cache line in this cache set, the measured time will be slightly higher. In a Flush+Reload attack, the attacker constantly flushes the targeted memory location using the `clflush` instruction. The attacker then measures how long it takes to reload the data. Based on the reload time the attacker determines whether a victim has accessed the data in the meantime. Due to its fine granularity, Flush+Reload has been used for attacks on various computations, e.g., web server function calls [87], user input [26, 52, 68], kernel addressing information [24], and cryptographic algorithms [4, 41, 85].

Covert channels are a particular use case of side channels. In this scenario, the attacker controls both the sender and the receiver and tries to leak information from one security domain to another, bypassing isolation imposed on the functional or the system level. Flush+Reload as well as Prime+Probe have both been used in high-performance covert channels [25, 54, 57].

2.2 Transient-execution Attacks

Modern processors are highly complex and large systems. Program code has a strict in-order instruction stream. However, if the processor would process this instruction stream strictly in order, the processor would have to stall until all operands of the current instruction are available, even though subsequent instructions might be ready to run. To optimize this case, modern processors first fetch and decode an instruction in the frontend. In many cases, instructions are split up into smaller micro-operations (μOPs) [15]. These

μOPs are then placed in the so-called Re-Order Buffer (ROB). μOPs that have operands also need storage space for these operands. When a μOP is placed in the ROB, this storage space is dynamically allocated from the load buffer, for memory loads, the store buffer, for memory stores, and the register file, for register operations. The ROB entry only references the load buffer and store buffer entries. While the operands of a μOP still might not be available after it was placed in the ROB, we can now schedule subsequent μOPs in the meantime. When a μOP is ready to be executed, the scheduler schedules them for execution. The results of the execution are placed in the corresponding registers, load buffer entries, or store buffer entries. When the next μOP in order is marked as finished, it is retired, and the buffered results are committed and become architectural.

As software is rarely purely linear, the processor has to either stall execution until a (conditional) branch is resolved or speculate on the most likely outcome and start executing along the predicted path. The results of those predicted instructions are placed in the ROB until the prediction has been verified. In the case where the prediction was correct, the instructions are retired in order. If the prediction was wrong, the processor reverts all architectural changes, flushes the pipeline and the ROB but does not revert any microarchitectural state changes, *i.e.*, loading data into the cache or TLB. Similarly, when an interrupt occurs, operations already executed out of order must be flushed from the ROB. We refer to instructions that have been executed speculatively or out-of-order but were never committed as *transient instructions* [8, 46, 53]. Spectre-type attacks [8, 9, 32, 45, 46, 48, 56] exploit the transient execution of instructions before the misprediction by one of the processor's prediction mechanisms is discovered. Meltdown-type attacks [3, 8, 37, 38, 45, 53, 73, 77, 80] exploit the transient execution of instructions before an interrupt or fault is handled.

2.3 Store Buffer

To interact with the memory subsystem (and also to hide some of the latency), modern CPUs have store and load buffers (also called memory order buffer [42]) which act as a queue. The basic mechanism is that the load buffer contains requests for data fetches from memory, while the store buffer contains requests for data writes to memory.

As long as the store buffer is not exhausted, memory stores are simply enqueued to the store buffer in the order they appear in the execution stream, *i.e.*, directly linked to a ROB entry. This allows the CPU to continue executing instructions from the current execution stream, without having to wait for the actual write to finish. This optimization makes sense, as writes in many cases do not influence subsequent instructions, *i.e.*, only loads to the same address are affected. Meanwhile, the store buffer asynchronously processes the stores, ensuring that the stores are written to memory. Thus, the store buffer avoids that the CPU has to stall while waiting for the memory subsystem to finish the write. At the same time, it guarantees that writes reach the memory subsystem in order, despite out-of-order execution. Figure 1 illustrates the role of the store buffer, as a queue between the store-data execution unit and the memory subsystem, *i.e.*, the L1 data cache.

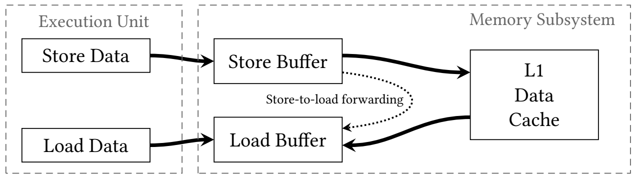


Figure 1: **A store operation stores the data in the store buffer before it is written to the L1 data cache. Subsequent loads can be satisfied from the store buffer if the data is not yet in the L1 data cache. This is called store-to-load forwarding.**

For every store operation that is added to the ROB, an entry is allocated in the store buffer. This entry requires both the virtual and physical address of the target. Only if there is no free entry in the store buffer, the frontend stalls until there is an empty slot available in the store buffer again [36]. Otherwise, the CPU can immediately continue adding subsequent instructions to the ROB and execute them out of order. On Intel CPUs, the store buffer has up to 56 entries [36].

According to Intel patents, the store buffer consists of two separate buffers: the *Store Address Buffer* and the *Store Data Buffer* [1, 2]. The store instruction is decoded into two μOPs, one for storing the address and one for storing data. Those two instructions can execute in either order, depending on which is ready first.

Although the store buffer hides the latency of stores, it also increases the complexity of loads. Every load has to search the store buffer for pending stores to the same address in parallel to the regular L1 lookup. If the address of a load matches the address of a preceding store, the value can be directly used from the store-buffer entry. This optimization for subsequent loads is called store-to-load forwarding [31].

Depending on the implementation of the store buffer, there are various ways of implementing such a search required for store-to-load forwarding, e.g., using content-addressable memory [81]. As loads and stores on x86 do not have to be aligned, a load can also be a partial match of a preceding store. Such a load with a partial match of a store-buffer entry can either stall, continue with stale data, or be resolved by the CPU by combining values from the store buffer and the memory [81].

Moreover, to speed up execution, the CPU might wrongly predict that values should be fetched from memory although there was a previous store, but the target of the previous store is not yet resolved. As a result, the processor can continue transient execution with wrong values, *i.e.*, stale values from memory instead of the recently stored value. This type of misprediction was exploited in Spectre v4 (Speculative Store Bypass) [32], also named Spectre-STL [8].

To speed up store-to-load forwarding, the processor might speculate that a load matches the address of a subsequent store if only the least significant 12 bits match [81]. This performance optimization can further reduce the latency of loads, but also leaks information across hyperthreads [74]. Furthermore, a similar effect also exists if the least significant 20 bits match [42].

2.4 Address Translation

Memory isolation is the basis of modern operating system security. For this purpose, processors support virtual memory as an abstraction and isolation mechanism. Processes work on virtual addresses instead of physical addresses and can architecturally not interfere with each other unintentionally, as the virtual address spaces are largely non-overlapping. The processor translates virtual addresses to physical addresses through a multi-level page translation table. The location of the translation table is indicated by a dedicated register, e.g., CR3 on Intel architectures. The operating system updates the register upon context switch with the physical address of the top-level translation table of the next process. The translation table entries keep track of various properties of the virtual memory region, e.g., user-accessible, read-only, non-executable, and present.

Translation Lookaside Buffer (TLB). The translation of a virtual to a physical address is time-consuming as the translation tables are stored in physical memory. On modern processors, the translation is required even for L1 cache accesses. Hence, the translation must be faster than the full L1 access, e.g., 4 cycles on recent Intel processors. Caching translation tables in regular data caches [35] is not sufficient. Therefore, processors have smaller special caches, translation-lookaside buffers (TLBs) to cache page table entries.

2.5 Address Space Layout Randomization

To exploit a memory corruption bug, an attacker often requires knowledge of addresses of specific data. To impede such attacks, different techniques like address space layout randomization (ASLR), non-executable stacks, and stack canaries have been developed. KASLR extends ASLR to the kernel, randomizing the offsets where code, data, drivers, and other mappings are located on every boot. The attacker then has to guess the location of (kernel) data structures, making attacks harder.

The double page fault attack by Hund et al. [33] breaks KASLR. An unprivileged attacker accesses a kernel memory location and triggers a page fault. The operating system handles the page fault interrupt and hands control back to an error handler in the user program. The attacker now measures how much time passed since triggering the page fault. Even though the kernel address is inaccessible to the user, the address translation entries are copied into the TLB. The attacker now repeats the attack steps, measuring the execution time of a second page fault to the same address. If the memory location is valid, the handling of the second page fault will take less time as the translation is cached in the TLB. Thus, the attacker learns whether a memory location is valid even though the address is inaccessible to user space.

The same effect has been exploited by Jang et al. [43] in combination with Intel TSX. Intel TSX extends the x86 instruction set with support for hardware transactional memory via so-called TSX transactions. A TSX transaction is aborted without any operating system interaction if a page fault occurs within it. This reduces the noise in the timing differences that was present in the attack by Hund et al. [33] as the page fault handling of the operating system is skipped. Thus, the attacker learns whether a kernel memory location is valid with almost no noise at all.

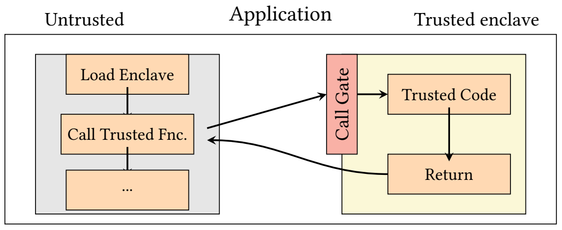


Figure 2: **In the SGX model, applications consist of an untrusted host application and a trusted enclave. The hardware prevents any direct access to the enclave code or data. The untrusted part uses the `EENTER` instruction to call enclave functions that are exposed by the enclave.**

The prefetch side channel presented by Gruss et al. [24] exploits the software prefetch instruction. The execution time of the instruction is dependent on the translation cache that holds the right entry. Thus, the attacker not only learns whether an inaccessible address is valid but also the corresponding page size.

2.6 Intel SGX

As computer usage has changed over the past decades, the need for a protected and trusted execution mechanism has developed. To protect trusted code, Intel introduced an instruction-set extension starting with the Skylake microarchitecture, called Software Guard Extension (SGX) [35]. SGX splits applications into two code parts, a trusted and an untrusted part. The trusted part is executed within a hardware-backed enclave. The processor guarantees that memory belonging to the enclave cannot be accessed by anyone except the enclave itself, not even the operating system. The memory is encrypted and, thus, also cannot be read directly from the DRAM module. Beyond this, there is no virtual memory isolation between trusted and untrusted part. Consequently, the threat model of SGX assumes that the operating system, other applications, and even the remaining hardware might be compromised or malicious. However, memory-safety violations [49], race conditions [79], or side channels [7, 72] are considered out of scope.

The untrusted part can only enter the enclave through a defined interface which is conceptually similar to system calls. After the trusted execution, the result of the computation, as well as the control flow, is handed back to the calling application. The process of invoking a trusted enclave function is illustrated in Figure 2. To enable out-of-the-box data sharing capabilities, the enclave has full access to the entire address space of the host. As this protection is not symmetric, it gives rise to enclave malware [71].

3 ATTACK PRIMITIVES

In this section, we introduce the three basic mechanisms for our attacks. First, *Data Bounce*, which exploits that stores to memory are forwarded even if the target address of the store is inaccessible to the user. We use *Data Bounce* to break both user and kernel space ASLR (cf. Section 4). Second, we exploit interactions between *Data Bounce* and the TLB in Fetch+Bounce. Fetch+Bounce enables attacks on the kernel on a page-level granularity, similar to controlled-channel attacks [84], page-cache attacks [20], TLBleed [16], and DRAMA [65] (cf. Section 5). Third, we augment Fetch+Bounce with

```
① mov (0) → $dummy
② mov $x → (p)
③ mov (p) → $value
④ mov ($mem + $value * 4096) → $dummy
```

Figure 3: ***Data Bounce* writes a known value to an accessible or inaccessible memory location, reads it back, encodes it into the cache, and finally recovers the value using a Flush+Reload attack. If the recovered value matches the known value, the address is backed by a physical page.**

speculative execution in Speculative Fetch+Bounce. Speculative Fetch+Bounce leads to arbitrary data leakage from memory (cf. Section 6).

As described in Section 2.3, unsuccessful or incorrect address matching in the store-to-load forwarding implementation can enable different attacks. For our attacks, we focus solely on the case where the address matching in the store-to-load forwarding implementation is successful and correct. We exploit store-to-load forwarding in the case where the address of the store and load are *exactly the same*, *i.e.*, we do not rely on any misprediction or aliasing effects.

3.1 *Data Bounce*

Our first attack primitive, *Data Bounce*, exploits the property of the store buffer that the full physical address is required for a valid entry. Although the store-buffer entry is already reserved in the ROB, the actual store can only be forwarded if the virtual and physical address of the store target are known [36].

Thus, stores can only be forwarded if the physical address of the store target can be resolved. As a consequence, virtual addresses without a valid mapping to physical addresses cannot be forwarded to subsequent loads. The basic idea of *Data Bounce* is to check whether a data write is forwarded to a data load from the same address. If the store-to-load forwarding is successful for a chosen address, we know that the chosen address can be resolved to a physical address. If done naïvely, such a test would destroy the currently stored value at the chosen address due to the write if the address is writable. Thus, we only test the store-to-load forwarding for an address in the transient-execution domain, *i.e.*, the write is never committed architecturally.

Figure 3 illustrates the basic principle of *Data Bounce*. First, we start transient execution. The easiest way is by generating a fault (①) and catching it (e.g., with a signal handler) or suppressing it (e.g., using Intel TSX). Alternatively, transient execution can be induced through speculative execution using a misspeculated branch [46], call [46], or return [48, 56]. For a chosen address p, we store any chosen value x using a simple data store operation (②). Subsequently, we read the value stored at address p (③) and encode it in the cache (④) in the same way as with Meltdown [53]. That is, depending on the value read from p, we access a different page of the contiguous memory *mem*, resulting in the respective page being cached. Using a straightforward Flush+Reload attack on the 256 pages of *mem*, the page with the lowest access time (*i.e.*, the cached page) directly reveals the value read from p.

We can then distinguish two different cases as follows.

Store-to-load forwarding. If the value read from p is x, *i.e.*, , the x-th page of *mem* is cached, the store was forwarded to the load. Thus, we know that p is backed by a physical page. The choice of the value x is of no importance for *Data Bounce*. Even in the unlikely case that p already contains the value x and the CPU reads the stale value from memory instead of the previously stored value x, we still know that p is backed by a physical page.

No store-to-load forwarding. If no page of *mem* is cached, the store was not forwarded to the subsequent load. This can have either a temporary reason or a permanent reason. If the virtual address is not backed by a physical page, the store-to-load forwarding always fails, *i.e.*, even retrying the experiment will not be successful. Different reasons to not read the written value back are, e.g., interrupts (context switches, hardware interrupts) or errors in distinguishing cache hits from cache misses (e.g., due to power scaling). However, we found that if *Data Bounce* fails multiple times when repeated for *addr*, it is almost certain that *addr* is not backed by a physical page.

In summary, if a value "bounces back" from a virtual address, the virtual address must be backed by a physical page. This effect can be exploited within the virtual address space of a process, e.g., to break ASLR in a sandbox (cf. Section 4.3). On CPUs where Meltdown is mitigated in hardware, KAISER [22] is not enabled, and the kernel is again mapped in user space [11]. In this case, we can also apply *Data Bounce* to kernel addresses. Even though we cannot *access* the data stored at the kernel address, we are still able to detect whether a particular kernel address is backed by a physical page. Thus, *Data Bounce* can still be used to break KASLR (cf. Section 4.1) on processors with in-silicon patches against Meltdown.

3.2 Fetch+Bounce

Our second attack primitive, Fetch+Bounce, augments *Data Bounce* with an additional interaction effect of the TLB and the store buffer. With this combination, it is also possible to detect the recent usage of physical pages.

Data Bounce is a very reliable side channel, making it is easy to distinguish valid from invalid addresses, *i.e.*, whether a virtual page is backed by a physical page. Additionally, the success rate (*i.e.*, how often *Data Bounce* has to be repeated) for valid addresses directly depends on which translations are stored in the TLB. With Fetch+Bounce, we further exploit this TLB-related side-channel information by analyzing the success rate of *Data Bounce*.

The store buffer requires the physical address of the store target (cf. Section 2.3). If the translation from virtual to physical address for the target address is not cached in the TLB, the store triggers a page-table walk to resolve the physical address. On our test machines, we observed that in this case the store-to-load forwarding fails once, *i.e.*, as the physical address of the store is not known, it is not forwarded to the subsequent load. In the other case, when the physical address is already known to the TLB, the store-to-load forwarding succeeds immediately.

With Fetch+Bounce, we exploit that *Data Bounce* succeeds immediately if the mapping for this address is already cached in the

```
①  for retry = 0...2
      mov $x → (p)
②     mov (p) → $value
      mov ($mem + $value * 4096) → $dummy
③     if flush_reload($mem + $x * 4096) then break
```

Figure 4: **Fetch+Bounce repeatedly executes *Data Bounce*. If *Data Bounce* is successful on the first try, the address is in the TLB. If it succeeds on the second try, the address is valid but not in the TLB.**

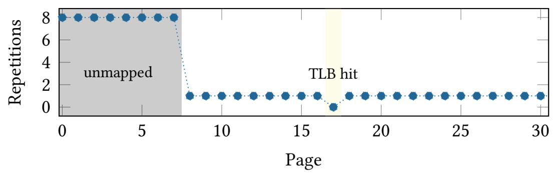


Figure 5: **Mounting Fetch+Bounce on a virtual memory range allows to clearly distinguish mapped from unmapped addresses. Furthermore, for every page, it allows to distinguish whether the address translation is cached in the TLB.**

TLB. Figure 4 shows how Fetch+Bounce works. The basic idea is to repeat *Data Bounce* (②) multiple times (①). There are 3 possible scenarios, which are also illustrated in Figure 5.

TLB Hit. If the address of the store is in the TLB, *Data Bounce* succeeds immediately, and the loop is aborted (③). Thus, `retry` is 0 after the loop.

TLB Miss. If the address of the store is not in the TLB, *Data Bounce* fails in the first attempt, as the physical address needs to be resolved before store-to-load forwarding. However, in this case, *Data Bounce* succeeds in the second attempt (*i.e.*, , `retry` is 1).

Invalid Address. If the address is invalid, `retry` is larger than 1. As only valid address are stored in the TLB [43], and the store buffer requires a valid physical address, store-to-load forwarding can never succeed. The higher `retry`, the (exponentially) more confidence is gained that the address is indeed not valid.

As *Data Bounce* can be used on inaccessible addresses (e.g., kernel addresses), this also works for Fetch+Bounce. Hence, with Fetch+Bounce it is possible to deduce for any virtual address whether it is currently cached in the TLB. The only requirement for the virtual address is that it is mapped to the attacker's address space.

Fetch+Bounce is not limited to the data TLB (dTLB), but can also leak information from the instruction TLB (iTLB). Thus, in addition to recent data accesses, it is also possible to detect which code pages have been executed recently. Again, this also works for inaccessible addresses, e.g., kernel memory.

Moreover, Fetch+Bounce cannot only be used to check whether a (possibly) inaccessible address is in the TLB but also force such an address into the TLB. While this effect might be exploitable on its own, we do not further investigate this side effect. For a real-world attack (cf. Section 5) this is an undesired side effect, as every

```
if (x < len(array))
    y = kernel[array[x] * 4096]
```

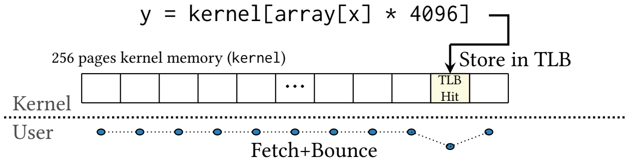


Figure 6: **Speculative Fetch+Bounce allows an attacker to use Spectre gadgets to leak data from the kernel, by encoding them in the TLB. The advantage over regular Spectre attacks is that no shared memory is required, gadgets are simpler as an attacker does not require control of the array base address but only over `x`. All cache-based countermeasures are circumvented.**

measurement with Fetch+Bounce destroys the information. Thus, to repeat Fetch+Bounce for one address, we must evict the TLB in between, e.g., using the strategy proposed by Gras et al. [16].

3.3 Speculative Fetch+Bounce

Our third attack primitive, Speculative Fetch+Bounce, augments Fetch+Bounce with transient-execution side effects on the TLB. The TLB is also updated during transient execution [70]. That is, we can even observe *transient* memory accesses with Fetch+Bounce.

As a consequence, Speculative Fetch+Bounce is a novel way to exploit Spectre. Instead of using the cache as a covert channel in a Spectre attack, we leverage the TLB to encode the leaked data. The advantage of Speculative Fetch+Bounce over the original Spectre attack is that there is no requirement for shared memory between user and kernel space. The attacker only needs control over `x` to leak arbitrary memory contents from the kernel. Figure 6 illustrates the encoding of the data, which is similar to the original Spectre attack [46]. Depending on the value of the byte to leak, we access one out of 256 pages. Then, Fetch+Bounce is used to detect which of the pages has a valid translation cached in the TLB. The cached TLB entry directly reveals the leaked byte.

3.4 Performance and Accuracy

All of the 3 attack primitives work on a page-level granularity, *i.e.*, 4 kB. This limit is imposed by the underlying architecture, as virtual-to-physical mappings can only be specified at page granularity. Consequently, TLB entries also have a page granularity. Hence, the spatial accuracy is the 4 kB page size, which is the same on all CPUs.

While the *spatial* accuracy is always the same, the *temporal* accuracy varies between microarchitectures and implementations. On the i9-9900K, we measured the time of *Data Bounce* over 1 000 000 repetitions and it takes on average 560 cycles per execution. In this implementation, we use Intel TSX to suppress the exception. When resorting to a signal handler for catching exceptions instead of Intel TSX, one execution of *Data Bounce* takes on average 2300 cycles.

Fetch+Bounce and Speculative Fetch+Bounce do not require any additional active part in the attack. They execute *Data Bounce* 3 times, thus the execution time is exactly three times higher than the execution time of *Data Bounce*.

Table 1: **Environments where we evaluated *Data Bounce*, Fetch+Bounce, and Speculative Fetch+Bounce.**

Environment	CPU	*Data Bounce*	Fetch+Bounce	Speculative Fetch+Bounce
Lab	Pentium 4 531	✓	✗	✗
Lab	i5-3230M	✓	✓	✓
Lab	i7-4790	✓	✓	✓
Lab	i7-6600U	✓	✓	✓
Lab	i7-6700K	✓	✓	✓
Lab	i7-8565U	✓	✓	✓
Lab	i7-8650U	✓	✓	✓
Lab	i9-9900K	✓	✓	✓
Lab	E5-1630 v4	✓	✓	✓
Cloud	E5-2650 v4	✓	✓	✓

Data Bounce has the huge advantage that there are no false positives. Store-to-load forwarding does not work for invalid addresses. Additionally, Flush+Reload when applied to individual pages does not have false positives, as the prefetcher on Intel CPUs cannot cross page boundaries [35]. Thus, if a virtual address is not backed by a physical page, *Data Bounce* never reports this page as mapped.

The number of false negatives reported by *Data Bounce* is also negligible. We do not exploit any race condition [53, 77] or aliasing effects [74] but rather a missing permission check. Hence, the store-to-load forwarding works reliably as expected [81]. This can also be seen in the real-world attacks (cf. Section 4), where the F1-score, *i.e.*, the harmonic average of precision and recall, is almost always perfect. We can conclude that *Data Bounce* is a highly practical side-channel attack with perfect precision and recall.

3.5 Environments

We evaluated *Data Bounce*, Fetch+Bounce, and Speculative Fetch+Bounce on multiple Intel CPUs. All attack primitives worked on all tested CPUs, which range from the Ivy Bridge microarchitecture (released 2012) to Whiskey Lake and Coffe Leak R (both released end of 2018). *Data Bounce* even works on Pentium 4 Prescott CPUs (released 2004). Table 1 contains the complete list of CPUs we used to evaluate the attacks.

The primitives are not limited to the Intel Core microarchitecture, but also work on the Intel Xeon microarchitecture. Thus, these attacks are not limited to consumer devices, but can also be used in the cloud. Furthermore, the attack primitives even work on CPUs which have silicon fixes for Meltdown and Foreshadow, such as the i7-8565U and i9-9900K [11].

For AMD, and ARM CPUs, we were not able to reproduce any of our attack primitives, limiting the attacks to Intel CPUs.

4 ATTACKS ON ASLR

In this section, we evaluate our attack on ASLR in different scenarios. As *Data Bounce* can reliably detect whether a virtual address is backed by a physical page, it is well suited for breaking all kinds of ASLR. In Section 4.1, we show that *Data Bounce* is the fastest way and most reliable side-channel attack to break KASLR on Linux, and Windows, both in native environments as well as in virtual machines. In Section 4.2, we demonstrate that *Data Bounce* also works from within SGX enclaves, allowing enclaves to de-randomize the host application. In Section 4.3, we describe that *Data Bounce* can even be mounted from JavaScript to break ASLR of the browser.

4.1 Breaking KASLR

In this section, we show that *Data Bounce* can reliably break KASLR. We evaluate the performance of *Data Bounce* in three different KASLR breaking attacks. First, we de-randomize the kernel base address. Second, we de-randomize the direct-physical map. Third, we find and classify modules based on detected size.

De-randomizing the Kernel Base Address. Jang et al. [43] state that the kernel text segment is mapped at a 16 MB boundary somewhere in the `0xffffffff80000000` - `0xffffffffc0000000` range. Given that range, the maximum kernel size is 1 GB. Combined with the 16 MB alignment, the kernel can only be mapped at one of 64 possible offsets, *i.e.*, 6 bits of entropy. This contradicts the official documentation in The Linux Kernel Archive [50], which states that the kernel text segment is mapped somewhere in the `0xffffffff80000000 - 0xffffffff9fffffff` range, giving us a maximum size of 512 MB. Our experiments show that Jang et al. [43] is correct with the address range, but that the kernel is aligned at a 8 times finer 2 MB boundary. We verified this by checking `/proc/kallsyms` after multiple reboots. With a kernel base address range of 1 GB and a 2 MB alignment, we get 9 bits of entropy, allowing the kernel to be placed at one of 512 possible offsets.

Using *Data Bounce*, we now start at the lower end of the address range and test all of the 512 possible offsets. If the kernel is mapped at a tested location, we will observe a cache hit. In our experiments, we see the first cache hit at exactly the same address given by `/proc/kallsyms`. Additionally, we see cache hits on all 2 MB aligned pages that follow. This indicates a 1 MB aliasing effect, supporting the claim made by Islam et al. [42]. We only observe the hit on 2 MB aligned pages that follow, as this is our step size.

Table 2 shows the performance of *Data Bounce* in de-randomizing kernel ASLR. We evaluated our attack on both an Intel Skylake i7-6600U (without KPTI) and a new Intel Coffee Lake i9-9900K that already includes fixes for Meltdown [53] and Foreshadow [77]. We evaluated our attack on both Windows and Linux, achieving similar results although the ranges differ on Windows. On Windows, the kernel also starts at a 2 MB boundary, but the possible range is `0xfffff80000000000` - `0xfffff80400000000`, which leads to 8192 possible offsets, *i.e.*, 13 bits of entropy [43].

For the evaluation, we tested 10 different randomizations (*i.e.*, 10 reboots), each one 100 times, giving us 1000 samples. For evaluating the effectiveness of our attack, we use the F1-score. On the i7-6600U and the i9-9900K, the F1-score for finding the kernel ASLR offset is 1 when testing every offset a single time, indicating that we always find the correct offset. In terms of performance, we outperform the previous state of the art [43] even though we have an 8 times larger search space. Furthermore, to evaluate the performance on a larger scale, we tested a single offset 100 million times. In that test, the F1-score was 0.9996, showing that *Data Bounce* virtually always works. The few misses that we observe are possibly due to the store buffer being drained or that our test program was interrupted.

De-randomizing the Direct-physical Map. In Section 2.5, we discussed that the Linux kernel has a direct-physical map that maps

Table 2: **Evaluation of *Data Bounce* in finding the kernel base address and direct-physical map, and kernel modules. Number of retries refers to the maximum number of times an offset is tested and number of offsets denotes the maximum number of offsets that need to be tried.**

Processor	Target	#Retries	#Offsets	Time	F1-Score
Skylake (i7-6600U)	base	1	512	72 µs	1
	direct-physical	3	64000	13.648 ms	1
	module	32	262144	1.713 s	0.98
Coffee Lake (i9-9900K)	base	1	512	42 µs	1
	direct-physical	3	64000	8.61 ms	1
	module	32	262144	1.33 s	0.96

the entire physical memory into the kernel virtual address space. To impede attacks that require knowledge about the map placement in memory, the map is placed at a random offset within a given range at every boot. According to The Linux Kernel Archive [50], the address range reserved for the map is `0xffff888000000000-0xffffc87fffffffff`, *i.e.*, a 64 TB region. The Linux kernel source code indicates that the map is aligned to a 1 GB boundary. This gives us 2^{16} possible locations.

Using this information, we now use *Data Bounce* to recover the location of the direct-physical map. Table 2 shows the performance of the recovery process. For the evaluation, we tested 10 different randomizations of the kernel (*i.e.*, 10 reboots) and for each offset, we repeated the detection 100 times. On the Skylake i7-6600U, we were able to recover the offset in under 14 ms if KPTI is disabled. On the Coffee Lake i9-9900K, where KPTI is no longer needed, we were able to do it in under 9 ms.

Finding and Classifying Kernel Modules. The kernel reserves 1 GB for modules and loads them at 4 kB-aligned offset. In a first step, we can use *Data Bounce* to detect the location of modules by iterating over the search space in 4 kB steps. As kernel code is always present and modules are separated by unmapped addresses, we can detect where a module starts and ends. In a second step, we use this information to estimate the size of all loaded kernel modules. The world-readable */proc/modules* file contains information on modules, including name, size, number of loaded instances, dependencies on other modules, and load state. For privileged users, it additionally provides the address of the module. We correlate the size from */proc/modules* with the data from our *Data Bounce* attack and can identify all modules with a unique size. On the i7-6600U, running Ubuntu 18.04 with kernel version 4.15.0-47, we have a total of 26 modules with a unique size. On the i9-9900K, running Ubuntu 18.10 with kernel version 4.18.0-17, we have a total of 12 modules with a unique size. Table 2 shows the accuracy and performance of *Data Bounce* for finding and classifying those modules.

The Strange Case of Non-Canonical Addresses. For valid kernel addresses, there are no false positives with *Data Bounce*. Interestingly, when used on a non-canonical address, *i.e.*, an address where the bits 47 to 63 are not all '0' or '1', *Data Bounce* reports this address to be backed by a physical page. However, these addresses are invalid by definition and can thus never refer to a physical address [35]. We guess that there might be a missing check in the store-to-load forwarding unit, which allows non-canonical addresses to enter the store buffer and be forwarded to subsequent loads despite not having a physical address associated. Although the possibility of such a behaviour is documented [30], it is still unexpected, and future work should investigate whether it could lead to security problems.

Table 3: **Comparison of microarchitectural attacks on KASLR. Of all known attacks, *Data Bounce* is by far the fastest and in contrast to all other attacks has no requirements.**

Attack	Time	Accuracy	Requirements
Hund et al. [33]	17 s	96 %	-
Gruss et al. [24]	500 s	N/A	cache eviction
Jang et al. [43]	5 ms	100 %	Intel TSX
Evtyushkin et al. [12]	60 ms	N/A	BTB reverse engineering
Data Bounce (our attack)	**42 µs**	100 %	-

Comparison to Other Side-Channel Attacks on KASLR.. Previous microarchitectural attacks on ASLR relied on address-translation caches [16, 24, 33, 43] or branch-predictor states [12, 13]. We compare *Data Bounce* against previous attacks on kernel ASLR [12, 24, 33, 43].

Table 3 shows that our attack is the fastest attack that works on any Intel x86 CPU. In terms of speed and accuracy, *Data Bounce* is similar to the methods proposed by Jang et al. [43] and Evtyushkin et al. [12]. However, one advantage of *Data Bounce* is that it does not rely on CPU extensions, such as Intel TSX, or precise knowledge of internal data structures, such as the reverse-engineering of the branch-target buffer (BTB). In particular, the attack by Jang et al. [43] is only applicable to selected CPUs starting from the Haswell microarchitecture, *i.e.*, it cannot be used on any CPU produced earlier than 2013. Similarly, Evtyushkin et al. [12] require knowledge of the internal workings of the branch-target buffer, which is not known for any microarchitecture newer than Haswell. *Data Bounce* works regardless of the microarchitecture, which we have verified by successfully running it on microarchitectures starting from Pentium 4 Prescott (released 2004) to Whiskey Lake and Coffee Lake R (both released end of 2018) (cf. Table 1).

4.2 Inferring ASLR from SGX

Data Bounce does not only work for privileged addresses (cf. Section 4.1), it also works for otherwise inaccessible addresses, such as SGX enclaves. We demonstrate 3 different scenarios of how *Data Bounce* can be used in combination with SGX.

Data Bounce *from Host to Enclave.* With *Data Bounce*, it is possible to detect enclaves in the EPC (Enclave Page Cache), the encrypted and inaccessible region of physical memory reserved for Intel SGX. Consequently, we can de-randomize the virtual addresses used by SGX enclaves.

However, this scenario is artificial, as an application has more straightforward means to determine the mapped pages of its enclave. First, applications can simply check their virtual address space mapping, e.g., using `/proc/self/maps` in Linux. Second, reading from EPC pages always returns '-1' [34], thus if a virtual address is also not part of the application, it is likely that it belongs to an enclave. Thus, an application can simply use TSX or install a signal handler to probe its address space for regions which return '-1' to detect enclaves.

For completeness, we also evaluated this scenario. In our experiments, we successfully detected all pages mapped by an SGX enclave in our application using *Data Bounce*. Like with KASLR before, we had no false positives, and the accuracy was 100 %.

Data Bounce *from Enclave to Host.* While the host application cannot access memory of an SGX enclave, the enclave has full access to the virtual memory of its host. Schwarz et al. [71] showed that this asymmetry allows building enclave malware by overwriting the host application's stack for mounting a return-oriented programming exploit. One of the primitives they require is a possibility to scan the address space for mapped pages without crashing the enclave. While this is trivial for normal applications using syscalls (e.g., by abusing the `access` syscall, or by registering a user-space signal handler), enclaves cannot rely on such techniques as they cannot execute syscalls. Thus, Schwarz et al. propose *TAP*, a primitive relying on TSX to test whether an address is valid without the risk of crashing the enclave.

The same behavior can also be achieved using *Data Bounce* without having to rely on TSX. By leveraging speculative execution to induce transient execution, not a single syscall is required to mount *Data Bounce*. As the `rdtsc` instruction cannot be used inside enclaves, we use the same counting-thread technique as Schwarz et al. [72]. The resolution of the counting thread is high enough to mount a Flush+Reload attack inside the enclave. In our experiments, a simple counting thread achieved 0.92 increments per cycle on a Skylake i7-8650U, which is sufficient to mount a reliable attack.

With 290 MB/s, the speed of *Data Bounce* in an enclave is a bit slower than the speed of *TAP* [71]. The reason is that *Data Bounce* requires a timer to mount a Flush+Reload attack, whereas *TAP* does not require any timing or other side-channel information. Still, *Data Bounce* is a viable alternative to the TSX-based approach for detecting pages of the host application from within an SGX enclave, in particular as many processors do not support TSX.

Data Bounce *from Enclave to Enclave.* Enclaves are not only isolated from the operating system and host application, but enclaves are also isolated from each other. Thus, *Data Bounce* can be used from a (malicious) enclave to infer the address-space layout of a different, inaccessible enclave.

We evaluated the cross-enclave ASLR break using two enclaves, one benign and one malicious enclave, started in the same application. Again, to get accurate timestamps for Flush+Reload, we used a counting thread.

As the enclave-to-enclave scenario uses the same code as the enclave-to-host scenario, we also achieve the same performance.

4.3 Breaking ASLR from JavaScript

Data Bounce cannot only be used from unprivileged native applications but also in JavaScript to break ASLR in the browser. In this section, we evaluate the performance of *Data Bounce* from JavaScript running in a modern browser. Our evaluation was done on Google Chrome 70.0.3538.67 (64-bit) and Mozilla Firefox 66.0.2 (64-bit).

There are two main challenges for mounting *Data Bounce* from JavaScript. First, there is no high-resolution timer available. Therefore, we need to build our own timing primitive. Second, as there

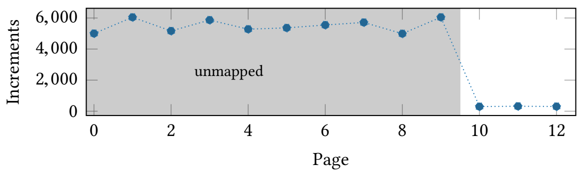


Figure 7: ***Data Bounce* with Evict+Reload in JavaScript clearly shows whether an address (relative to a base address) is backed by a physical page and thus valid.**

is no flush instruction in JavaScript, Flush+Reload is not possible. Thus, we have to resort to a different covert channel for bringing the microarchitectural state to the architectural state.

Timing Primitive. To measure timing with a high resolution, we rely on the well-known use of a counting thread in combination with shared memory [17, 69]. As Google Chrome has re-enabled `SharedArrayBuffers` in version 67[1], we can use the existing implementations of such a counting thread.

In Google Chrome, we can also use `BigUint64Array` to ensure that the counting thread does not overflow. This improves the measurements compared to the `Uint32Array` used in previous work [17, 69] as the timestamp is increasing strictly monotonically. In our experiments, we achieve a resolution of 50 ns in Google Chrome, which is sufficient to distinguish a cache hit from a miss.

Covert Channel. As JavaScript does not provide a method to flush an address from the cache, we have to resort to eviction as shown in previous work [17, 46, 61, 69, 78]. Thus, our covert channel from the microarchitectural to the architectural domain, *i.e.*, the decoding of the leaked value which is encoded into the cache, uses Evict+Reload instead of Flush+Reload.

For the sake of simplicity, we can also just access an array which has a size 2-3 times larger than the last-level cache to ensure that the array is evicted from the cache. For our proof-of-concept, we use this simple approach as it is robust and works for the attack. While the performance increases significantly when using targeted eviction, we would require 256 eviction sets. Building them would be time consuming and prone to errors.

Illegal Access. In JavaScript, we cannot access an inaccessible address architecturally. However, as JavaScript is compiled to native code using a just-in-time compiler in all modern browsers, we can leverage speculative execution to prevent the fault. Hence, we rely on the same code as Kocher et al. [46] to speculatively access an out-of-bounds index of an array. This allows to iterate over the memory (relative from our array) and detect which pages are mapped and which pages are not mapped.

Full Exploit. When putting everything together, we can distinguish for every location relative to the start array whether it is backed by a physical page or not. Due to the limitations of the JavaScript sandbox, especially due to the slow cache eviction, the speed is orders of magnitude slower than the native implementation, as it can be seen in Figure 7. Still, we can detect whether a

[1]https://bugs.chromium.org/p/chromium/issues/detail?id=821270

virtual address is backed by a physical page within 450 ms, making *Data Bounce* also realistic from JavaScript.

5 FETCH+BOUNCE

While *Data Bounce* can already be used for powerful attacks, Fetch+Bounce, the TLB-augmented variant of *Data Bounce*, allows for even more powerful attacks as we show in this section. So far, most microarchitectural attacks which can attack the kernel, such as Prime+Probe [68] or DRAMA [65], require at least some knowledge of physical addresses. Since physical address information is not provided to unprivileged application, these attacks either require additional side channels [23, 68] or have to blindly attack targets until the correct target is found [72].

With Fetch+Bounce we can directly retrieve side-channel information for any target virtual address, regardless of whether it can be accessed in the current privilege level or not. In particular, we can detect whether a virtual address has a valid translation in either the iTLB or dTLB. This allows an attacker to infer whether an address was recently used.

Fetch+Bounce can attack both the iTLB and dTLB. Using Fetch+Bounce, an attacker can detect recently accessed *data pages* on the current hyperthread. Moreover, an attacker can also detect *code pages* recently used for instruction execution on the current hyperthread.

As the measurement with Fetch+Bounce results in a valid mapping of the target address, we also require a method to evict the TLB. While this can be as simple as accessing (dTLB) or executing (iTLB) data on more pages than there are entries in the TLB, this is not an optimal strategy. Instead, we rely on the reverse-engineered eviction strategies from Gras et al. [16].

The attack process is the following:

① Build eviction sets for the target address(es)
② Fetch+Bounce on the target address(es) to detect activity
③ Evict address(es) from iTLB and dTLB
④ Goto 2

In Section 5.1, we show that Fetch+Bounce allows an unprivileged application to spy on TSX transactions. In Section 5.2, we demonstrate an attack on the Linux kernel using Fetch+Bounce.

5.1 Breaking the TSX Atomicity

Intel TSX guarantees the atomicity of all instructions and data accesses which are executed inside a TSX transaction. Thus, Intel TSX has been proposed as a security mechanism against side-channel attacks [21, 27].

With Fetch+Bounce, an attacker can break the atomicity of TSX transactions by recovering the TLB state after the transaction. While Intel TSX reverts the effect of all instructions, including the invalidation of modified cache lines, the TLB is not affected. Thus, Fetch+Bounce can be used to detect which addresses are valid in the TLB, and thus infer which addresses have been accessed within a transaction. We verified that this is not only possible after a successful commit of the transaction, but also after a transaction aborted.

As a consequence, an attacker can abort a transaction at an arbitrary point (e.g., by causing an interrupt or a conflict in the cache) and use Fetch+Bounce to detect which pages have been accessed until this point in time. From that, an attacker can learn memory

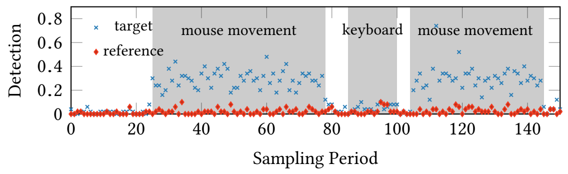


Figure 8: **Mouse movement detection. The mouse movements are clearly detected. The USB keyboard activity does not cause more TLB hits than observed as a baseline.**

access patterns, which should be invisible due to the guarantees provided by Intel TSX.

5.2 Inferring Control Flow of the Kernel

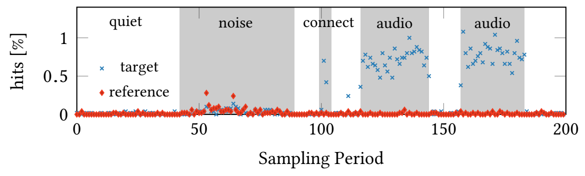
The kernel is a valuable target for attackers, as it processes all user inputs coming from I/O devices. Microarchitectural attacks targeting user input directly in the kernel usually rely on Prime+Probe [59, 61, 67, 68] and thus require knowledge of physical addresses.

With Fetch+Bounce, we do not require knowledge of physical addresses to spy on the kernel. In the following, we show that Fetch+Bounce can spy on any type of kernel activity. We illustrate this with the examples of mouse input and Bluetooth events.

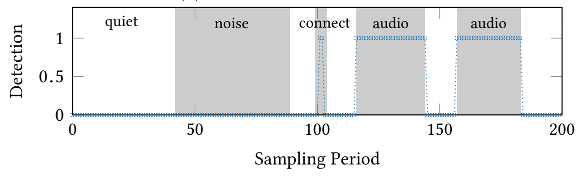
As a simple proof of concept we monitor the first 8 pages of a target kernel module. To obtain a baseline for the general kernel activity, and thus the TLB activity for kernel pages, we also monitor one reference page from a kernel module that is rarely used (in our case `i2c_i801`). By comparing the activity on the 8 pages of the kernel module to the baseline, we can determine whether the kernel module is currently actively used or not. To get the best results we use Fetch+Bounce with both the iTLB and dTLB. This makes the attack independent of the type of activity in the kernel module, *i.e.*, there is no difference if code is executed or data is accessed. Our spy changes its hyperthread after each Fetch+Bounce measurement. This reduces the attack's resolution, but allows it to detect activity on all hyperthreads. For further processing, we sum the resulting TLB hits over a *sampling period* which consists of 5000 measurements, and then apply a basic detection filter to this sum: We calculate the ratio of hits on the target pages and the reference page. If the number of hits on the target pages is above a sanity lower bound and more importantly, above the number of cache hits on the reference page, *i.e.*, above the baseline, then, the page was recently used (cf. Figure 9b).

Detecting User Input. We now investigate how well Fetch+Bounce works for spying on input-handling code in the kernel. While Schwarz et al. [68] attacked the kernel code for PS/2 keyboards and laptops, we target the kernel module for USB human-interface devices. This has the advantage that we can attack input from a large variety of modern USB input devices.

We first locate the kernel module using *Data Bounce* as described in Section 4.1. With 12 pages (kernel 4.15.0), the kernel module does not have a unique size among all modules but is one of only 3. Thus, we can either try to identify the correct module or monitor all of them.



(a) **Fetch+Bounce raw hit rate.**



(b) **Fetch+Bounce detection decisions.**

Figure 9: **Detecting Bluetooth events by monitoring TLB hits via Fetch+Bounce on pages at the start of the `bluetooth` kernel module.**

Figure 8 shows the results of using Fetch+Bounce on a page of the `usbhid` kernel module. It can be clearly seen that mouse movement results in a higher number of TLB hits. USB keyboard input, however, seems to fall below the detection threshold with our simple method. Given this attack's low temporal resolution, repeated accesses to a page are necessary for clear detection. Previous work has shown that such an event trace can be used to infer user input, e.g., URLs [51, 61].

Detecting Bluetooth Events. Bluetooth events can give valuable information about the user's presence at the computer, e.g., connecting (or disconnecting) a device usually requires some form of user interaction. Tools, such as Dynamic Lock on Windows 10 [58], use connect and disconnect events to unlock and lock a computer automatically. Thus, these events are apparently a useful indicator to detect whether the user is currently using the computer. Also, it is useful to monitor these events as a trigger signal for UI redressing attacks.

To spy on these events, we first locate the Bluetooth kernel module using *Data Bounce*. As the Bluetooth module is rather large (134 pages on kernel 4.15.0) and has a unique size, it is easy to distinguish it from other kernel modules.

Figure 9 shows a Fetch+Bounce trace while generating Bluetooth events. While there is a constant noise floor due to TLB collisions (Figure 9a), we can see a clear increase in TLB hits on the target address for every Bluetooth event. After applying our detection filter, we can detect events such as connecting and playing audio over the Bluetooth connection with a high accuracy (Figure 9b).

Our results indicate that the precision of the detection and distinction of events with Fetch+Bounce can be significantly improved. Future work should investigate profiling code pages of kernel modules, similar to previous template attacks [26].

6 LEAKING KERNEL MEMORY

In this section, we present Speculative Fetch+Bounce, a novel covert channel to leak memory using Spectre. Most Spectre attacks, including the original Spectre attack, use the cache as a covert channel

```
if ( index < bounds )
    y = oracle[ data[index] * 4096 ];
```

Listing 1: A simple Spectre-PHT gadget, which allows speculative access of `data` out of bounds and encodes the value in `oracle`.

to encode values leaked from the kernel [9, 32, 45, 46, 48, 56, 60, 70]. Other covert channels for Spectre attacks, such as port contention [6] or AVX [70] have since been presented. However, it is unclear how commonly such gadgets can be found and can be exploited in real-world software.

With Speculative Fetch+Bounce, we show how the TLB effects on the store buffer (cf. Section 5) can be combined with speculative execution to leak kernel data. We show that any cache-based Spectre gadget can be used for Speculative Fetch+Bounce. As the secret-dependent page access also populates the TLB, such a gadget also encodes the information in the TLB. With *Data Bounce*, we can then reconstruct which of the pages was accessed and thus infer the secret.

While at first, the improvements over the original Spectre attack might not be obvious, there are 2 huge advantages.

Advantage 1: It requires less control over the Spectre gadget. First, for Speculative Fetch+Bounce, an attacker requires less control over the Spectre gadget. In a Spectre-PHT (aka Spectre Variant 1) attack, a gadget similar to the one illustrated in Listing 1 is required. There, an attacker requires full control over `index`, and also certain control over `oracle`. Specifically, the base address of `oracle` has to point to user-accessible memory which is shared between attacker and victim. Furthermore, the base address has to either be known or be controlled by the attacker. This limitation potentially reduces the number of exploitable gadgets.

Advantage 2: It requires no shared memory. Second, with Speculative Fetch+Bounce, we get rid of the shared-memory requirement. Especially on modern operating systems, shared memory is a limitation, as these operating systems provide stronger kernel isolation [22]. On such systems, only a few pages are mapped both in user and kernel space, and they are typically inaccessible from the user space. Moreover, the kernel can typically not access user space memory due to supervisor mode access prevention (SMAP). Hence, realistic Spectre attacks have to resort to Prime+Probe [76]. However, Prime+Probe requires knowledge of physical addresses, which is not exposed on modern operating systems.

With Speculative Fetch+Bounce, it is not necessary to have a memory region which is user accessible and shared between user and kernel space. For Speculative Fetch+Bounce, it is sufficient that the base address of `oracle` points to a kernel address which is also mapped in user space. Even in the case of KPTI [55], there are still kernel pages mapped in the user space. On kernel 4.15.0, we identified 65536 such kernel pages (*i.e.*, 256 MB) when KPTI is enabled, and multiple gigabytes when KPTI is disabled. Hence, `oracle` only has to point to any such range of mapped pages. Thus, we expect that there are simpler Specter gadgets which are sufficient to mount this attack.

6.1 Leaking Data

To evaluate Speculative Fetch+Bounce, we use a custom `ioctl` in the Linux kernel containing a Spectre gadget as illustrated in Listing 1. The `oracle` array points to a kernel address which is also mapped in the user space but not user accessible. Furthermore, we can control `index` from user space, *i.e.*, it is provided as an argument to the `ioctl`.

By first providing in-bounds values for `index`, we mistrain the branch predictor in the kernel in-place [8]. Then, by providing an out-of-bounds value for `index`, the gadget encodes the speculatively accessed value in the TLB. Finally, in user space, we use *Data Bounce* to detect which kernel page of `oracle` has a valid TLB entry. The TLB entry directly depends on the secret leaked from the kernel.

We cannot ensure that speculative execution always misspeculates, hence, we have to re-run Speculative Fetch+Bounce multiple times per byte to leak. As with Fetch+Bounce (cf. Section 5), this again requires TLB eviction after every run of Speculative Fetch+Bounce. With this approach, we can reliably leak data from the kernel in the same way and with the same efficiency as in the original Spectre attack [46]. However, we reduced the requirements for an attacker significantly, as our approach also works with active SMAP and there is no need for shared memory.

7 DISCUSSION & RELATED WORK

With *Data Bounce*, we demonstrate a powerful side-channel attack to detect whether a virtual address has a valid mapping by exploiting store-to-load forwarding. While similar attacks are known, *Data Bounce* is both the most reliable and fastest attack so far. The attack which comes closest in terms of reliability and performance is the TSX-based attack by Jang et al. [43]. However, TSX is only supported by 34% of the Core CPUs and 43% of the Xeon CPUs released since 2013, which are currently available for sale [66]. *Data Bounce* does work both with and without TSX. When leveraging TSX for *Data Bounce*, the performance increases by factor 4 which even outperforms Jang et al. [43]. The F1-score, *i.e.*, both precision and recall, are not affected regardless of whether TSX is used or not. As a result, *Data Bounce* is applicable to a much wider range of CPUs, and also in restricted environments such as JavaScript. Thus, *Data Bounce* can also be used in similar scenarios as the JavaScript-based ASLR break by Gras et al. [18].

Gras et al. [16] demonstrated that sharing the TLB across hyperthreads enables eviction-based attacks on the TLB. However, this attack requires to first reverse engineer the TLB for building precise eviction sets of the target mapping. Fetch+Bounce is a similar side-channel attack but does not require knowledge of TLB sets. Fetch+Bounce works directly with the virtual address of the target mapping, thus reducing the noise of addresses mapping to the same set in the TLB. On a high level, Gras et al. [16] showed a Prime+Probe-type attack on the TLB, whereas we show a more precise Evict+Reload-type attack on the TLB.

With Speculative Fetch+Bounce, we show a fast and practical covert channel for Spectre attacks which can replace Flush+Reload in certain scenarios. While other covert channels have been proposed [6, 45, 70, 76], Flush+Reload is still the most reliable covert channel. As Speculative Fetch+Bounce can exploit the same gadgets, and gadgets with fewer requirements, we already know from previous work that such real-world gadgets exist and have been exploitable in the wild [46]. In line with previous Spectre papers, finding new gadgets is an orthogonal problem and thus is not discussed in this paper.

Although the information leakage is caused by the hardware, countermeasures against *Data Bounce* are possible. The basic idea is to not have different security domains in the same address space. Since Meltdown, KAISER [22] was deployed on all modern operating systems to ensure that the kernel is not mounted in the user's address space. This does not only prevent Meltdown but also side-channel attacks on kernel addresses [24, 33, 43] including *Data Bounce*, Fetch+Bounce, and Speculative Fetch+Bounce. Thus, we suggest to unconditionally enable this countermeasure even if the CPU reports that it is not affected by Meltdown. Additionally, we propose to apply the same principle to SGX and sandboxes, similar to site isolation [75]. In general, future hardware and software designs should ensure that different security domains are not shared in the same address space to reduce the attack surface.

While preventing the attacks is possible, detecting the attacks is significantly harder. Depending on the implementation, *Data Bounce* does not require any operating-system interaction at all. For example, when implemented using speculative execution for exception prevention, no syscall is required, and architecturally, no exception is triggered. Thus, *Data Bounce* is completely invisible to the operating system. Several works suggested relying on performance counters to detect ongoing side-channel attacks [10, 25, 29, 40, 63, 86], e.g., by detecting an anomaly in cache misses. However, it is unclear how such detection mechanisms perform in real-world scenarios. Especially for the KASLR break, where the total runtime is only 42 µs, it is questionable whether this is detectable among the average system noise. It is even more questionable whether the system could in time respond to a potential ongoing attack.

8 CONCLUSION

In this paper, we demonstrated *Data Bounce*, a Meltdown-like attack on recent patched CPUs. We showed that correct store-to-load forwarding via the store buffer introduces potent channels to leak data and meta data. We showed that we can break KASLR on fully patched machines in 42 µs. We demonstrated using the same side channel to also reveal the address space of Intel SGX enclaves, and break ASLR from JavaScript. In combination with TLB state changes we were able to break the atomicity of TSX as well as monitor the control flow of the kernel. Finally, we found that Spectre v1 gadgets can also be exploited using *Data Bounce*. We showed that this allows us to leak arbitrary kernel memory in realistic scenarios.

Our work shows that the hardware fixes for Meltdown in Whiskey Lake and Coffe Lake CPUs are clearly not sufficient. We conclude that software-based isolation of user and kernel space should remain enabled even on the most recent processor generations.

ACKNOWLEDGMENTS

We thank Moritz Lipp and Vedad Hadžić from Graz University of Technology and Julian Stecklina from Cyberus Technology for contributing ideas and experiments. This work has been supported by the Austrian Research Promotion Agency (FFG) via the project

ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] Abramson, J. M., Akkary, H., Glew, A. F., Hinton, G. J., Konigsfeld, K. G., and Madland, P. D. Method and apparatus for performing a store operation, Apr. 2002. US Patent 6,378,062.

[2] Abramson, J. M., Akkary, H., Glew, A. F., Hinton, G. J., Konigsfeld, K. G., Madland, P. D., Papworth, D. B., and Fetterman, M. A. Method and apparatus for dispatching and executing a load operation to memory, Feb. 1998. US Patent 5,717,882.

[3] ARM Limited. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism, 2018.

[4] Benger, N., van de Pol, J., Smart, N. P., and Yarom, Y. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In *CHES'14* (2014).

[5] Bernstein, D. J. Cache-Timing Attacks on AES, http://cr.yp.to/antiforgery/cachetiming-20050414.pdf 2004.

[6] Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., and Kurmus, A. SMoTherSpectre: exploiting speculative execution through port contention. *arXiv:1903.01843* (2019).

[7] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT* (2017).

[8] Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv:1811.05441* (2018).

[9] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., and Lai, T. H. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv:1802.09085* (2018).

[10] Chiappetta, M., Savas, E., and Yilmaz, C. Real time detection of cache-based side-channel attacks using hardware performance counters. ePrint 2015/1034, 2015.

[11] Cutress, I. Spectre and Meltdown in Hardware: Intel Clarifies Whiskey Lake and Amber Lake, https://www.anandtech.com/show/13301/spectre-and-meltdown-in-hardware-intel-clarifies-whiskey-lake-and-amber-lake Aug. 2018.

[12] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Jump over aslr: Attacking branch predictors to bypass aslr. In *MICRO* (2016).

[13] Evtyushkin, D., Riley, R., Abu-Ghazaleh, N. C., ECE, and Ponomarev, D. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS'18* (2018).

[14] fG! Measuring OS X Meltdown Patches Performance, https://reverse.put.as/2018/01/07/measuring-osx-meltdown-patches-performance/ Jan 2018.

[15] Fog, A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers, 2016.

[16] Gras, B., Razavi, K., Bos, H., and Giuffrida, C. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium* (2018).

[17] Gras, B., Razavi, K., Bosman, E., Bos, H., and Giuffrida, C. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS* (2017).

[18] Gras, B., Razavi, K., Bosman, E., Bos, H., and Giuffrida, C. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS'17* (2017).

[19] Gruss, D., Hansen, D., and Gregg, B. Kernel isolation: From an academic idea to an efficient patch for every computer. *USENIX ;login* (2018).

[20] Gruss, D., Kraft, E., Tiwari, T., Schwarz, M., Trachtenberg, A., Hennessey, J., Ionescu, A., and Fogh, A. Page cache attacks. In *CCS* (2019).

[21] Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., and Costa, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security Symposium* (2017).

[22] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is Dead: Long Live KASLR. In *ESSoS* (2017).

[23] Gruss, D., Lipp, M., Schwarz, M., Genkin, D., Juffinger, J., O'Connell, S., Schoechl, W., and Yarom, Y. Another Flip in the Wall of Rowhammer Defenses. In *S&P* (2018).

[24] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS* (2016).

[25] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA* (2016).

[26] Gruss, D., Spreitzer, R., and Mangard, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015).

[27] Guan, L., Lin, J., Luo, B., Jing, J., and Wang, J. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *S&P* (2015).

[28] Hansen, D. KAISER: unmap most of the kernel from userspace page table, https://lkml.org/lkml/2017/10/31/884 2017.

[29] Herath, N., and Fogh, A. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In *Black Hat Briefings* (2015).

[30] Hily, S., Zhang, Z., and Hammarlund, P. Resolving false dependencies of speculative load instructions. US Patent 7.603,527, 2009.

[31] Hooker, R. E., and Eddy, C. Store-to-load forwarding based on load/store address computation source information comparisons, 2013. US Patent 8,533,438.

[32] Horn, J. speculative execution, variant 4: speculative store bypass, 2018.

[33] Hund, R., Willems, C., and Holz, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P* (2013).

[34] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference, May 2016. Rev 1.5.

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

[36] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2017.

[37] Intel. Intel Analysis of Speculative Execution Side Channels, https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf July 2018.

[38] Intel. Speculative Execution Side Channel Mitigations, May 2018. Revision 3.0.

[39] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *S&P'15* (2015).

[40] Irazoqui, G., Eisenbarth, T., and Sunar, B. Mascat: Preventing microarchitectural attacks before distribution. In *CODASPY* (2018).

[41] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14* (2014).

[42] Islam, S., Moghimi, A., Bruhns, I., Krebbel, M., Gulmezoglu, B., Eisenbarth, T., and Sunar, B. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. *arXiv:1903.00446* (2019).

[43] Jang, Y., Lee, S., and Kim, T. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS* (2016).

[44] Johnson, K. KVA Shadow: Mitigating Meltdown on Windows, https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/ Mar 2018.

[45] Kiriansky, V., and Waldspurger, C. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).

[46] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *S&P* (2019).

[47] Kocher, P. C. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996).

[48] Koruyeh, E. M., Khasawneh, K., Song, C., and Abu-Ghazaleh, N. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT* (2018).

[49] Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., and Kang, B. B. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security* (2017).

[50] Linux. Complete virtual memory map with 4-level page tables, https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt 2019.

[51] Lipp, M., Gruss, D., Schwarz, M., Bidner, D., Maurice, C., and Mangard, S. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS* (2017).

[52] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium* (2016).

[53] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium* (2018).

[54] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-Level Cache Side-Channel Attacks are Practical. In *S&P* (2015).

[55] LWN. The current state of kernel page-table isolation, https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/ Dec. 2017.

[56] Maisuradze, G., and Rossow, C. ret2spec: Speculative execution using return stack buffers. In *CCS* (2018).

[57] Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., and Römer, K. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS* (2017).

[58] Microsoft. Lock your windows 10 pc automatically when you step away from it, https://support.microsoft.com/en-us/help/4028111/windows-lock-your-windows-10-pc-automatically-when-you-step-away-from 2019.

[59] Monaco, J. SoK: Keylogging Side Channels. In *S&P* (2018).

[60] O'Keeffe, D., Muthukumaran, D., Aublin, P.-L., Kelbert, F., Priebe, C., Lind, J., Zhu, H., and Pietzuch, P. Spectre attack against SGX enclave, Jan. 2018.

[61] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The Spy

in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS* (2015).

[62] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[63] Payer, M. HexPADS: a platform to detect "stealth" attacks. In *ESSoS* (2016).

[64] Percival, C. Cache missing for fun and profit. In *BSDCan* (2005).

[65] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium* (2016).

[66] Preisvergleich Internet Services AG. Intel with Packaging: tray, Listed since: from 2013. https://geizhals.eu/?cat=cpu1151&xf=3362_2013%7E590_tray Retrieved 2019-04-22.

[67] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS* (2009).

[68] Schwarz, M., Lipp, M., Gruss, D., Weiser, S., Maurice, C., Spreitzer, R., and Mangard, S. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS* (2018).

[69] Schwarz, M., Maurice, C., Gruss, D., and Mangard, S. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC* (2017).

[70] Schwarz, M., Schwarzl, M., Lipp, M., and Gruss, D. NetSpectre: Read Arbitrary Memory over Network. *arXiv:1807.10535* (2018).

[71] Schwarz, M., Weiser, S., and Gruss, D. Practical enclave malware with Intel SGX. In *DIMVA* (2019).

[72] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., and Mangard, S. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA* (2017).

[73] Stecklina, J., and Prescher, T. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480* (2018).

[74] Sullivan, D., Arias, O., Meade, T., and Jin, Y. Microarchitectural Minefields: 4K-aliasing Covert Channel and Multi-tenant Detection in IaaS Clouds. In *NDSS* (2018).

[75] The Chromium Projects. Site Isolation, 2018.

[76] Trippel, C., Lustig, D., and Martonosi, M. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *arXiv:1802.03802* (2018).

[77] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium* (2018).

[78] Vila, P., Köpf, B., and Morales, J. F. Theory and practice of finding eviction sets. *arXiv:1810.01497* (2018).

[79] Weichbrodt, N., Kurmus, A., Pietzuch, P., and Kapitza, R. Asyncshock: Exploiting synchronisation bugs in Intel SGX enclaves. In *ESORICS* (2016).

[80] Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., and Yarom, Y. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report* (2018).

[81] Wong, H. Store-to-load forwarding and memory disambiguation in x86 processors, http://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/ jan 2014.

[82] Wu, Z., Xu, Z., and Wang, H. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. *IEEE/ACM Transactions on Networking* (2014).

[83] Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., and Schlichting, R. An exploration of L2 cache covert channels in virtualized environments. In *CCSW'11* (2011).

[84] Xu, Y., Cui, W., and Peinado, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P* (May 2015).

[85] Yarom, Y., and Falkner, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).

[86] Zhang, T., Zhang, Y., and Lee, R. B. Cloudradar: A real-time side-channel attack detection system in clouds. In *RAID* (2016).

[87] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS* (2014).