Store-to-Leak Forwarding: There and Back Again

Claudio Canella, Lukas Giner, Michael Schwarz

claudio.canella@iaik.tugraz.at, lukas.giner@iaik.tugraz.at, michael.schwarz@cispa.saarland

Abstract

Modern processors optimize performance by using branch prediction and out-of-order execution techniques, which have been exploited by Meltdown and Spectre. Another optimization that has received far less attention is store-to-load forwarding, where the value of a previous store is forwarded to a subsequent load.

In the introduction of the talk, we provide the necessary background for the talk, i.e., information on in-order and out-of-order execution, and transient execution and its side effects. We then discuss how the store buffer plays an essential role in store-to-load forwarding and how several attacks exploit the different paths of its matching logic. Afterward, we discuss our attack primitives targeting the store buffer and demonstrate them in a real-world attack demo. We also discuss how Meltdown has been fixed in hardware and how we reverse-engineered these mitigations. Based on this information, we then present a microarchitectural KASLR break. Finally, we present how such microarchitectural KASLR breaks can be prevented using our countermeasure.

1 Overview

In this whitepaper, we cover the topics of our talk and also provide technical background. The paper is a pre-print of two papers "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs" [2] and "KASLR: Break It, Fix It, Repeat" [1]. While attacks like Meltdown and Spectre and their effects received widespread attention, other optimizations have received far less attention. We provide new insights into these optimizations, show how they can be exploited for attacks, and how some of these attacks can be mitigated using a software workaround. As other attacks target the same optimization, we show how these attacks differ despite exploiting the same optimization. We also provide insights into how Meltdown has been fixed in hardware.

The main takeaways of both the talk and the whitepaper are as follows.

1. Two new KASLR breaks as well as side-channel leakage of control flow and kernel memory based on the TLB and transient execution.
2. There are effects that are closely related to Meltdown.
3. How Intel fixed Meltdown in hardware.
4. Processor optimizations introduce new problems that have not been considered for a long time.

5. Processors are still vulnerable to store-to-leak even with all MDS mitigations enabled.

References

[1] Canella, C., Schwarz, M., Haubenwallner, M., Schwarzl, M., and Gruss, D. KASLR: Break It, Fix It, Repeat. In *AsiaCCS* (2020).

[2] Schwarz, M., Canella, C., Giner, L., and Gruss, D. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725* (2019).

Store-to-Leak Forwarding (Updated and Extended Version)

Michael Schwarz
CISPA Helmholtz Center for Information Security
michael.schwarz@cispa.saarland

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.

This updated extended version of the original paper includes new results and explanations on the root cause of the vulnerability and shows how it is different to MDS attacks like Fallout [8].

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 [16, 27, 65, 67, 88]. With knowledge about the internal CPU state, it is possible to attack cryptographic algorithms [4, 41, 43, 57, 65, 67, 88], spy on user interactions [28, 55, 71], or covertly transmit data [57, 60, 85, 86]. With the recent discovery of Meltdown [56], Foreshadow [80], and Foreshadow-NG [83], 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, 43], 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 [14, 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 [17, 21, 30, 46]. The software mitigations are based on the idea of separating user and kernel space in stricter ways [24]. While such a stricter separation does not only prevent Meltdown, it also prevents other microarchitectural attacks on the kernel [24], e.g., microarchitectural KASLR (kernel address-space layout randomization) breaks [26, 35, 45]. Still, software mitigations may incur significant performance overheads, especially for workloads that require frequent switching between kernel and user space [21]. 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 [34], 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 false dependencies. 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 [48], 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 [24].

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*).
(5) We provide an analysis of the microarchitecture explaining the root cause of the vulnerability we discovered.

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 provide a root-cause analysis on the microarchitectural level in Section 7. We discuss the context of our attack and related work in Section 8. We conclude in Section 9.

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, 27, 49, 65, 88]. Today, the most important techniques are Prime+Probe [65, 67] and Flush+Reload [88]. 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 [90], user input [28, 55, 71], kernel addressing information [26], and cryptographic algorithms [4, 43, 88].

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 [27, 57, 60].

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) [18]. 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* [11, 48, 56]. Spectre-type attacks [11, 12, 34, 47, 48, 50, 59] exploit the transient execution of instructions before the misprediction by one of the processor's prediction mechanisms is discovered. Meltdown-type attacks [3, 11, 37, 38, 47, 56, 76, 80, 83] 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 [44]) 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.

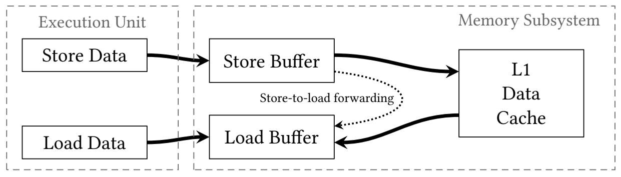


Figure 1: **A store operation stores the data in the store buffer before it is written to the L1 data cache. Subsequent loads can be satisfied from the store buffer if the data is not yet in the L1 data cache. This is called store-to-load forwarding.**

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 [39]. 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 [39].

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 [33].

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 [84]. 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 [84].

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) [34], also named Spectre-STL [11].

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 [84]. This performance optimization can further reduce the latency of loads, but also leaks information across hyperthreads [77]. Furthermore, a similar effect also exists if the least significant 20 bits match [44]. If the load causes a fault, this even leads to a Meltdown-type data leakage [8].

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 [40] 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. [35] breaks KASLR. An unprivileged attacker accesses a kernel memory location and triggers a page fault. The operating system handles the page fault interrupt and hands control back to an error handler in the user program. The attacker now measures how much time passed since triggering the page fault. Even though the kernel address is inaccessible to the user, the address translation entries are copied into the TLB. The attacker now repeats the attack steps, measuring the execution time of a second page fault to the same address. If the memory location is valid, the handling of the second page fault will take less time as the translation is cached in the TLB. Thus, the attacker learns whether a memory location is valid even though the address is inaccessible to user space.

The same effect has been exploited by Jang et al. [45] in combination with Intel TSX. Intel TSX extends the x86 instruction set with support for hardware transactional memory via so-called TSX transactions. A TSX transaction is aborted without any operating system interaction if a page fault occurs within it. This reduces the noise in the timing differences that was present in the attack by Hund et al. [35] 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.

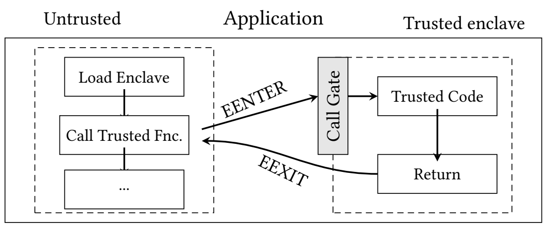


Figure 2: **In the SGX model, applications consist of an untrusted host application and a trusted enclave. The hardware prevents any direct access to the enclave code or data. The untrusted part uses the `EENTER` instruction to call enclave functions that are exposed by the enclave.**

The prefetch side channel presented by Gruss et al. [26] 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) [40]. 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 [52], race conditions [82], or side channels [7, 75] 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 [74].

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 [87], page-cache attacks [22], TLBleed [19], and DRAMA [68] (cf. Section 5). Third, we augment *Fetch+Bounce* with speculative

```
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.**

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 [39].

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 [48], call [48], or return [50, 59]. 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 [56]. 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 [24] is not enabled, and the kernel is again mapped in user space [14]. 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 TLB. Figure 4 shows how *Fetch+Bounce* works. The basic idea is to

```
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.**

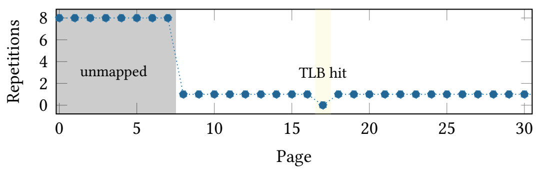


Figure 5: **Mounting *Fetch+Bounce* on a virtual memory range allows to clearly distinguish mapped from unmapped addresses. Furthermore, for every page, it allows to distinguish whether the address translation is cached in the TLB.**

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 [45], 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 measurement with *Fetch+Bounce* destroys the information. Thus,

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

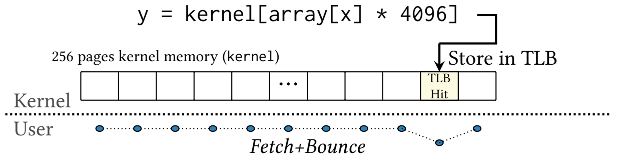


Figure 6: ***Speculative Fetch+Bounce* allows an attacker to use Spectre gadgets to leak data from the kernel, by encoding them in the TLB. The advantage over regular Spectre attacks is that no shared memory is required, gadgets are simpler as an attacker does not require control of the array base address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

to repeat *Fetch+Bounce* for one address, we must evict the TLB in between, e.g., using the strategy proposed by Gras et al. [19].

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 [73]. 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 [48]. 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*.

Data Bounce has the huge advantage that there are no false positives. Store-to-load forwarding does not work for invalid addresses.

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	✓	✓	✓
Lab	Xeon Silver 4208	✗	✗	✗
Cloud	E5-2650 v4	✓	✓	✓
Cloud	Unspecified Cascade Lake	✗	✗	✗

Additionally, Flush+Reload when applied to individual pages does not have false positives, as the prefetcher on Intel CPUs cannot cross page boundaries [40]. 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 [56, 80] or aliasing effects [77] but rather a missing permission check. Hence, the store-to-load forwarding works reliably as expected [84]. 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 up until Whiskey Lake and Coffee Lake R (both released end of 2018)[1]. *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 [14].

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.[2] 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.

[1]Since the time of writing, we have tested our attack primitives on even more CPUs and it appears that newer CPUs include fixes.

[2]Since the time of writing, we discovered an even faster and more reliable KASLR break [10].

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. [45] 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 [53], 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. [45] 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. [44]. 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 [56] and Foreshadow [80]. 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 [45].

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 [45] 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.

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

De-randomizing the Direct-physical Map. In Section 2.5, we discussed that the Linux kernel has a direct-physical map that maps 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 [53], 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 [40]. 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 [32], 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. [35]	17 s	96 %	-
Gruss et al. [26]	500 s	N/A	cache eviction
Jang et al. [45]	5 ms	100 %	Intel TSX
Evtyushkin et al. [15]	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 [19, 26, 35, 45] or branch-predictor states [15, 16]. We compare *Data Bounce* against previous attacks on kernel ASLR [15, 26, 35, 45].

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. [45] and Evtyushkin et al. [15]. 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. [45] 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. [15] 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' [36], 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. [74] 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. [75]. 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* [74]. 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).

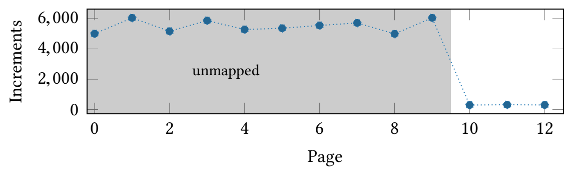


Figure 7: ***Data Bounce* with Evict+Reload in JavaScript clearly shows whether an address (relative to a base address) is backed by a physical page and thus valid.**

There are two main challenges for mounting *Data Bounce* from JavaScript. First, there is no high-resolution timer available. Therefore, we need to build our own timing primitive. Second, as there is no flush instruction in JavaScript, Flush+Reload is not possible. Thus, we have to resort to a different covert channel for bringing the microarchitectural state to the architectural state.

Timing Primitive. To measure timing with a high resolution, we rely on the well-known use of a counting thread in combination with shared memory [20, 72]. As Google Chrome has re-enabled `SharedArrayBuffers` in version 67[3], 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 [20, 72] 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 [20, 48, 64, 72, 81]. 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. [48] 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

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

speed is orders of magnitude slower than the native implementation, as it can be seen in Figure 7. Still, we can detect whether a 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 [71] or DRAMA [68], 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 [25, 71] or have to blindly attack targets until the correct target is found [75].

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. [19].

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 [23, 29].

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 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 [62, 64, 70, 71] 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. [71] 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.

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 [54, 64].

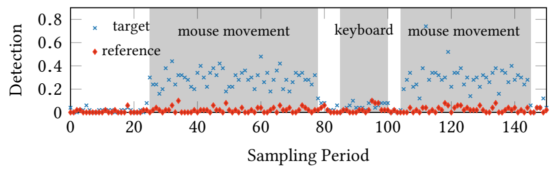
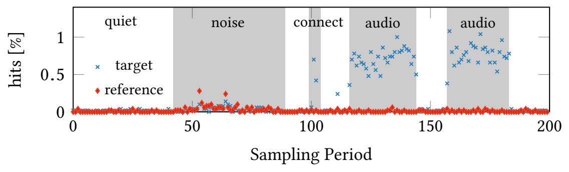
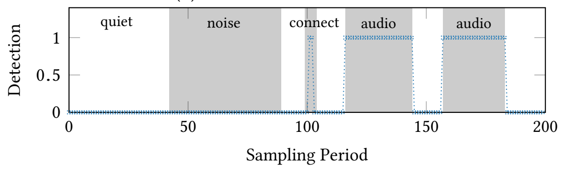


Figure 8: **Mouse movement detection. The mouse movements are clearly detected. The USB keyboard activity does not cause more TLB hits than observed as a baseline.**



(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.**

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 [61], 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 [28].

```
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`.

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 to encode values leaked from the kernel [12, 34, 47, 48, 50, 59, 63, 73]. Other covert channels for Spectre attacks, such as port contention [6] or AVX [73] have since been presented. However, it is unclear how commonly such gadgets can be found and can be exploited in real-world software.

With *Speculative Fetch+Bounce*, we show how 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 [24]. 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 [79]. 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 [58], 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 [11]. 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 [48]. 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 MICROARCHITECTURAL ROOT-CAUSE ANALYSIS

We now peform a microarchitectural root-cause analysis to distinguish *Data Bounce* from the *write transient forward* effect described in the Fallout paper [8]. Generally, store-to-load forwarding uses two components: the store buffer and the memory disambiguation predictor. As store-to-load forwarding intends to forward data from a previous store that has not yet been written to the memory subsystem to a subsequent load from the same address, we can distinguish four different cases.

The first case is a **true positive match**. In the true positive match case, the store buffer contains an entry that potentially matches with the full physical address. The store-to-load forwarding logic hence forwards the data to the load as this is expected to be the correct behavior.

The second case is then the **true negative match.** Contrary to the true positive match, the store buffer does not find a potentially matching entry and, indeed, there is no match with a full physical address. The store buffer then does not forward any information as this is again the expected behavior. In this work, we exploit this as negative information in combination with the true positive case to determine the validity of addresses.

The third is the **false negative match** case. In this case, the store buffer finds no matching store even though a full physical address match existed. As expected, no forwarding takes place and the executed load works on outdated values, e.g., from the L1 cache. One likely situation for this was shown by Horn [34] where the load operation was scheduled earlier than the store it depends on, hence no store buffer entry can exist and the load transiently uses stale data.

The final case is then the **false positive match**. At first, the store buffer finds a matching store, which later on turns out to not be a full physical address match. Nevertheless, the load continues as a zombie load before being squashed and transiently forwards data to dependent instructions. Islam et al. [44] exploited this behavior to obtain physical-address information via a timing attack. In its essence, this is the *write transient forwarding effect* exploited in the Fallout attack [8].

Based on this analysis, we can conclude that the effects used in this paper clearly differs from the effect exploited in Fallout with the first exploiting a combination of the true positive and negative match and the second one exploiting the false positive match. Note that a similar analysis has been performed by Canella et al. [9].

Experimental Verification. Designing an experiment on modern CPUs to verify experimentally that the effects have different root causes is difficult. However, for this evaluation, we can rely on older microarchitecture. We used a Pentium 4 531 from 2004 running Ubuntu 13.04 (kernel 3.8.0) with all mitigations disabled. The Pentium 4 already supports out-of-order execution and speculative execution. We verified that our CPU is vulnerable to transient-execution attacks by successfully running a Spectre-PHT and Spectre-BTB PoC [11].

On the Pentium 4, we can reliably reproduce *Data Bounce*, *i.e.*, the cases of a true positive and true negative match. This result shows that Meltdown-type effects are already observable on the pre-Core microarchitectures. However, the CPU is not vulnerable to Fallout, *i.e.*, there is no false positive match in the store buffer. Hence, from this experiment we can conclude that the effects observed in this paper and in the Fallout paper are indeed different effects, and that the *write transient forwarding effect* exploited in Fallout was introduced only with later microarchitectures.

Offset Match. Fallout [8] describes the *write transient forward* effect to forward the previous store transiently to the load with the exact *same* page offset. However, this is not entirely accurate. In our experiments, we have observed this effect as long as the page offset of the load is within the range (*store_offset*, *store_offset+store_size*) where *store_offset* is the page offset of the store instruction and *store_size* the number of bytes stored (e.g., 1 byte for a `movb` store). This shows that not only the page offset is taken into account but also the number of bytes actually written by the store instruction. For example, if the store operation is a `movq` instruction the page offset of the faulting load operation can be up to 7 bytes higher. Further, we experimentally verified that we can index a store to an `XMM` register (128 bit) with a different offset up to 15 and to an `YMM` register (256 bit) up to 31 bytes.

8 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. [45]. 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 [69]. *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. [45]. 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. [20].

Gras et al. [19] 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. [19] 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, 47, 73, 79], 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 [48]. 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 [24] was deployed on all modern operating systems to ensure that the kernel is not mounted in the user's address space. This prevents Meltdown but also claims to prevent side-channel attacks on kernel addresses [26, 35, 45] including *Data Bounce*, *Fetch+Bounce*, and *Speculative Fetch+Bounce*. As Canella et al. [10] have shown does this claim not hold completely as they were still able to de-randomize the kernel. They analyzed the commonalities of side-channel attacks on kernel addresses and propose FLARE to mitigate such attacks. Unfortunately, Koschelet al. [51] have shown that this is also insufficient. Additionally, we propose to apply the same principle to SGX and sandboxes, similar to site isolation [78]. 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 [13, 27, 31, 42, 66, 89], 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.

9 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 Coffee 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 and ARM. 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. US Patent 6,378,062, April 2002.
[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, February 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* (2014).
[5] Bernstein, D. J. Cache-Timing Attacks on AES, http://cr.yp.to/antiforgery/cachetiming-20050414.pdf 2005.
[6] Bhattacharyya, A., Sandulescu, A., Neugschwandt ner, M., Sorniotti, A., Falsafi, B., Payer, M., and Kurmus, A. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS* (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., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., and Yarom, Y. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS* (2019).
[9] Canella, C., Khasawneh, K. N., and Gruss, D. The Evolution of Transient-Execution Attacks. In *GLSVLSI* (2020).

[10] Canella, C., Schwarz, M., Haubenwallner, M., Schwarzl, M., and Gruss, D. KASLR: Break It, Fix It, Repeat. In *AsiaCCS* (2020).

[11] 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. In *USENIX Security Symposium* (2019). Extended classification tree and PoCs at https://transient.fail/.

[12] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., and Lai, T. H. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P* (2019).

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

[14] 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 August 2018.

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

[16] Evtyushkin, D., Riley, R., Abu-Ghazaleh, N. C., ECE, and Ponomarev, D. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS* (2018).

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

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

[19] 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).

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

[21] Gruss, D., Hansen, D., and Gregg, B. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. *USENIX ;login* (2018).

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

[23] 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).

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

[25] 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).

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

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

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

[29] 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).

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

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

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

[33] 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.

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

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

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

[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, 2018. Revision 3.0.

[39] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2019.

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

[41] 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* (2015).

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

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

[44] Islam, S., Moghimi, A., Bruhns, I., Krebbel, M., Gulmezoglu, B., Eisenbarth, T., and Sunar, B. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium* (2019).

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

[46] 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.

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

[48] 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).

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

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

[51] Koschel, J., Giuffrida, C., Bos, H., and Razavi, K. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In *EuroS&P* (2020).

[52] 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 Symposium* (2017).

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

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

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

[56] 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).

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

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

[59] Maisuradze, G., and Rossow, C. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS* (2018).

[60] 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).

[61] 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.

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

[63] 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.

[64] 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).

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

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

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

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

[69] 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.

[70] 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).

[71] 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).

[72] 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).

[73] Schwarz, M., Schwarzl, M., Lipp, M., and Gruss, D. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS* (2019).

[74] Schwarz, M., Weiser, S., and Gruss, D. Practical Enclave Malware with Intel SGX. In *DIMVA* (2019).

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

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

[77] 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).

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

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

[80] 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).

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

[82] Weichbrodt, N., Kurmus, A., Pietzuch, P., and Kapitza, R. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS* (2016).

[83] 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, https://foreshadowattack.eu/foreshadow-NG.pdf 2018.

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

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

[86] 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).

[87] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P* (2015).

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

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

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

KASLR: Break It, Fix It, Repeat

Claudio Canella
Graz University of Technology

Michael Schwarz
Graz University of Technology

Martin Haubenwallner
Graz University of Technology

Martin Schwarzl
Graz University of Technology

Daniel Gruss
Graz University of Technology

ABSTRACT

In this paper, we analyze the hardware-based Meltdown mitigations in recent Intel microarchitectures, revealing that illegally accessed data is only zeroed out. Hence, while non-present loads stall the CPU, illegal loads are still executed. We present EchoLoad, a novel technique to distinguish load stalls from transiently executed loads. EchoLoad allows detecting physically-backed addresses from unprivileged applications, breaking KASLR in 40 µs on the newest Meltdown- and MDS-resistant Cascade Lake microarchitecture. As EchoLoad only relies on memory loads, it runs in highly-restricted environments, e.g., SGX or JavaScript, making it the first JavaScript-based KASLR break. Based on EchoLoad, we demonstrate the first proof-of-concept Meltdown attack from JavaScript on systems that are still broadly not patched against Meltdown, *i.e.*, 32-bit x86 OSs.

We propose FLARE, a generic mitigation against known microarchitectural KASLR breaks with negligible overhead. By mapping unused kernel addresses to a reserved page and mirroring neighboring permission bits, we make used and unused kernel memory indistinguishable, *i.e.*, a uniform behavior across the entire kernel address space, mitigating the root cause behind microarchitectural KASLR breaks. With incomplete hardware mitigations, we propose to deploy FLARE even on recent CPUs.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

meltdown; side-channel attack; transient execution; kaslr; countermeasure; reverse engineering

1 INTRODUCTION

CPUs are optimized for performance and efficiency. Some optimizations are exposed to the user via the instruction-set architecture (ISA), the hardware-software interface, but most are transparent to the developer. CPU vendors can implement any optimizations while still adhering to the ISA, without taking security into account.

As a consequence, many attacks on the microarchitectural CPU state have been published [25]. Most of these are side-channel attacks, including attacks on cryptographic algorithms [4, 45, 46, 60, 69, 71, 96], on user interactions [34, 58, 78], but also covert data transmission [60, 62, 94, 95]. Meltdown [59], Foreshadow [87, 93], RIDL [89], ZombieLoad [79], and Fallout [9] are recent microarchitectural attacks that go beyond side-channel attacks and directly leak (arbitrary) data instead of metadata. These attacks, commonly referred to as Meltdown-type transient-execution attacks [10], exploit the lazy fault handling property of some CPUs. With lazy fault handling, the CPU continues using data in the out-of-order execution even if the loading of the data resulted in a fault, e.g., failed the privilege-level check. While the data never becomes visible on the architectural level, it is encoded in the microarchitectural state, *i.e.*, in the cache. From there, it is made visible on the architectural level using microarchitectural side-channel attacks.

Meltdown-type transient-execution attacks [10] break the hardware enforced-isolation between the trusted kernel and untrusted user programs. These attacks, which are present in most Intel CPUs, showed that it is not always possible to protect an application against side-channel attacks. This is contrary to the belief that side-channel attacks have to be prevented by the application itself [5, 46].

Deeply rooted in the CPU, either close to or in the critical path, most transient-execution attacks cannot be fixed with microcode updates. However, on recent CPUs, Intel introduced hardware mitigations for the first Meltdown-type attacks [17, 39, 40]. On the latest microarchitecture (Cascade Lake), all known Meltdown-type attacks are mitigated in hardware [41]. However, due to their severity and the ease of exploitation, operating systems (OSs) rolled out software-based mitigations to prevent Meltdown [30]. These software mitigations introduce a stricter separation of user and kernel space [31, 32]. This stricter separation does not only prevent Meltdown but also prevents other microarchitectural attacks on the kernel [31], e.g., KASLR (kernel address space layout randomization) breaks [32, 37, 49] which allow an attacker to de-randomize the location of the kernel in the address space. As a drawback, these software mitigations may introduce significant performance overhead. This is especially true for workloads that need frequent switching between user and kernel space [30]. Consequently, hardware manufacturers solved the underlying root cause directly in recent CPUs, making the software mitigations obsolete.

Even though new CPUs are not vulnerable anymore to the original Meltdown attack, we show that they still show signs of Meltdown-type effects. In this paper, we investigate CPUs that have hardware-based Meltdown mitigations. We analyze these fixes and develop the hypothesis that they only prevent the data from being used in subsequent operations, not the actual load. We confirm this hypothesis by showing that the fixes introduce new side effects, namely that on illegal accesses to kernel addresses, the CPU zeroes out the data but still performs the load. In contrast, loads from non-present pages stall the CPU. We present a method based on Flush+Reload [96] to distinguish the stalling behavior of loads. With this method, we can exploit the side effects of the Meltdown mitigations to break KASLR reliably. By probing the kernel space for load stalls, we detect whether the probed virtual address is physically backed, revealing the location of the kernel. We demonstrate that these effects can also be exploited on older CPUs, which are affected by Meltdown but protected by software mitigations.

Our KASLR break, EchoLoad, works on all major OSs (Linux, Windows, macOS, and Android x86_64). We tested the KASLR break on Intel microarchitectures from Arrandale (2010) to Cascade Lake (2019) on Atom, Core, and Xeon CPUs. Even on Cascade Lake with fixes for Meltdown and MDS [41], we de-randomize the kernel in 40 µs (F-score 1, $n = 10^9$). Hence, our KASLR break is the fastest and most reliable one published. Moreover, EchoLoad is the only KASLR break that only relies on memory loads and works on Intel microarchitectures since at least 2010. EchoLoad even works on KPTI, the Linux software mitigation for Meltdown.

As EchoLoad does not require anything but memory loads, it works in restricted environments such as SGX and JavaScript. We highlight that EchoLoad can aid kernel exploitation from within SGX enclaves, facilitating SGX malware [81, 82]. In contrast to previous ASLR breaks from JavaScript [7, 29, 76], we are the first to demonstrate a microarchitectural KASLR break from JavaScript on x86 OSs. We also show that on older unpatched x86 OSs, Meltdown can even be exploited from JavaScript. This is particularly dangerous for any Windows XP machines (1–3% of Desktop computers [68]), for which no software patches are available, but which are still running in official, commercial, industrial, or personal environments. Our attack will also soon be possible on unprotected 64-bit systems as WebAssembly plans to extend the size of linear memory indices to 64 bit [91]. We pinpoint the remaining challenges for widely deployable JavaScript-based Meltdown exploits.

To mitigate all microarchitectural attacks on KASLR, including EchoLoad, we present FLARE (Fake Load Address REsponse). The basic idea is to back the entire kernel address space with physical pages. FLARE prevents previous attacks [9, 32, 37, 49, 76] by hiding the kernel within a virtual-address range appearing to be valid. Our proof-of-concept implementation has a memory overhead of only 12 kB, and no measurable runtime overhead.

FLARE is compatible with KPTI on Meltdown-affected CPUs, and forms a low-cost mitigation on CPUs with hardware fixes. We evaluated our open-source proof-of-concept implementation of FLARE[1] for Linux for both cases. Our results show that FLARE indeed prevents all known microarchitectural attacks on KASLR.

We conclude that while the hardware mitigation for Meltdown fixes the problem of Meltdown-US [10], they introduce a new side effect by merely zeroing out data that is illegally accessed, enabling EchoLoad. Based on our analysis of the behavior of AMD and ARM CPUs, we believe that the only complete solution for Meltdown is to treat inaccessible pages the same way as unmapped pages. Furthermore, the software-based isolation of user and kernel space [31] is not sufficient, and we thus suggest to deploy FLARE to prevent microarchitectural attacks on the kernel.

Contributions. The contributions of this work are:

(1) We analyze Meltdown hardware fixes on Intel CPUs and discover a Meltdown-related effect on Meltdown-fixed Intel CPUs.
(2) We present KASLR and ASLR breaks, even from SGX and including the first KASLR break from JavaScript.
(3) We show a JavaScript Meltdown attack on 32-bit x86.
(4) We propose FLARE, a mitigation preventing currently known microarchitectural attacks on KASLR with negligible overhead.

[1]https://github.com/IAIK/FLARE

Outline. Section 2 provides background on ASLR and attacks on ASLR. We analyze Meltdown hardware fixes in Section 3. Section 4 presents our new mitigation for microarchitectural KASLR breaks. Section 5 evaluates FLARE's performance and efficacy against attacks. Section 6 discusses related work. Section 7 concludes.

Responsible Disclosure. We responsibly disclosed our findings to Intel on August 5, 2019, and Intel acknowledged them.

2 BACKGROUND

In this section, we provide the background on caches, transient execution and transient-execution attacks, virtual memory, Intel SGX, and address space layout randomization (ASLR).

2.1 Cache Attacks

Caches were designed to hide the latency of memory accesses, creating a timing side channel. Over the past two decades, many different attack techniques have been proposed [5, 33, 54, 69, 96]. Two of these attacks are Prime+Probe [69, 71] and Flush+Reload [96]. In a Prime+Probe attack, an attacker constantly measures how long it takes to fill a cache set with the same set of data. Whenever a victim accesses a cache line mapping to the same cache set, the attacker will measure a higher runtime for the filling. In a Flush+Reload attack, an attacker constantly flushes a cache line and reloads the data. By measuring how long the reload takes, the attacker can infer whether a victim has accessed the data in the meantime. As Flush+Reload exhibits low noise and has a fine granularity, it has been used for attacks on user input [34, 58, 78], cryptographic algorithms [4, 46, 96], and web server function calls [97].

Side channels can also be used to build covert channels. In a covert channel, the attacker controls both the sender and receiver. The goal is then to leak information from one security domain to another, bypassing isolation on both the functional and system level. Both Prime+Probe and Flush+Reload have been used in high-performance covert channels [33, 60, 62].

2.2 Transient-execution Attacks

Another optimization is out-of-order execution, avoiding CPU stalls when in-order instructions wait for operands. Instructions are decoded into micro-operations (μOPs) [22] and placed in the Re-Order Buffer (ROB), along with their operands. While waiting for operands, μOPs whose operands are already available are scheduled in the meantime. Results of the out-of-order executed instructions are stored until they can be retired.

Modern software is rarely linear but contains branches. To avoid pipeline stalls upon unresolved branch conditions, modern CPUs implement speculative execution, predicting the most likely outcome of the branch and starting execution along the predicted path. The results are again placed in the ROB until retirement, *i.e.*, the prediction has been verified. If the prediction was correct, a significant speedup is achieved. Otherwise, the CPU has to revert all results and needs to flush the pipeline and the ROB. Unfortunately, microarchitectural state changes, such as loading data into the cache or TLB, are not reverted. This allows an attacker to use microarchitectural covert channels to exfiltrate the secret data. Speculative or out-of-order executed instructions that were never committed to the architectural state are also referred to as *transient* *instructions* [10, 53, 59]. Spectre-type attacks exploit transient execution before a misprediction is discovered [10, 36, 51, 53, 55, 61]. Meltdown-type attacks exploit transient execution before a fault or interrupt is handled [3, 9, 10, 39, 40, 51, 59, 79, 85, 87, 89, 93].

Meltdown. Meltdown exploited lazy exception handling in modern CPUs [59]. The attacker triggers a page fault but suppresses it via fault handling, TSX transactions, or misspeculation. While the CPU knows that the access is not allowed, the exception is only raised at the retirement stage. Hence, dependent instructions receive the data and can then, e.g., encode the value in the cache which the attacker can leak using a technique like Flush+Reload.

2.3 Intel SGX

In recent years, software vendors discovered that specific security properties, e.g., for DRM, in theory, are much easier to achieve with trusted-execution mechanisms. Consequently, hardware vendors reacted and developed different trusted-execution environments [1, 42, 50]. Intel developed an instruction-set extension called Software Guard Extension (SGX) [42]. With SGX, an application is split into a trusted and an untrusted part. To protect the former, it is executed within a hardware-backed enclave. In the SGX threat model, neither the OS nor any other application is trusted. Therefore, the CPU guarantees that any memory belonging to the enclave cannot be accessed by anyone else than the enclave. The SGX threat model also allows the remaining hardware to be malicious or compromised. Consequently, the SGX memory is encrypted, protecting it from being directly read from the DRAM module. Additional threats like memory-safety violations [56], side channels [8, 82], or race conditions [77, 92] are considered out of scope and remain an enclave developer's responsibility.

The SGX interface to let the untrusted part enter an enclave conceptually resembles system calls. Once the trusted execution is finished, the result of its computation as well as the control flow is returned to the callee. However, SGX protection mechanisms are one-sided: SGX allows data sharing between the trusted and the untrusted part by giving enclaves full access to the entire host application's address space. Recently, it has been shown that this asymmetric protection gives rise to enclave malware [81].

2.4 Address Translation

Modern OSs rely on memory isolation for security purposes. Hence, CPUs support virtual memory for abstraction and memory isolation. Processes work on virtual addresses and cannot architecturally interfere with each other as the virtual address spaces are non-overlapping and overlapping areas are protected according to the processes' requirements. These virtual addresses have to be translated to physical addresses using multi-level page tables. A dedicated translation-table register indicates the location of the first-level table, e.g., CR3 on Intel architectures. Upon a context switch, the OS updates the translation-table register with the physical address of the first-level page table of the process scheduled next. Page-table entries do not only provide translations but also define properties of memory regions, e.g., executable or not.

2.5 Address Space Layout Randomization

Since the introduction of non-executable (NX) bits, memory corruption attacks have to rely on existing code in the victim process instead of code injection [86]. Shacham et al. [83] generalized the concept of code-reuse attacks, which is now widely known as return-oriented programming (ROP). Subsequently, a variety of code-reuse attack techniques have been described [6, 12, 13, 28, 75].

Code-reuse attacks require knowing addresses of specific code snippets. Similarly, data-only attacks [11, 47] require knowledge of addresses, e.g., of specific data structures. Over the years, many different mitigation techniques have been developed [86], e.g., NX stacks, stack canaries, and ASLR. The idea behind ASLR is to make the addresses of code and data unknown to an attacker by randomizing them. Typically, ASLR randomizes the base address of the executable, stack, heap, and shared libraries. Hence, even if an attacker hijacks the control flow, it is significantly harder to exploit bugs in an application as the location of code snippets usable for code-reuse attacks is unknown. By brute-forcing the location, the chances are high that the process will crash, and any ongoing attack is unsuccessful. Furthermore, an application is re-randomized on every startup, reducing the chances of a successful attack.

General Idea of KASLR. While ASLR initially only protected user-space applications, the kernel space was later on also protected by KASLR [20, 49], e.g., introduced in Windows in 2007 [49], macOS in 2012 [2], and Linux in 2014 [20]. The kernel consists of multiple segments that are individually mapped into the kernel address space. These segments include the code (*i.e.*, text segment), drivers or modules, and data (e.g., stack, heap). The KASLR implementations of the three major OSs (Linux, Windows, macOS) only use coarse-grained randomization, *i.e.*, randomized base address. Fine-grained KASLR implementations using code diversification have been proposed [27, 72] but are not used in practice.

Another property of KASLR implementations is that the kernel is mapped using either 4 kB or 2 MB pages. The mapping is 2 MB-aligned [76], reducing the number of possible offsets. Moreover, the order of the randomized segments is not changed, e.g., in Linux, the text segment always has a lower address than the modules [57]. Consequently, KASLR provides a lower entropy than typical user-space ASLR implementations [20]. However, if an exploit attempt fails, it likely crashes the kernel. Hence, an attacker only has one shot, and exploitation techniques relying on a large number of retries cannot be used against the kernel if KASLR is active.

Linux. In Linux 5.x, most sections are independently randomized at boot, including the direct-physical map, vmalloc and ioremap space (vmalloc area), virtual-memory map (vmemmap), text segment, and modules [24]. The text segment is mapped between `0xffff ffff 8000 0000` and `0xffff ffff c000 0000` with a maximum size of 1 GB [76]. As the kernel has to be aligned to a 2 MB boundary, the randomization has 9 bits of entropy. Therefore, the kernel is placed at one of 512 possible offsets. Modules are mapped using 4 kB pages in a 1 GB range following the text segment. Unmapped pages follow each module before a new module starts [49].

Start and end addresses for the direct-physical map, the vmalloc area, and the vmemmap are documented [57], but analyzing the start addresses on repeated restarts shows that they are only correct if KASLR is disabled. Therefore, we analyzed the KASLR implementation of Linux kernel version 5.2.9. This analysis showed that the possible start address is indeed `0xffff 8880 0000 0000` for the direct-physical map. It is then placed at a random offset from the start address, aligned to a 1 GB boundary. The vmalloc space is placed at a random offset relative to the end of the direct-physical map with at least 1 GB between them. The vmemmap area is then randomized starting from the end of the vmalloc area, again with at least 1 GB between them. The range of possible addresses is, therefore, from `0xffff 8880 0000 0000` to `0xffff fdff ffff ffff`, always with a 1 GB alignment and the preserved order.

Windows. Windows randomizes almost everything except the HAL heap once at boot [44]. Windows first introduced KASLR with Vista [49] and improved it over time [32]. Windows 7 maps the kernel, followed by the drivers in the same range with the same randomization. The address range of the kernel and drivers is `0xffff f800 0000 0000` to `0xffff f803 ffff ffff` [49]. KASLR on Windows 10 differs from Windows 7 as there is a separate area for the kernel and drivers. The kernel is still mapped in the same virtual address range, but drivers are now mapped in the range of `0xffff f800 0000 0000` to `0xffff f80f ffff ffff` [23]. The kernel is also 2 MB-aligned, resulting in 8192 possible offsets. Drivers are mapped with 4 kB pages with a 16 kB alignment.

macOS. Starting with macOS 10.8 (Mountain Lion), the kernel, `kexts` (kernel modules), and zones are randomized [70]. For instance, the kernel is mapped in the range of `0xffff ff80 0000 0000` to `0xffff ff80 2000 0000` with a 2 MB alignment, resulting in 256 possible offsets. The offset at which the kernel is placed relative to the start of the address range is called *kslide*. According to Chen and He [14], kernel and `kexts` share the same `kslide`.

3 A NOVEL (K)ASLR BREAK

In this section, we first analyze the Meltdown hardware mitigation on new Intel CPUs. We then introduce EchoLoad, an attack primitive that exploits incomplete Meltdown countermeasures to break KASLR. We detail how we can use it to break KASLR from an unprivileged user-space application, JavaScript, and SGX.

3.1 Analyzing the Meltdown Mitigation

The Meltdown vulnerability allowed unprivileged users to leak kernel memory (cf. Section 2.2). The immediate workaround was KAISER [31], a software-only solution to unmap the kernel when running in user space. With the Whiskey Lake microarchitecture, Intel fixed the vulnerability in hardware without providing further details on how their fix works. CPUs with the hardware mitigation indicate that they are not vulnerable by having the `RDCL_NO` bit set in the `IA32_ARCH_CAPABILITIES` model-specific register [42].

Lipp et al. [59] argued that stalling the CPU until the permission check is done might be too costly. We suspect that such a change also requires redesigning a significant part of the CPU's pipeline. As the first CPUs with hardware mitigations already shipped approximately one and a half years after Meltdown was disclosed to Intel, we expect only minor hardware changes as mitigation.

Hypothesis. We hypothesize that instead of stalling on an illegal memory load, the CPU zeroes out the result. Hence, the CPU still loads inaccessible memory locations, but instead of providing the real value to dependent instructions, it always provides '0'.

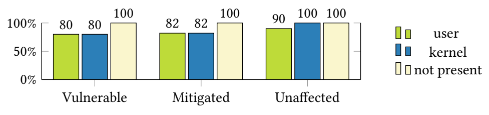


Figure 1: Loads from non-present pages always stall, loads to kernel addresses stall on unaffected AMD CPUs.

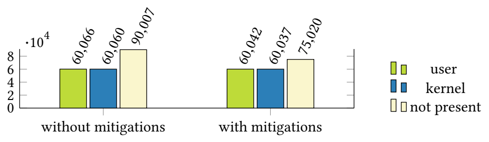


Figure 2: Issued load μOPs for user and kernel addresses (Intel). Only invalid loads from non-present pages are reissued.

Verification. We get the first indication that our hypothesis is correct by simply mounting a Meltdown attack. When running the Meltdown attack on a Xeon Silver 4208 CPU which has the `RDCL_NO` bit set, we always get '0's. To verify our hypothesis, we further analyzed performance counters on three different systems: a Meltdown-vulnerable Intel CPU (i7-8650U), an Intel CPU with hardware mitigations (Xeon Silver 4208), and a non-affected AMD CPU (Ryzen Threadripper 1920X). For all systems, we evaluate performance counters when executing the following code 10^4 times: `if ( transient_begin()) { *(volatile char*)0; oracle[*address]; }`. The function `transient_begin` either starts a TSX transaction if available, or sets up a signal handler for segmentation faults [59]. The null-pointer access is required to always cause an exception.

The first performance counter of interest is the number of CPU stalls when executing the above code. On Intel CPUs, we use `CYCLE_ACTIVITY.STALLS_MEM_ANY`, and on AMD CPUs the "Dispatch Stalls" counter. We set `address` to a valid kernel address. As baselines, we choose a mapped user address as well as a non-present address for `address`. Figure 1 shows the results of the performance counters for all 3 systems. For comparison, we normalized the values such that the highest value on each system represents 100 %.

All CPUs stall when accessing a non-present virtual address. The AMD CPU also stalls when accessing a kernel address. Both Intel CPUs with and without mitigations show the same stall behavior. Hence, even the Intel CPUs with Meltdown mitigations do not stall when accessing a kernel address. This indicates that the memory load for the kernel address is actually issued.

We substantiate this observation by analyzing another performance counter. With the counters `UOPS_DISPATCHED_PORT.PORT_2` and `UOPS_DISPATCHED_PORT.PORT_3`, we can track the number of μOPs issued on the load ports. The sum of these two counters is the number of all memory loads. Figure 2 shows the number of memory loads when running the code mentioned above with a user-space, kernel-space, and non-present address both on an Intel CPU with and without hardware mitigation. When trying to load from a non-present page, the load faults and the load instruction is re-issued [79]. The number of issued loads for kernel addresses is the same as for user-space addresses on both CPUs. This indicates that these loads succeed and do not have to be re-issued.

Finally, we show that the issued loads for kernel addresses indeed load data from the memory hierarchy, and not, e.g., from an internal

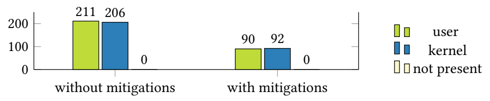


Figure 3: Number of cycles L1D cache misses are pending. User and kernel addresses reach the memory hierarchy, non-present pages do not.

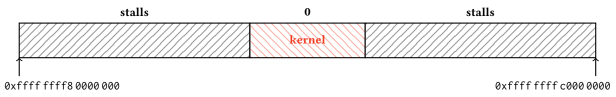


Figure 4: Reading addresses not physically backed stalls the CPU, while kernel addresses return '0' (or the actual data).

```
1 if (transient_begin()) {
2   *(volatile char*)(mem + *address);
3 }
4 if (flush_reload(mem)) return ADDRESS_MAPPED;
5 else return ADDRESS_NOT_MAPPED;
```

Listing 1: The main part of EchoLoad. The address `mem` is only cached if the access to `address` does not stall.

buffer containing '0'. Thus, we monitor the number of cycles that L1 data-cache misses are waiting to be retrieved. Figure 3 shows the values of the performance counter `L1D_PEND_MISS.PENDING_CYCLES` for the previously shown code. While non-present pages do not cause an L1 miss, both user-space and kernel addresses cause L1 misses. This is even the case for CPUs with hardware mitigations against Meltdown, showing that loads to kernel addresses retrieve the actual value, and only later on zero it out.

3.2 Breaking KASLR

EchoLoad is a new microarchitectural KASLR attack exploiting Meltdown-related side effects. EchoLoad reliably breaks KASLR, regardless of OS, software mitigations, and microcode updates. EchoLoad works on all Intel CPUs since 2010, even if they are not affected by Meltdown, e.g., CPUs with the RDCL_NO bit. In contrast to the KASLR break by Schwarz et al. [76], EchoLoad also works on the new Cascade Lake, which is not affected by Meltdown or MDS.

General Idea. The general idea is to distinguish whether accessing a kernel address in the transient-execution domain leads to a stall. We exploit the fact that instructions can only be executed out of order if their data dependencies are fulfilled. Hence, we dereference a user-space memory location where the address is computed based on the value of the kernel address that is being tested.

Listing 1 shows this central part of EchoLoad. First, an attacker induces transient execution by provoking a fault or a misspeculation in Line 1. If the access to `address` in Line 2 stalls, the user address cannot be computed before the transient execution aborts. Otherwise, the user address is dereferenced and, thus, cached before the transient execution aborts. After the transient execution, the attacker probes the user- address in Line 4, e.g., using Flush+Reload. If the user address is cached, `address` is valid, *i.e.*, physically backed. Otherwise, `address` is not valid, *i.e.*, not physically backed. Figure 4 illustrates the general idea of EchoLoad.

CPUs with Meltdown Fixes. To break KASLR on CPUs with Meltdown fixes, we run EchoLoad on all 512 possible kernel offsets (cf. Section 2.5). Only where a physical page backs the tested address, we read 0, on other addresses the CPU stalls.

As the CPU stalls on all reads from addresses that the kernel is not mapped to, we observe no false positives. This makes EchoLoad a very reliable attack that even works on Cascade Lake CPUs.

CPUs without Meltdown Fixes. On CPUs without Meltdown fixes, we cannot rely on the CPU returning 0 for reads on kernel pages. Instead, if KPTI is disabled, we read the actual content of the page. As the content of the page is code, there are 256 possible addresses which could be dereferenced.

As the 256 possible addresses are contiguous, and the cache line size is typically 64 byte, they fall into one out of 4 possible cache lines. Testing 4 adjacent cache lines with Flush+Reload triggers the stride prefetcher [38] on Intel CPUs. Instead, we can exploit the L2 adjacent cache line prefetcher (spatial prefetcher) [38], which fetches the sibling cache line whenever a cache miss is handled. Hence, we only have to check 2 cache lines using Flush+Reload, which works without triggering the stride prefetcher. Consider a case with 4 adjacent cache lines. If the data we read falls into line 0 and we check line 1, we observe a hit on line 1 because the prefetcher also loads it into the cache. The same is true if the data falls into line 3, and we check line 4. By merely checking cache lines 1 and 3, we detect all possible accesses.

We can even further increase the performance by only checking one cache line. By using a kernel module, we investigated the beginning of the kernel text segment and determined that it is always the same across kernel versions (*i.e.*, `0x48`).

EchoLoad also works with KPTI [30] as the pages still mapped with KPTI use the same randomization offset as the rest of the kernel code. While the value differs with KPTI (*i.e.*, `0xf`), it is still the same across kernel versions that use it. As we only look for the beginning of the kernel and we know that the value remains constant, we can reduce the number of cache lines we need to check to 1. This further improves the performance of our KASLR break.

EchoLoad and LVI-NULL. On CPUs that have already received fixes for Meltdown, EchoLoad is the inverse of the LVI-NULL attack [88]. While LVI-NULL abuses the fixes to inject a dummy value of zero to dependent transient instructions in a victim, EchoLoad exploits the inverse effect, *i.e.*, the retrieving of a zero value, to break KASLR.

Evaluation. We evaluated EchoLoad on different Intel microarchitectures running Linux (cf. Table 1). On all CPUs, we evaluated our attack with both KPTI enabled and disabled. These experiments show that KPTI does not prevent EchoLoad. If KPTI is disabled, EchoLoad detects the symbol `startup_64`. With KPTI, it detects the symbol `__entry_text_start`, which is the trampoline required to enter the kernel. As Android is based on the Linux kernel, the behavior on Android is the same. While we evaluate the ability and performance of EchoLoad to leak the kernel code offset, it can equally leak the offsets of all other randomized parts of the kernel.

For the performance evaluation, we used the same setup that Schwarz et al. [76] describe in their paper. We tested 10 different randomizations (*i.e.*, 10 reboots), each 100 times. Using this approach, we have a sample size of 10^3. We evaluated the performance on

Table 1: Environments where we evaluated EchoLoad and Data Bounce (KPTI disabled).

CPU	µarch.	EchoLoad	Data Bounce
Intel Atom x5-Z8300	Cherry Trail	✓	✓
Intel Core i5-450M	Arrandale	✓	✓
Intel Core i5-3230M	Ivy Bridge	✓	✓
Intel Core i5-8250U	Kaby Lake R	✓	✓
Intel Core i7-4790	Haswell	✓	✓
Intel Core i7-6700K	Skylake	✓	✓
Intel Core i7-8650U	Kaby Lake R	✓	✓
Intel Core i7-8565U	Whiskey Lake	✓	✓
Intel Core i9-9900K	Coffee Lake	✓	✓
Intel Xeon E5-1630 v4	Broadwell	✓	✓
Intel Xeon Silver 4208	Cascade Lake	✓	✗
Intel Cascade Lake (Google Cloud)	Cascade Lake	✓	✗
AMD Ryzen Threadripper 1920X	Zen	✗	✗
AMD Ryzen 7 3700	Zen 2	✗	✗
ARM Cortex-A57	A57	✗	✗

Table 2: Performance of EchoLoad in terms of runtime and F-score. Each possible offset is tested a single time.

CPU		Speculation	TSX	Segfault
i7-6700K	Time (F-Score)	63 µs (0.999)	48 µs (1.000)	133 µs (1.000)
i9-9900K	Time (F-Score)	33 µs (1.000)	29 µs (1.000)	86 µs (1.000)
Xeon Silver 4208	Time (F-Score)	51 µs (0.994)	40 µs (1.000)	127 µs (1.000)

a selected number of architectures in all three cases, namely misspeculation, TSX, and segfault handling. Table 2 shows the result of this evaluation. In all tested cases, we achieve almost perfect accuracy. On the i9-9900K, we outperform Data Bounce in terms of time required while matching the accuracy.

Similar to Schwarz et al. [76], we tested EchoLoad with TSX on a larger scale. For that, we tested the same offset 100 million times and repeated the experiment 10 times for a total of 1 billion tries. On all three CPUs (cf. Table 2, we achieved an average F-score of 1, giving us perfect accuracy in detecting the KASLR offset.

On Windows 10, we also tested 10 different randomizations (*i.e.*, 10 reboots), each 100 times. In all cases, we successfully found the location of the kernel image. On macOS 10.11.6, instead of 10 randomizations, we repeated the experiment 100 times to verify that the given kernel range is still correct [14]. We then successfully recovered the kernel location in all 100 randomizations.

3.3 Breaking (K)ASLR from SGX

As EchoLoad only requires memory accesses, it also works in restricted environments. We demonstrate EchoLoad in SGX enclaves breaking host ASLR, victim-enclave ASLR, and KASLR.

While it is also possible to use EchoLoad for detecting the location of SGX enclaves from the host application, this is an artificial scenario. First, the host maps the enclave to its location and, thus, knows where the enclave is. Second, on Linux, the host can access this information from the pseudo file `/proc/self/maps`, containing all virtual-address mappings of the current process. Finally, the host can also probe the virtual memory for the enclave, e.g., using a signal handler to catch segmentation faults. If a region returns `0xff`, it is likely to be an EPC page of an enclave.

EchoLoad from Enclave to Host. *TAP* is a method to break host ASLR from an enclave using Intel TSX [81]. It allows scanning the host address space for mapped pages to mount a ROP attack from inside the enclave, impersonating the host application.

```
if(xbegin() == (~0u)) { *(volatile char*)mem; xend(); }
if(flush_reload(mem)) return ROLL_BACK;
else return IMMEDIATE_ABORT;
```

Listing 2: Analyzing the behavior of the TSX abort. If the transaction is aborted on `xbegin`, `mem` cannot be cached. If the transaction is just rolled back on `xend`, `mem` is cached.

While *TAP* only worked for CPUs with TSX, it does not work on CPUs with MDS fixes in microcode at all. With the microcode update, all TSX transactions abort immediately when started inside an SGX enclave [43]. We further analyzed whether the transaction aborts immediately, or is only rolled back in all cases.

Listing 2 shows the code we use to analyze the TSX-transaction aborts. If the transaction aborts already at the `xbegin` instruction, the memory dereference is never executed. If the transaction executes but then rolls back the executed instructions, the dereference of the address still causes the memory location to be in the cache.

Our results show that the transaction is never started as the address `mem` is never cached after the transaction. Hence, we cannot even use TSX to access memory locations transiently. We observe the same behavior outside an SGX enclave when setting the `TSX_FORCE_ABORT` MSR to 1. While this MSR is documented to abort every TSX transaction on *commit* [98], we verified with our test (Listing 2) that the transaction is not even started.

Consequently, even if TSX is re-enabled in SGX via a microcode update, it can be manually disabled with the `TSX_FORCE_ABORT` MSR to protect against attacks such as *TAP*. This is the case on the Amazon EC2 cloud [79]. In contrast to Data Bounce [76], EchoLoad works on the newest CPU generation, as it does not require TSX. Thus, EchoLoad can be used to mount SGX ROP attacks [81] even if TSX is disabled, once more enabling such attacks.

Due to the unavailability of syscalls and the `rdtsc` instruction inside SGX, we mount EchoLoad behind a misspeculated branch and use a counting thread [82] as a timer. We achieve a speed of 388 Mbit/s for scanning the host address space with EchoLoad. Hence, EchoLoad is a viable alternative to *TAP* to de-randomize the host application from an SGX enclave.

EchoLoad from Enclave to Enclave. Enclaves might not only want to de-randomize the host application but also learn information about other enclaves. While enclaves are mutually untrusted and, thus, cannot access each other, EchoLoad can be used to learn the address-space layout of other enclaves. Moreover, assuming that enclaves have unique sizes, an enclave can even detect which other enclaves are used by the host by detecting their size.

We evaluated EchoLoad in the cross-enclave scenario by loading two enclaves in our test application. One enclave is malicious and leverages EchoLoad to learn which other enclaves are used by the host application. We use the same experiment as for de-randomizing the host to scan the address space for other enclaves. We successfully detect the location and the size of the second enclave used by the host application. The speed for scanning the address space is the same as for de-randomizing the host application.

EchoLoad from Enclave to Kernel. Enclaves may foster stealthy exploits [48, 65, 81, 82]. In this work, we add another primitive to malware hidden inside SGX. With EchoLoad, an enclave can de-randomize KASLR, which is a prerequisite for many kernel exploits.

The same code which is used to de-randomize the host application can be used to de-randomize the kernel. We evaluated EchoLoad inside an SGX enclave to find the KASLR offset. Due to the use of misspeculation and a timing thread, the performance is worse than in native code. However, we still detect the KASLR offset with an F-score of 1 ($n = 10^3$).

3.4 Meltdown and KASLR Break in JavaScript

EchoLoad can even be mounted from a JavaScript sandbox. We demonstrate EchoLoad, and as an extension Meltdown, from the Spidermonkey JavaScript engine 60.1.3 used in Firefox.

There are two challenges for mounting EchoLoad in JavaScript. First, both JavaScript and WebAssembly currently only support a 32-bit linear memory index, restricting arrays to 4 GB [19]. While this prevents EchoLoad on a 64-bit OS, it does not prevent it on 32-bit OSs, which only support a 32-bit virtual address space. Hence, we evaluate this attack on Ubuntu 16.04 (Kernel 4.15.0-60) i686 on an Intel i7-4790. While we are currently limited to 32-bit systems, the WebAssembly developers are planning to increase the size of linear memory indices from 32-bit to 64-bit, allowing the attack on all commodity systems that are not patched against Meltdown [91]. Second, the Spectre mitigations do not only reduce the resolution of the high-resolution timer [67], but also harden the bounds check for arrays, preventing speculative out-of-bounds accesses by default [66]. As our focus is not demonstrating a Spectre attack but a Meltdown-related effect, we use a version of the engine that allows speculative out-of-bounds accesses, as in previous work [53]. To develop widely deployable Meltdown and EchoLoad exploits, further research is necessary to investigate whether other misprediction mechanisms may provide a suitable workaround to the hardened out-of-bounds checks. Note that previous work has already shown that some of these mitigations can be circumvented [35].

Building Blocks. An alternative to the high-resolution timer is a counting thread which is commonly used for microarchitectural attacks in JavaScript [29, 53, 80]. Furthermore, as the `clflush` instruction is not available in JavaScript, we resort to Evict+Reload as described in related work [29, 76, 90]. Instead of measuring only one address in our Evict+Reload, we use amplification on multiple cache lines [63]. With amplification, we encode the out-of-bounds access into multiple different cache lines to achieve more reliable results. To access a kernel address during transient execution, we hide an out-of-bounds array access behind a misspeculated branch.

EchoLoad from JavaScript. By combining the building blocks, we can implement EchoLoad in JavaScript. On average, it takes 25.09 ms ($n = 10^3$, $\sigma_{\bar{x}} = 5.92$) to find the start of the kernel image. The detected offset is relative to the base of the array, which is used for the out-of-bounds accesses. However, an attacker can leverage any JavaScript ASLR break [29] to recover the array base address, and from that compute the absolute address of the kernel image.

Meltdown from JavaScript. Contrary to Linux, many 32-bit OSs still in use do not have Meltdown patches (e.g., Windows XP). Hence, we show that with the building blocks, we can mount a Meltdown attack from JavaScript on such systems. Relying on EchoLoad for the KASLR break, we can even target specific locations in the kernel.

Table 3: We compare microarchitectural attacks on KASLR. EchoLoad outperforms all previous microarchitectural attacks on KASLR while having no requirements.

Attack	Time	Accuracy	Requirements
Hund et al. [37]	17 s	96 %	-
Gruss et al. [32]	500 s	N/A	cache eviction
Jang et al. [49]	5 ms	100 %	Intel TSX
Evtyushkin et al. [21]	60 ms	N/A	BTB reverse engineering
Canella et al. [9]	0.27 s	100 %	MDS vulnerable CPU
Schwarz et al. [76]	42 µs	100 %	Intel CPU before Cascade Lake
EchoLoad (our attack)	29 µs	100 %	-

To evaluate the attack performance of the proof of concept, we disable KPTI and leak a known value from the kernel. Our JavaScript attack leaks 2 B/s, with an error rate of 0.3 % ($n = 10^3$).

3.5 Other Side-Channel Attacks on KASLR

Microarchitectural attacks on KASLR so far relied on either branch-predictor states [21], address-translation caches [32, 37, 49], or store-buffer optimizations [9, 76]. We compare EchoLoad to previous attacks on KASLR [21, 32, 37, 49, 76].

Our attack outperforms all state-of-the-art KASLR breaks on Intel x86 CPUs (cf. Table 3). We outperform Data Bounce [76] in terms of speed and match it in accuracy while having lower requirements. Similar to Data Bounce, EchoLoad also has the advantage over previous microarchitectural attacks that it does not require Intel TSX, or knowledge of internal data structures like the branch-target buffer (BTB) or the store buffer.

For instance, Evtyushkin et al. [21] assume an attacker knows how the BTB works internally, which has not yet been reverse-engineered for microarchitectures after Haswell. Moreover, with the widely-deployed Spectre mitigations [10, 39], the BTB is either cleared on context switch or not shared between privilege levels. Hence, this attack does not work on state-of-the-art CPUs anymore.

The double page-fault attack [37] was the first microarchitectural attack on KASLR. By accessing a kernel memory location, an attacker first triggers a page fault. This triggers an interrupt which is handled by the OS. After handling the interrupt, the OS returns control to a pre-installed error handler in the user-space program. In the error handler, the attacker measures the time it took to handle the fault. The attacker then repeats the attack step, again measuring the time it took to handle the fault. If the kernel address is valid, the first illegal access has created a TLB entry. This speeds up the handling of the second fault, creating a timing side channel. Consequently, a user-space attacker can infer whether a kernel address is valid or not. The requirement for this attack is that the user can install a signal handler to handle segmentation faults. Hence, native code execution is required.

Jang et al. [49] retrofitted the attack by Hund et al. [37] with Intel TSX. TSX is an x86 instruction-set extension introducing hardware transactional memory. If a page fault occurs within a transaction, it is aborted without architecturally raising a fault and, hence, without any OS interaction. This allows the attack to skip the page fault handling of the OS, significantly speeding up the attack and reducing its noise. The approach by Jang et al. [49] only works on CPUs starting from Haswell as it relies on Intel TSX. This extension is not present on low-end CPUs or any CPUs built before 2013 and can be disabled on newer CPUs as well. Intel TSX is, for example, disabled on the Amazon EC2 cloud [79].

Gruss et al. [32] use the software prefetch instruction as a side channel. This side channel exploits that the execution time of the prefetch instruction depends on whether the translation cache holds the correct entry. As the TLB can only hold addresses for which a valid translation, *i.e.*, a physical page is mapped to it, the location of the kernel is revealed due to it consisting of the only valid address mapping within the predefined region. With this attack, the attacker additionally learns the page size that is used for the mapping.

Fallout [9] demonstrates a KASLR break on MDS-vulnerable Intel CPUs. First, they ensure that a user-controlled value is in the store buffer. Then, they attempt to access an address with the same *page offset* ,which is inaccessible. On MDS-vulnerable CPUs, the store-buffer content is transiently forwarded to faulting loads on valid kernel addresses, revealing the location of the kernel. Fallout [9] relies on the opportunistic store-buffer behavior that virtual addresses are likely equivalent if the least-significant 12 bits match. However, this is only the case on MDS-vulnerable Intel CPUs that are not patched. Hence, this KASLR break does not work on CPUs indicating that they are MDS-resistant via the `MDS_NO` flag in the `IA32_ARCH_CAPABILITIES` model-specific register, e.g., on the newest Cascade Lake CPUs.

Data Bounce [76] breaks KASLR by exploiting that the CPU only performs store-to-load forwarding if a physical page backs a virtual address, *i.e.*, the virtual address can be resolved to a physical address. Using this approach, they can break KASLR on all Intel CPUs going back to 2004. They claim that the attack has perfect accuracy and only requires 42 µs to detect the correct kernel location. One of the advantages of this approach over Jang et al. [49] is that it does not require TSX and, hence, is applicable to a broader range of CPUs. However, the behavior of store-to-load forwarding was changed in Cascade Lake CPUs to prevent this attack (cf. Table 1). Hence, while their approach works on microarchitectures starting from the Pentium 4 Prescott to Whiskey Lake and Coffee Lake R, it does not work on the recent Cascade Lake.

EchoLoad relies on the load stalling behavior of the CPU, an effect which has not been exploited so far. As this effect is deeply rooted in the design of the microarchitecture, it cannot easily be fixed (cf. Section 3.1), neither in software nor hardware. Moreover, the attack does not have any requirements as it solely relies on memory loads. As a consequence, even the most recent Cascade Lake is affected by EchoLoad.

4 FLARE: MITIGATING KASLR BREAKS

In this section, we propose FLARE, a defense against KASLR attacks rooted in a CPU's microarchitecture.

FLARE has a negligible memory overhead of only a few kilobytes and next to no runtime overhead. FLARE tackles the root causes of all the microarchitectural KASLR breaks discussed in Section 3.5. It builds on ideas from KAISER [31] and LAZARUS [26] to fix remaining weaknesses efficiently and securely.

The challenge is to fully eliminate differences in:

C1: timing and behavior for mapped and unmapped pages,
C2: timing for different page sizes, and
C3: timing between executable and NX pages.

As we show in this section, FLARE successfully tackles these challenges. However, before we justify these challenges, we briefly introduce a threat model. We then discuss implementation details, corner cases, and pitfalls in Section 4.1.

Threat Model. Our attacker can run unprivileged native code on an up-to-date OS. Furthermore, the attacker knows the exact version of the Linux kernel that the victim uses and, hence, knows the exact structure of the kernel image in memory.

C1: Differences for Mapped and Unmapped Pages. In Section 3.5, we discuss that recent attacks, including EchoLoad, can distinguish mapped from unmapped pages [9, 32, 37, 49, 76]. Therefore, the first challenge is to prevent an attacker from detecting the KASLR offset based on that information.

To tackle this challenge, we map all unmapped virtual addresses in the randomization range to a dummy physical page. Therefore, none of the known attacks that rely on distinguishing mapped from unmapped addresses can de-randomize the kernel anymore.

C2: Timing Differences for Page Sizes. In Section 3.5, we discuss that the attack by Gruss et al. [32] can distinguish different page sizes: Even if the entire kernel space has a valid mapping, different page sizes can create a unique pattern which de-randomizes the kernel. This is especially a problem as the kernel uses different page sizes for its mapping (cf. Figure 5), possibly creating such a unique pattern. We tackle this challenge by avoiding different page sizes in the kernel altogether.

C3: Timing Difference between Executable and NX Pages. Jang et al. [49] showed that there is a timing difference between executable and NX pages. We analyzed the kernel and discovered that executable and NX pages are strictly separated. That is, after the first NX page in the address space there is not a single executable page in the remaining address space. To prevent this straightforward KASLR break, we randomize the executable and the NX range separately and pad them each with executable and NX pages respectively to the full randomization range.

4.1 Implementation Details

The different Linux kernel regions (cf. Section 2.5) are mapped with different properties, *i.e.*, different page sizes and permissions (e.g., executable and NX). Note that we only need to protect the trampoline code if KPTI is active, while we have to protect the following regions without KPTI.

Text Segment. Figure 5 shows that the text segment is mapped using both 4 kB and 2 MB pages. To address C1, we map the entire range where the text segment can be mapped using 4 kB pages, preventing the attacker from seeing the actual text-segment range. To address C2, we map the text segment only with 4 kB pages, preventing attacks that distinguish page sizes [32]. This is not a large kernel change as this is already an ongoing development (cf.

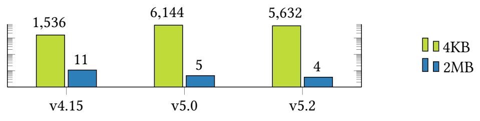


Figure 5: On recent Linux, several 2 MB pages in the kernel text segment have been replaced by 4 kB pages.

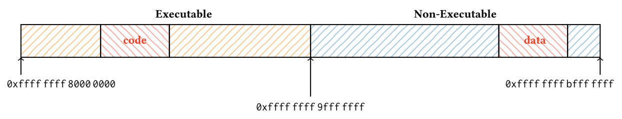


Figure 6: With FLARE, all possible kernel offsets are physically backed, *i.e.*, any potentially read value from this range will be zero. Code and data is independently randomized in 512 MB ranges. This setup allows preventing all currently known microarchitectural attacks on KASLR.

Figure 5). The kernel already supports disabling the use of non-4 kB pages by clearing the CPU capability X86_FEATURE_PSE.

Furthermore, to tackle C3, we use the solution shown in Figure 6. We split the randomization range of the text segment in half. We then use one half for the randomization of executable pages, *i.e.*, the kernel code, and the other for NX pages, *i.e.*, the kernel data. Both regions are then randomized independently to not leak their corresponding start and end addresses. This split does not introduce any compatibility issues, even with relative addressing, as we stay within the maximum addressable range of 4 GB.

Modules. Modules already use 4 kB pages only, solving C2. In our proposal, we pad the code and data sections of every module to a multiple of 1 MB, depending on the size of the largest currently loaded module. We then map the remaining offsets in the address range with dummy modules using 4 kB pages that look exactly the same as the actual modules, *i.e.*, same size for code and data sections. Consequently, using the technique by Jang et al. [49] in the memory range for kernel modules, we only see executable and NX regions of all the same size. This mitigates the templating attack by Jang et al. [49] as the attacker cannot infer anymore which module is real and which one is not. With this approach, we solve all three challenges.

Naturally, the privileged user can dynamically load modules which takes the place of a previous dummy module. Likewise, for the unload, a dummy module replaces the kernel module mapping. However, the implementation should be careful not to leave a small time window open for an attack. In our FLARE proof-of-concept, we enable the loading of modules by first removing the dummy mapping by hooking the function *load_module*. Then the module is loaded, and afterward, the mitigation is re-applied. However, a proper implementation should exchange the page-table entries directly instead. This way, it is guaranteed that no time window is left for the attacker to observe the short unmapping from a concurrent microarchitectural attack, as there simply is no short unmapping. Furthermore, the loading and unloading of modules typically does not happen for an average user.

Direct-Physical Map, Vmalloc, Vmemmap. We analyzed how the direct-physical map, vmalloc, and vmemmap are mapped. None of the pages mapped in this region is executable. Thus, we tackle challenges C1 and C3 by mapping all pages in the corresponding randomization regions in our dummy mapping as NX.

Currently, the kernel does not use an explicit randomization range for each of the three regions. Instead, the kernel uses one large range and only guarantees to preserve their order. To mitigate the attack by Gruss et al. [32], all three must use the same page size. We verified that this is already the case when clearing the

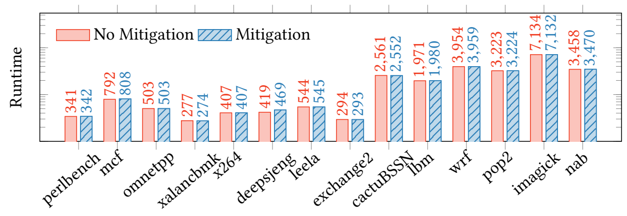


Figure 7: Runtime overhead of FLARE on SPEC CPU 2017.

X86_FEATURE_PSE CPU capability at boot. As this causes significant pressure on the TLB, we propose a different approach.

We propose that the kernel uses an explicit randomization range for each of the three regions. Hence, to tackle C2, we can enforce that the kernel consistently uses one page size per region. This mitigates the attack by Gruss et al. [32]. In the analysis for our defense, we empirically determined the page sizes used for each region. On our test machine running Linux kernel 4.15, the kernel indicates during the boot process that 1 GB pages are used for mapping the direct-physical map. However, our analysis revealed that it is mapped using all three page sizes, *i.e.*, 4 kB, 2 MB, and 1 GB. Similarly, the vmalloc area uses both 4 kB and 2 MB pages. The vmemmap area consisted of 2 MB pages only.

Based on this analysis, we propose to consistently use 2 MB pages for the vmemmap region and 4 kB pages for the vmalloc region. For the direct-physical map, we use 1 GB pages. Unfortunately, we cannot use such a huge dummy page for our mapping as we would reduce the available physical memory by 1 GB. Instead, we pick 1 GB of RAM, which is already mapped in the direct-physical map, and map it using a 1 GB page in our dummy mapping. Hence, we avoid the additional memory overhead without increasing the risk for exploitation as we map the page as NX.

5 EVALUATION

In this section, we evaluate the overhead of FLARE in three aspects, namely runtime overhead using the SPEC CPU 2017 benchmarks [16], module loading overhead, as well as the memory overhead. We also evaluate the efficacy of FLARE by analyzing how successful it is in preventing microarchitectural attacks on KASLR.

5.1 Overhead Analysis

Runtime. We create our dummy mapping directly in the *init_mm* struct which is copied into every newly created process. We only have to apply our mapping once, and every new process has the mitigation enabled. Hence, we expect no runtime overhead.

We confirmed this using the LMbench microbenchmark suite [64]. We evaluated process-creation time (fork and exec) and context switches on an Intel i7-8650U (Linux kernel 5.0.0-15). This involves a larger number of TLB invalidations and address resolutions, *i.e.*, the situations that may see a performance penalty. For process creation, we do not encounter any overhead. Both with and without FLARE, the process creation takes on average 61.14 µs ($n = 10^5$, $\sigma_{\bar{x}} = 0.27$). Similarly, there is no difference in the syscall latency. In both cases, the latency is on average 1.03 µs ($n = 10^5$, $\sigma_{\bar{x}} = 0.006$).

For a real-world workload, we evaluated the runtime overhead using the SPEC CPU 2017 benchmark. We ran the benchmark once with our mitigation and once without it on an Intel Xeon Silver 4208. We excluded some benchmarks in both the *intspeed* and *fpspeed* benchmark as they already crashed or did not compile on our vanilla Linux system. Figure 7 shows the results. As expected by the design of FLARE, we exhibit next to no runtime overhead.

Module Loading. Next, we evaluated the increase in module loading time. We first establish a baseline by loading and unloading a simple test module 10^4 times. We then load the FLARE proof of concept, which requires removing and re-applying the dummy mapping for every module load, thus overapproximating the overhead of our mitigation. We again load and measure the required time 10^4 times. We only observe a 4 % increase from 2.39 ms to 2.48 ms per module load. When implemented in the Linux kernel, the module memory allocation logic is made aware of the dummy mappings so that they are treated like free memory. Thus, overheads are avoided entirely except in cases where the modules have to be re-padded, where we observe the overheads to be negligible.

Memory. Finally, we analyzed the memory overhead of FLARE, which is minimal in our proof of concept. We always map the same dummy page in the paging hierarchy and re-use the same page directory and page table. We do not need a new PDPT, as we are working on existing 1 GB ranges. Therefore, we only require one page each for the new page directory, page table, and one page to point to. As all these pages are 4 kB, the maximum overhead is 12 kB. To map huge dummy pages, the maximum overhead is only increased by 2 MB. The direct-physical map padding with 1 GB pages does not consume additional memory (cf. Section 4).

5.2 Mitigate Microarchitectural KASLR Breaks

In a first step, we evaluated the effectiveness of FLARE in preventing breaking the randomization of the kernel text segment. Using a vanilla Linux 5.0 kernel, we test microarchitectural attacks on KASLR that are not mitigated through orthogonal countermeasures (cf. Section 3.5) with KPTI disabled (cf. Figure 8). In all cases, we first establish a baseline of the attack without FLARE in place, which shows the exact position of the kernel with all attacks.

We then load FLARE and re-evaluate all attacks. We see for each attack that the kernel can no longer be distinguished from other positions. With EchoLoad (Figure 8a), all offsets are backed by a physical page, the load succeeds, but the CPU returns zero for the illegal access. The stall percentage is based on cache hits and misses on the probe array, not performance counter values. With the prefetch side channel (Figure 8b), we see that the prefetch instruction can now also prefetch all other possible locations, mitigating the KASLR break. Data Bounce (Figure 8c) also no longer distinguishes kernel locations from dummy mappings as store-to-load forwarding works for all possible offsets. The double-page fault (Figure 8d) as well as the DrK attack (Figure 8e) also do not work anymore, exhibiting the same timing across the whole address range. With Fallout (Figure 8f), we also see no difference anymore as every page allows to trigger the WTF effect. An attack that tries to detect our dummy mapping based on timing the page-table walk is also not possible. Even though the physical page is shared across all dummy mappings, a TLB entry for one mapping is not shared with another. Hence, each access to a new page requires a full page-table walk. Our dummy mapping can also not be uncoverd via the cache as an access to a privileged address does not load the data into it [42, 76, 79]. Based on the results shown in Figure 8, none of the currently known microarchitectural attacks that are not mitigated through orthogonal countermeasures (cf. Section 3.5) can de-randomize the kernel location despite FLARE. This empirically confirms that we solve challenge C1.

Next, we de-randomize the kernel based on the timing difference between executable and NX pages [49]. We confirm that tackling only C1 and C2 is insufficient (cf. Figure 9). However, full FLARE (cf. Figure 9) separates the regions and the switch from executable to NX is not visible in this region anymore but at the pre-defined start of the randomization range (cf. Figure 6).

Next, we used the prefetch side-channel attack to try to break KASLR based on different page sizes. The different levels visible in the default case of Figure 8b show the different paging levels for the address we test. If nothing is mapped in the PML4, we observe the highest time. There is a drop in the access time for addresses with no PDPT entry, and another drop for addresses that map to an entry in the page table, *i.e.*, a 4 kB page. Thus, the prefetch side channel shows the different paging levels [32]. With FLARE in place, we can no longer see the difference in page sizes as all possible locations as well as the kernel are mapped using 4 kB pages. Thus, we empirically confirmed that our strategy for C3 works, defeating microarchitectural attacks on KASLR based on different page sizes.

6 RELATED WORK

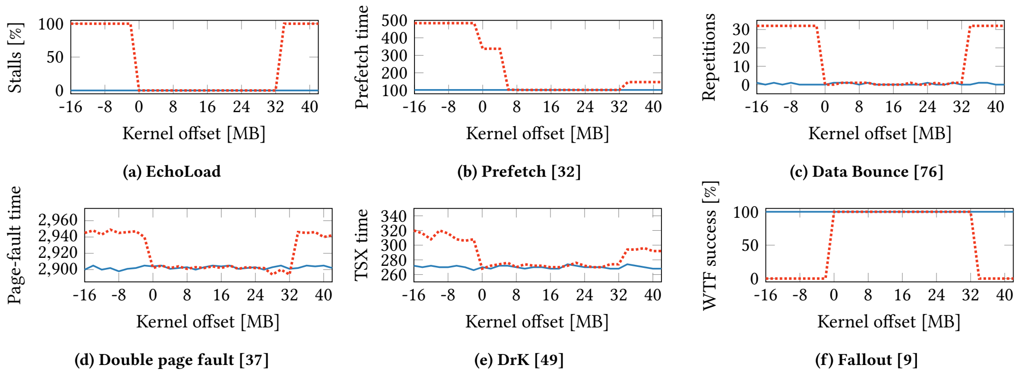
With the advent of KASLR, many different attacks have been proposed to break KASLR. One problem is that the kernels of the major OSs cannot change the randomization at runtime. Hence, if an attacker knows the KASLR offset, it is valid until the next time the OS is rebooted. So far, most of the attacks on KASLR relied either on software vulnerabilities or side-channel attacks on the microarchitecture as discussed in Section 3.5.

6.1 Software-based KASLR Breaks

On Linux, parts of kernel pointers are often disclosed inadvertently through kernel interfaces, e.g., due to uninitialized structure fields or structure padding [84]. There have been many such software vulnerabilities in kernels (e.g., CVE-2012-6138, CVE-2013-1825, 1826, 1827, 1873, 2634, 2635, 2636) that revealed parts of kernel addresses. Similarly, for Windows, multiple methods leak kernel pointers, e.g., using the `Win32ThreadInfo` or the Desktop heap [74]. Other attacks on KASLR relied on the fact that kernel addresses were used as unique identifiers [84], or as seed for pseudo-random numbers [52]. For debugging reasons, kernel addresses were often visible in log files or debugging interfaces such as the perf subsystem [18].

6.2 Mitigating Software-based KASLR Attacks

While software bugs causing KASLR breaks can be easily fixed, there are also general concepts for preventing address leakage from the kernel. Linux introduced a setting to mask kernel pointers in log files with a random mask [73]. Thus, a developer sees which pointers are the same, but an attacker cannot learn the actual pointer value and, thus, the KASLR offset. This mitigation reduces the risk of leaking the KASLR offset without impairing the debugging capabilities. The PaX Team proposed STACKLEAK [15], a mechanism to



(a) EchoLoad (b) Prefetch [32] (c) Data Bounce [76]

(d) Double page fault [37] (e) DrK [49] (f) Fallout [9]

Figure 8: Detecting the kernel (offset 0 to 32 MB) with all known microarchitectural attacks on KASLR without FLARE (⋯⋯) and with FLARE (—). For all attacks, FLARE successfully prevents the detection of the kernel.

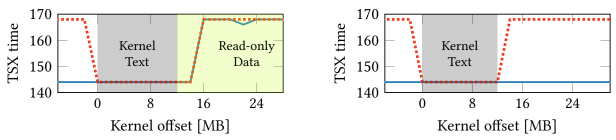


Figure 9: DrK [49] distinguishes executable from NX pages. An attacker can observe the switch to NX pages directly after the executable pages (left), when tackling only C1 and C2 (—) and entirely without (⋯⋯) FLARE. With full FLARE (right), this attack is also fully mitigated.

clear kernel-stack memory which is no longer in use. This reduces accidental address leakage from uninitialized stack values.

6.3 Mitigate Microarchitectural KASLR Breaks

While microarchitectural attacks on KASLR cannot be simply fixed in software, there are software-based workarounds. Gruss et al. [31, 32] proposed stronger kernel isolation to prevent microarchitectural attacks on the kernel by unmapping the kernel address space when running in user space. Thus, in theory, there is no valid kernel address in user space, preventing all microarchitectural attacks on the kernel. However, while their proposal is deployed on all major OSs to prevent Meltdown [59], it cannot prevent our KASLR break (cf. Section 3.2). The reason is that x86 requires some kernel pages always to be mapped, even when running in user space [31].

Lazarus [26] proposed a similar approach to KAISER [31]. It is based on fencing the kernel paging entries off from those of the user space by separating user and kernel page tables. Therefore, the Memory Management Unit can no longer use entries pointing to kernel space memory from user space. Contrary to KAISER, Lazarus uses dummy mappings to hide the context switching code while KAISER separates it from the rest of the kernel code section. However, it does not tackle the challenges we identified and does not defeat all known microarchitectural attacks on KASLR.

7 CONCLUSION

In this paper, we analyzed Intel's recent hardware fixes for Meltdown. Our analysis led to the understanding that illegal memory accesses do not lead to a CPU stall, but instead, the illegally loaded data is zeroed-out. With EchoLoad, we presented a novel technique based on Flush+Reload to distinguish stalling loads from transiently executed ones. Hence, EchoLoad enables an attacker to detect physically-backed kernel addresses and break KASLR. Our KASLR break is the fastest and most reliable microarchitectural KASLR break presented so far, taking only 40 µs to de-randomize the kernel. The only requirement for EchoLoad are memory loads, allowing it to be mounted from restricted environments such as SGX and JavaScript. We presented the first JavaScript-based Meltdown attack and KASLR break on the systems that do not receive Meltdown patches, *i.e.*, x86 32-bit OSs.

With FLARE, we proposed a generic approach for protecting the kernel against microarchitectural KASLR breaks. We verified that FLARE mitigates the root cause behind current microarchitectural KASLR breaks and yields a uniform behavior across the kernel address space. Thus, considering the state of the hardware mitigations, we propose to deploy FLARE even on the most recent CPU generations.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their comments and suggestions that helped improving the paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). 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. Additional funding was provided by generous gifts from Intel, ARM, and Cloudflare. 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] Tiago Alves. 2004. TrustZone: Integrated Hardware and Software Security.
[2] Apple Inc. 2012. OS X Mountain Lion Core Technologies Overview. http://movies.apple.com/media/us/osx/2012/docs/OSX_MountainLion_Core_Technologies_Overview.pdf
[3] ARM Limited. 2018. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism.
[4] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. 2014. Ooh Aah... Just a Little Bit: A small amount of side channel can go a long way. In *CHES*.
[5] Daniel J. Bernstein. 2004. Cache-Timing Attacks on AES.

[6] Erik Bosman and Herbert Bos. 2014. Framing Signals - A Return to Portable Shellcode. In *S&P.*

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

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

[9] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS.*

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium.*

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

[12] Nicholas Carlini and David A. Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security.*

[13] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *CCS.*

[14] Liang Chen and Qidan He. 2016. Shooting the OS X El Capitan Kernel Like a Sniper.

[15] Jonathan Corbet. 2018. Preventing kernel-stack leaks. https://lwn.net/Articles/748642/

[16] Standard Performance Evaluation Corporation. 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/

[17] Ian Cutress. 2018. 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

[18] Lizzie Dixon. 2017. Breaking KASLR with perf. https://blog.lizzie.io/kaslr-and-perf.html

[19] ecma international. 2018. ECMAScript 2018 Language Specification. https://www.ecma-international.org/ecma-262/9.0/index.html

[20] Jake Edge. 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/

[21] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO.*

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

[23] Ulf Frisk. 2016. Windows 10 KASLR Recovery with TSX. http://blog.frizk.net/2016/11/windows-10-kaslr-recovery-with-tsx.html

[24] Thomas Garnier. 2016. Kernel memory randomization and trampoline page tables. https://medium.com/@mxatone/kernel-memory-randomization-and-trampoline-page-tables-9f73827270ab

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

[26] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. 2017. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In *RAID.*

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

[28] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *S&P.*

[29] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS.*

[30] Daniel Gruss, Dave Hansen, and Brendan Gregg. 2018. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. *USENIX ;login* (2018).

[31] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *ESSoS.*

[32] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS.*

[33] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA.*

[34] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium.*

[35] Noam Hadad and Jonathan Afek. 2018. Overcoming (some) Spectre browser mitigations. https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/

[36] Jann Horn. 2018. speculative execution, variant 4: speculative store bypass.

[37] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P.*

[38] Intel. [n.d.]. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

[39] Intel. 2018. 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

[40] Intel. 2018. Speculative Execution Side Channel Mitigations. Revision 3.0.

[41] Intel. 2019. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling

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

[43] Intel. 2019. Performance Monitoring Impact of Intel Transactional Synchronization Extension Memory. https://cdrdv2.intel.com/v1/dl/getContent/604224

[44] Alex Ionescu. 2016. Twitter: Windows KASLR. https://twitter.com/aionescu/status/725399988306644992

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

[46] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14.*

[47] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *CCS.*

[48] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX.*

[49] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS.*

[50] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD Memory Encryption.

[51] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).

[52] Amit Klein and Benny Pinkas. 2019. From IP ID to Device ID and KASLR Bypass. In *USENIX Security.*

[53] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P.*

[54] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO.*

[55] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT.*

[56] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *USENIX Security Symposium.*

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

[58] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium.*

[59] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium.*

[60] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P.*

[61] G. Maisuradze and C. Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS.*

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

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

[64] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *USENIX ATC.*

[65] Andrei Mogage, Rafael Pires, Vlad Crăciun, Pascal Felber, and Emanuel Onica. 2019. Supply chain malware targets SGX: Take care of what you sign (Practical Experience Report). In *SRDS.*

[66] Mozilla. 2019. Index Masking in Firefox. https://bugzilla.mozilla.org/show_bug.cgi?id=1430051

[67] Mozilla. 2019. performance.now resolution. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

[68] Net Applications.com. 2019. Desktop Operating System Market Share. http://www.netmarketshare.com/operating-system-market-share.aspx

[69] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA.*

[70] Matthew Panzarino. 2012. Apple releases OS X 10.8 Mountain Lion Developer Preview 2, lists known issues. https://thenextweb.com/apple/2012/03/16/apple-releases-os-x-10-8-mountain-lion-developer-preview-2-to-mac-developers/

[71] Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*.

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

[73] Dan Rosenberg. 2010. kptr_restrict for hiding kernel pointers. https://lwn.net/Articles/420403/

[74] Morten Schenk. 2019. Development of a new Windows 10 KASLR Bypass (in One WinDBG Command). https://www.offensive-security.com/vulndev/development-of-a-new-windows-10-kaslr-bypass-in-one-windbg-command/

[75] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *S&P*.

[76] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. 2019. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725* (2019).

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

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

[79] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.

[80] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*.

[81] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical Enclave Malware with Intel SGX. In *DIMVA*.

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

[83] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*.

[84] Brad Spengler. 2013. KASLR: An Exercise in Cargo Cult Security. https://grsecurity.net/kaslr_an_exercise_in_cargo_cult_security.php

[85] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480* (2018).

[86] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *S&P*.

[87] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.

[88] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[89] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[90] Pepe Vila, Boris Köpf, and Jose Morales. 2019. Theory and Practice of Finding Eviction Sets. In *S&P*.

[91] WebAssembly. 2019. *Features to add after the MVP*. https://github.com/WebAssembly/design/blob/master/FutureFeatures.md

[92] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS*.

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

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

[95] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An exploration of L2 cache covert channels in virtualized environments. In *CCSW'11*.

[96] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*.

[97] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*.

[98] Peter Zijlstra. 2019. Implement support for TSX Force Abort. https://lkml.org/lkml/2019/3/12/1352