ZombieLoad: Cross-Privilege-Boundary Data Sampling

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

Moritz Lipp
Graz University of Technology
moritz.lipp@iaik.tugraz.at

Daniel Moghimi
Worcester Polytechnic Institute
amoghimi@wpi.edu

Jo Van Bulck
imec-DistriNet, KU Leuven
jo.vanbulck@cs.kuleuven.be

Julian Stecklina
Cyberus Technology
julian.stecklina@cyberus-technology.de

Thomas Prescher
Cyberus Technology
thomas.prescher@cyberus-technology.de

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

ABSTRACT

In early 2018, Meltdown first showed how to read arbitrary kernel memory from user space by exploiting side-effects from transient instructions. While this attack has been mitigated through stronger isolation boundaries between user and kernel space, Meltdown inspired an entirely new class of fault-driven transient-execution attacks. Particularly, over the past year, Meltdown-type attacks have been extended to not only leak data from the L1 cache but also from various other microarchitectural structures, including the FPU register file and store buffer.

In this paper, we present the ZombieLoad attack which uncovers a novel Meltdown-type effect in the processor's fill-buffer logic. Our analysis shows that faulting load instructions (*i.e.*, loads that have to be re-issued) may transiently dereference unauthorized destinations previously brought into the fill buffer by the current or a sibling logical CPU. In contrast to concurrent attacks on the fill buffer, we are the first to report data leakage of recently loaded and stored stale values across logical cores even on Meltdown- and MDS-resistant processors. Hence, despite Intel's claims [36], we show that the hardware fixes in new CPUs are not sufficient. We demonstrate ZombieLoad's effectiveness in a multitude of practical attack scenarios across CPU privilege rings, OS processes, virtual machines, and SGX enclaves. We discuss both short and long-term mitigation approaches and arrive at the conclusion that disabling hyperthreading is the only possible workaround to prevent at least the most-powerful cross-hyperthread attack scenarios on current processors, as Intel's software fixes are incomplete.

CCS CONCEPTS

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

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354252

KEYWORDS

side-channel attack, transient execution, fill buffer, Meltdown

ACM Reference Format:
Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3354252

1 INTRODUCTION

In 2018, Meltdown [46] was the first microarchitectural attack completely breaching the security boundary between the user and kernel space and, thus, allowed to leak arbitrary data. While Meltdown was fixed using a stronger isolation between user and kernel space, the underlying principle turned out to be an entire class of transient-execution attacks [7]. Over the past year, researchers demonstrated that Meltdown-type attacks cannot only leak kernel data to user space, but also leak data across user processes, virtual machines, and SGX enclaves [70, 77]. Furthermore, leakage is not limited to the L1 cache but can also originate from other microarchitectural structures, such as the register file [69] and, as shown in concurrent work, the fill buffer [74], load ports [74], and the store buffer [54].

Instead of executing the instruction stream in order, most modern processors can re-order instructions while maintaining architectural equivalence. Instructions may already have been executed when the CPU detects that a previous instruction raises an exception. Hence, such instructions following the faulting instruction (*i.e.*, transient instructions) are rolled back. While the rollback ensures that there are no architectural effects, side effects might remain in the microarchitectural state. Most Meltdown-type attacks exploit overly aggressive optimizations around out-of-order execution.

For many years, the microarchitectural state was considered invisible to applications, and hence security considerations were often limited to the architectural state. Specifically, microarchitectural elements often do not distinguish between different applications or privilege levels [7, 12, 38, 46, 58, 62, 65].

In this paper, we show that, first, there still are unexplored microarchitectural buffers, and second, both architectural and microarchitectural faults can be exploited. With our notion of "microarchitectural faults", *i.e.*, faults that cause a memory request to be re-issued internally without ever becoming architecturally visible, we demonstrate that Meltdown-type attacks can also be triggered without raising an architectural exception such as a page fault. Based on this, we demonstrate ZombieLoad, a novel, extremely powerful Meltdown-type attack targeting the fill-buffer logic.

ZombieLoad exploits that load instructions which have to be re-issued internally, may first transiently compute on stale values belonging to previous memory operations from either the current or a sibling hyperthread. Using established transient-execution attack techniques, adversaries can recover the values of such "zombie load" operations. Importantly, in contrast to all previously known transient-execution attacks [7], ZombieLoad reveals recent data values *without* adhering to any explicit address-based selectors. Hence, we consider ZombieLoad an instance of a novel type of *microarchitectural data sampling* (MDS) attacks. Unlike concurrent data sampling attacks like RIDL [74] or Fallout [54], our work includes the first and only attack variant that can leak data even on the most recent Intel Cascade Lake CPUs which are reportedly resistant against all known Meltdown, Foreshadow, and MDS variants. We present microarchitectural data sampling as the missing link between traditional memory-based side-channels which correlate data addresses within a victim execution, and existing Meltdown-type transient-execution attacks that can directly recover data values belonging to an explicit address. In this paper, we combine primitives from traditional side-channel attacks with incidental data sampling in the time domain to construct extremely powerful attacks with targeted leakage in the address domain. This not only opens up new attack avenues but also re-enables attacks that were previously assumed mitigated.

We demonstrate ZombieLoad's real-world implications in a multitude of practical attack scenarios that leak across processes, privilege boundaries, and even across logical CPU cores. Furthermore, we show that we can leak Intel SGX enclave secrets loaded from a sibling logical core, even on Foreshadow-resistant CPUs. We demonstrate that ZombieLoad attackers may extract sealing keys from Intel's architectural quoting enclave, ultimately breaking SGX's confidentiality and remote attestation guarantees. ZombieLoad is furthermore not limited to native code execution, but also works across virtualization boundaries. Hence, virtual machines can attack not only the hypervisor but also different virtual machines running on a sibling logical core. We conclude that disabling hyperthreading, in addition to flushing several microarchitectural states during context switches, is the only possible workaround to prevent this extremely powerful attack.

Contributions. The main contributions of this work are:

(1) We present ZombieLoad, a powerful data sampling attack leaking data accessed on the same or sibling hyperthread.
(2) We combine incidental data sampling in the time domain with traditional side-channel primitives to construct a targeted information flow similar to regular Meltdown attacks.
(3) We demonstrate ZombieLoad in several real-world scenarios: cross-process, cross-VM, user-to-kernel, and SGX. ZombieLoad even works on Meltdown-resistant hardware.
(4) We show that ZombieLoad breaks the security guarantees of Intel SGX, even on Foreshadow-resistant hardware.
(5) We are the first to do post-processing of the leaked data within the transient domain to eliminate noise.

Outline. Section 2 provides background. Section 3 gives an overview of ZombieLoad, and introduces a novel classification for memory-based side-channel attacks. Section 4 describes attack scenarios and their attacker models. Section 5 introduces and evaluates the basic primitives required for mounting ZombieLoad. Section 6 demonstrates ZombieLoad in real-world attack scenarios. Section 7 discusses possible countermeasures. We conclude in Section 8.

Responsible Disclosure. We reported leakage of uncacheable-typed memory from a concurrent hyperthread on March 28, 2018, to Intel. We clarified on May 30, 2018 that we attribute the source of this leakage to the LFB. In our experiments, this works identically for Foreshadow, undermining the completeness of L1-flush-based mitigations. This issue was acknowledged by Intel and tracked under CVE-2019-11091 (MDSUM). We responsibly disclosed ZombieLoad Variant 1 to Intel on April 12, 2019. Intel verified and acknowledged our attack and assigned CVE-2018-12130 (MFBDS) to this issue. Both MDSUM and MFBDS were part of the Microarchitectural Data Sampling (MDS) embargo ending on May 14, 2019. We responsibly disclosed ZombieLoad Variant 2 (which is the only MDS attack that works on Cascade Lake CPUs) to Intel on April 24, 2019. This issue, which Intel refers to as Transactional Asynchronous Abort (TAA) is assigned CVE-2019-11135 and is part of an ongoing embargo ending on November 12, 2019. On May 16, 2019, we reported to Intel that their mitigations using `VERW` are incomplete and can be circumvented, which they verfied and acknowledged.

2 BACKGROUND

In this section, we describe the background required for this paper.

2.1 Transient Execution Attacks

Today's high-performance processors typically implement an *out-of-order execution* design, allowing the CPU to utilize different execution units in parallel. The instruction stream is decoded *in-order* into simpler micro-operations (μOPs) [13] which can be executed as soon as the required operands are available. A dedicated reorder buffer stores intermediate results and ensures that instruction results are committed to the architectural state in-order. Any fault that occurred during the execution of an instruction is handled at instruction retirement, leading to a pipeline flush which squashes any outstanding μOP results from the reorder buffer.

In addition, modern CPUs employ *speculative execution* optimizations to avoid stalling the instruction pipeline until a conditional branch is resolved. The CPU predicts the outcome of the branch and continues execution along that direction. We refer to instructions that are executed speculatively or out-of-order but whose results are never architecturally committed as *transient instructions* [7, 46, 70].

While the results and the architectural effects of transient instructions are discarded, measurable microarchitectural side effects

may remain and are not reverted. Attacks that exploit these side effects to observe sensitive information are called *transient execution attacks* [7, 43, 46]. Typically, these attacks utilize a cache-based covert channel to transmit the secret data observed transiently from the microarchitectural domain to an architectural state. In line with a recent exhaustive survey [7], we refer to attacks exploiting misprediction [27, 41, 43, 44, 50] as Spectre-type, whereas attacks exploiting transient execution after a CPU exception [7, 41, 46, 69, 70, 77] are classified as belonging to Meltdown-type.

2.2 Memory Subsystem

In this section, we overview memory loads in out-of-order CPUs.

Caches. CPUs contain small and fast caches storing frequently used data. Caches are typically organized in multiple levels that are either private per core or shared amongst them. Modern CPUs typically use n-way set-associative caches containing n cache lines per set, each typically 64 B wide. Usually, Intel CPUs have a private first-level instruction (L1I) and data cache (L1D) and a unified L2 cache. The last-level cache (LLC) is shared across all cores.

Virtual Memory. CPUs use virtual memory to provide memory isolation between processes. Virtual addresses are translated to physical memory locations using multi-level translation tables. The translation table entries define the properties, e.g., access control or memory type, of the referenced memory region. The CPU contains the translation-look-aside buffer (TLB) consisting of additional caches to store address-translation information.

Memory Order Buffer. μOPs dealing with memory operations are handled by dedicated execution units. Typically, Intel CPUs contain 2 units responsible for loading and one for storing data. The *memory order* buffer (MOB), incorporating a *load buffer* and a *store buffer*, controls the dispatch of memory operations and tracks their progress to resolve memory dependencies.

Data Loads. For every dispatched load operation an entry is allocated in the load buffer and the reorder buffer. To determine the physical address, the upper 36 bit of the linear address are translated by the memory management unit. Concurrently, the untranslated lower 12 bit are already used to index the cache set in the L1D [17]. If the address translation is in the TLB, the physical address is available immediately. Otherwise, the page miss handler (PMH) performs a page-table walk to retrieve the address translation as well as the corresponding permission bits. If the requested data is in the L1D (cache hit), the load operation can be completed.

If data is not in the L1D, it needs to be served from higher levels of the cache or the main memory via the line-fill buffer (LFB). The LFB serves as an interface to other caches and the main memory and keeps track of outstanding loads. Memory accesses to uncacheable memory regions, and non-temporal moves all go through the LFB.

On a fault, e.g., a physical address is not available, the page-table walk does not immediately abort [17]. An instruction in a pipelined implementation must undergo each stage and is simply reissued in case of a fault [1]. Only at the retirement of the faulting μOP, the fault is handled, and the pipeline is flushed [16, 17].

2.3 Processor Extensions

Microcode. To support more complex instructions, *microcode* allows implementing higher-level instructions using multiple hardware-level instructions. This allows processor vendors to support complex behavior and even extend or modify CPU behavior through microcode updates [28]. Preferably, new architectural features are implemented as microcode extensions, e.g., Intel SGX [39].

While the execution units perform the fast-paths directly in hardware, more complex slow-path operations, such as faults or page-table modifications, are typically performed by issuing a *microcode assist* which points the sequencer to a predefined microcode routine [11]. To do so, the execution unit associates an event code with the result of the faulting micro-op. When the micro-op of the execution unit is committed, the event code causes the out-of-order scheduler to squash all in-flight micro-ops in the reorder buffer [11]. The microcode sequencer uses the event code to read the micro-ops associated with the event in the microcode [5].

Intel TSX. Intel TSX is an x86 instruction set extension for hardware transactional memory [35] introduced with Intel Haswell CPUs. With TSX, particular code regions are executed transactionally. If the entire code regions completes successfully, memory operations within the transaction appear as an atomic commit to other logical processors. If an issue occurs during the transaction, a transactional abort rolls back the execution to an architectural state before the transaction, discarding all performed operations. Transactional aborts can be caused by different issues: Typically, a conflicting memory operation occurs where another logical processor either reads from an address which has been modified within the transaction or writes to an address which is used within the transaction. Further, the amount of read and written data within the transaction may not exceed the size of the LLC and L1 cache respectively [28]. In addition, some instructions or system event might cause the transaction to abort as well [35].

Intel SGX. With the Skylake microarchitecture, Intel introduced Software Guard Extension (SGX), an instruction-set extension for isolating trusted code [28]. SGX executes trusted code inside so-called *enclaves*, which are mapped in the virtual address space of a conventional host application process but are isolated from the rest of the system by the hardware itself. The threat model of SGX assumes that the operating system and all other running applications could be compromised and, therefore, cannot be trusted. Any attempt to access SGX enclave memory in non-enclave mode results in a dummy value `0xff` [29]. Furthermore, to protect against physical attackers probing the memory bus, the SGX hardware transparently encrypts the used memory region [11].

A dedicated `eenter` instruction redirects control flow to an enclave entry point, whereas `eexit` transfers back to the untrusted host application. Furthermore, in case of an interrupt or fault, SGX securely saves CPU registers inside the enclave's save state area (SSA) before vectoring to the untrusted operating system. Next, the `eresume` instruction can be used to restore processor state from the SSA frame and continue a previously interrupted enclave.

SGX-capable processors feature cryptographic key derivation facilities through the `egetkey` instruction, based on a CPU-level

master secret and a secure measurement of the calling enclave's initial code and data. Using this key, enclaves can securely *seal* secrets for untrusted persistent storage, and establish secure communication channels with other enclaves residing on the same processor. Furthermore, to enable remote attestation, Intel provides a trusted *quoting enclave* which unseals an Intel-private key and generates an asymmetric signature over the local enclave identity report.

Over the past years, researchers have demonstrated various attacks to leak sensitive data from SGX enclaves, e.g., through memory safety violations [45], race conditions [76], or side-channels [55, 65, 72, 73]. More recently, SGX was also compromised by transient-execution attacks [9, 70] which necessitated microcode updates and increased the processor's security version number (SVN). All SGX key derivations and attestations include SVN to reflect the current microcode version, and hence security level.

3 ATTACK OVERVIEW

In this section, we provide an overview of ZombieLoad. We describe what can be observed using ZombieLoad and how that fits into the landscape of existing side-channel attacks. By that, we show that ZombieLoad is a novel category of side-channel attacks, which we refer to as *data-sampling attacks*, opening a new research field.

3.1 Overview

ZombieLoad is a transient-execution attack [7] which observes the values of memory loads and stores on the current CPU core. ZombieLoad exploits that the fill buffer is used by all logical CPUs of a CPU core and that it does not distinguish between processes or privileges.

Whenever the CPU encounters a memory load during execution, it reserves an entry in the load buffer. If the load was not an L1 hit, it requires a fill-buffer entry. When the requested data has been loaded, the memory subsystem frees the corresponding load- and fill-buffer entries, and the load instruction may retire. Similarly, if stores miss the L1 or are evicted from the L1, they are temporarily stored in a fill-buffer entry as well.

However, we observed that under certain complex microarchitectural conditions (e.g., a fault), where the load requires a microcode assist, it may first read stale values before being re-issued eventually. As with any Meltdown-type attack, this opens up a transient-execution window where this value can be used for subsequent calculations. Thus, an attacker can encode the leaked value into a microarchitectural element, such as the cache.

In contrast to previous Meltdown-type attacks, however, it is not possible to select the value to leak based on an attacker-specified address. ZombieLoad simply leaks any value which is currently loaded or stored by the physical CPU core. While this at first sounds like a massive limitation, we show that this opens a new field of data sampling-based transient-execution attacks. Moreover, in contrast to previous Meltdown-type attacks, ZombieLoad considers all privilege boundaries and is not limited to a specific one. Meltdown [46] can only leak data from the attacker's address space, Foreshadow [70] focussed exclusively on SGX enclaves, Foreshadow-NG [77] afterwards investigated cross-process and cross-VM leakage, and Fallout [54] can only leak kernel data on the same logical core. We show that ZombieLoad is an even more powerful attack in combination with existing side-channel techniques.

3.2 Microarchitectural Root Cause

For Meltdown, Foreshadow, Fallout, and RIDL, the source of the leakage is apparent. Moreover, for these attacks, there are plausible explanations on what is going wrong in the microarchitecture, *i.e.*, what the root cause of the leakage is [46, 54, 70, 77]. For ZombieLoad, however, this is not entirely clear.

While we identified some necessary building blocks to observe the leakage (cf. Section 5), we can only provide a hypothesis on why the interaction of the building blocks leads to the observed leakage. As we could only observe data leakage on Intel CPUs, we assume that this is indeed an implementation issue (such as Meltdown) and not a design issue (as with Spectre). For our hypothesis, we combined our observations with the little official documentation of the fill buffer [28, 34] and Intel's MDS analysis [33]. Ultimately, we could neither prove nor disprove our hypothesis, leaving the verification or falsification of our hypothesis to future work.

Stale-Entry Hypothesis. Every load is associated with an entry in the load buffer and potentially an entry in the fill buffer [34].

When a load encounters a complex situation, such as a fault, it requires a microcode assist [28]. This microcode assist triggers a machine clear, which flushes the pipeline. On a pipeline flush, instructions which are already in flight still finish execution [26].

As this has to be as fast as possible to not incur additional delays, we expect that fill-buffer entries are optimistically matched as long as parts of the physical address match. Thus, the load continues with a wrong fill-buffer entry, which was valid for a previous load or store. This leads to a use-after-free vulnerability [22] in the hardware. Intel documents the fill buffer as being competitively shared among hyperthreads [28], giving both logical cores access to the entire fill buffer (cf. Appendix A). Consequently, the stale fill-buffer entry can also be from a previous load or store of the sibling logical core. As a result, the load instruction loads valid data from a previous load or store.

Leakage Source. We devised 2 experiments to reduce the number of possible sources of the leaked data.

In our first experiment, we marked a page as "uncacheable" and flushed it from the cache. As a result, every memory load from the page circumvents all cache levels and goes directly to the fill buffer [28]. We then write the secret onto the uncacheable page to ensure that there is no copy of the data in the cache. When loading data from the page, we see leakage in the order of bytes per second, e.g., 5.91 B/s ($\sigma_{\bar{x}} = 0.18$, $n = 100$, where n is the number of experiments and $\sigma_{\bar{x}}$ is the standard error of the mean) on an i7-8650U. We can attribute this leakage to the fill buffer. This was also exploited in concurrent work [74]. Our hypothesis is further backed by the `MEM_LOAD_RETIRED.FB_HIT` performance counter, which shows multiple thousand line-fill-buffer hits (117 330 FB_HIT/s ($\sigma_{\bar{x}} = 511.57$, $n = 100$)).

Intel claims that the leakage is entirely from the fill buffer [33]. This is also what Van Schaik et al. [74] conclude for their RIDL attack. However, our second experiment shows that the line-fill buffer might not be the only source of the leakage for ZombieLoad. We rely

on Intel TSX to ensure that memory accesses do not reach the line-fill buffer as follows. Inside a transaction, we first write the secret value to a memory location which was previously initialized with a different value. The write inside the transaction ensures that the address is in the *write set* of the transaction and thus in L1 [34, 61]. Evicting data from the write set from the cache leads to a transactional abort [34]. Hence, any subsequent memory access to the data from the write set ensures that it is served from the L1, and therefore, no request to the line-fill buffer is sent [28]. In this experiment, we see a much higher rate of leakage, which is in the order of kilobytes per second. More importantly, we only see the value written inside the TSX transaction and not the value that was at the memory location before starting the transaction. Our hypothesis that the line-fill buffer is not the only source of the leakage is further backed by observing performance counters. The `MEM_LOAD_RETIRED.FB_HIT` and `MEM_LOAD_RETIRED.L1_MISS` performance counters do not increase significantly. In contrast, the `MEM_LOAD_RETIRED.L1_HIT` performance counter shows multiple thousand L1 hits.

While accessing the data to leak on the victim core, we monitored the `MEM_LOAD_RETIRED.FB_HIT` performance counter on the attacker core for 10 s. If the address was cached, we measured a Pearson correlation of $r_p = 0.02$ ($n = 100$) between the correct recoveries and line-fill buffer hits, indicating no association. However, while continuously flushing the data on the victim core, ensuring that a subsequent access must go through the LFB, we measure a strong correlation of $r_p = 0.86$ ($n = 100$). This result indicates that the line-fill buffer is not the only source of leakage. However, a different explanation might be that the performance counters are not reliable in such corner cases. Van Schaik et al. [74] reported that the RIDL attack can only leak data which is not served from the cache, *i.e.*, which has to go through the fill buffers. Hence, we conclude that RIDL indeed leaks from fill buffers, whereas the ZombieLoad leakage might not be entirely attributed to the fill buffer. Future work has to investigate whether other microarchitectural elements, e.g., the load buffer, are also involved in the observed data leakage.

Comparison to RIDL. In concurrent work, Van Schaik et al. [74] presented the RIDL attack, which also leaks data from the fill buffers, as well as from the load ports. Table 1 shows a table which summarizes the main differences between RIDL and ZombieLoad. The most crucial difference between the attacks is that ZombieLoad still works on the newest generation of Intel CPUs (Cascade Lake with stepping B1) which are not affected by RIDL or Fallout. RIDL can only leak loads which are not currently in the L1 cache. ZombieLoad can leak all loads, independent whether they are currently in the L1 cache or not. ZombieLoad has a thorough analysis of the microarchitectural root cause, which leads to more variants with unique features, such as leakage on an MDS-resistant CPU.

3.3 Classification

In this section, we introduce a way to classify memory-based side-channel and transient-execution attacks. For all these attacks, we assume a target program which executes a memory operation at a certain *address* with a specific data *value* at the program's current *instruction pointer*. Figure 1 illustrates these three properties as the corner of a triangle, and techniques which let an attacker infer one of the properties based on one or both of the other properties.

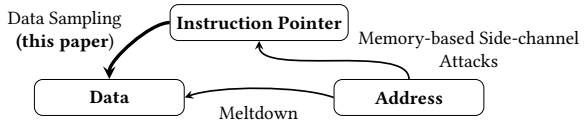

Figure 1: **The 3 properties of a memory operation: instruction pointer of the program, target address, and data value. So far, there are techniques to infer the instruction pointer from target address and the data value from the address. With ZombieLoad, we show the first instance of an attack which infers the data value from the instruction pointer.**

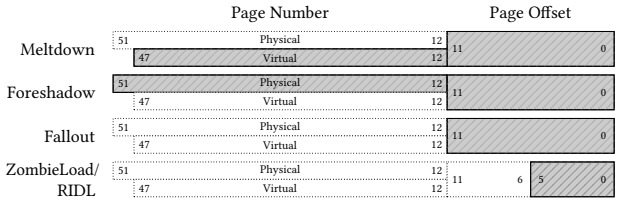

Figure 2: **Meltdown-type attacks provide a varying degree of target control (gray hatched), from full virtual addresses in the case of Meltdown to nearly no control for ZombieLoad.**

Traditional memory-based side-channel attacks allow an attacker to observe the location of memory accesses. The granularity of the location observation depends on the spatial accuracy of the used side channel. Most common memory-based side-channel attacks [18, 20, 21, 23, 38, 57, 58, 73, 80, 81] have a granularity between one cache line [20, 21, 23, 81] *i.e.*, usually 64 B, and one page [18, 38, 73, 80], *i.e.*, usually 4 kB. These side channels establish a connection between the time domain and the space domain. The time domain can either be the wall time or also commonly the execution time of the program which correlates with the instruction pointer. These classic side channels provide means of connecting the address of a memory access to a set of possible instruction pointers, which then allows reconstructing the program flow. Thus, side-channel resistant applications have to avoid secret-dependent memory access to not leak secrets to a side-channel attacker.

Since early 2018, with transient-execution attacks [7] such as Meltdown [46] and Spectre [43], there is a second type of attacks which allow an attacker to observe the value stored at a memory address. Meltdown provided the most control over target address. With Meltdown, the full virtual address of the target data is provided, and the corresponding data value stored at this address is leaked. The success rate depends on the location of the data, *i.e.*, whether it is in the cache or main memory. However, the only constraint for Meltdown is that the data is addressable using a virtual address [46]. Other Meltdown-type attacks [54, 70] also connect addresses to data values. However, they often impose additional constraints, such as that the data has to be cached in L1 [70, 77], the physical address has to be known [77], or that an attacker can choose only parts of the target address[54, 74].

Figure 2 illustrates which parts of the virtual and physical address an attacker can choose to target data values to leak. For Meltdown, the virtual address is sufficient to target data in the same address space [46]. Foreshadow already requires knowledge of the physical

Table 1: **Comparison between the RIDL attack [74] and ZombieLoad.**

	RIDL	ZombieLoad
Leakage Source	Fill Buffer, Load Port	Fill Buffer
Leaked Loads	Uncached Loads Only (Fill Buffer)	All Loads (Fill Buffer)
Leaked Stores	All Stores (Fill Buffer)	All Stores (Fill Buffer)
Known Variants	1 or 2[†]	5
Exploited Fault	Page Fault	Microcode Assist, Page Fault
Fixed with Countermeasures	✓	✗
Works on MDS-resistant CPUs	✗	✓ (Variant 2)

[†] The RIDL paper [74] only describes one variant leaking from the fill buffers, but also mentions a variant leaking from the load ports without further description or evaluation.

address and the least-significant 12 bits of the virtual address to target any data in the L1, not limited to the own address space [70, 77]. When leaking the last writes from the store buffer, an attacker is already limited in choosing which value to leak. It is only possible to filter stores based on the least-significant 12 bits of the virtual address, a more targeted leakage is not possible [54].

Zombie loads, which are exploited by ZombieLoad and RIDL [74], provide no control over the leaked address to an attacker. The only possible target selection is the byte index inside the loaded data, which can be seen as an address with up to 6-bit in case an entire cache line is loaded. Hence, we do not count ZombieLoad and RIDL as an attack which leaks data values based on the address. Instead, from the viewpoint of the target control, ZombieLoad and RIDL are more similar to traditional memory-based side-channel attacks. With ZombieLoad and RIDL, an attacker observes the data value of a memory access. Thus, this side channel establishes a connection between the time domain and the data value. Again, the time domain correlates with the instruction pointer of the target address. ZombieLoad and RIDL are the first instances of a class of attacks which connects the *instruction pointer* with the *data value* of a memory access. We refer to such attacks as *data sampling attacks*. Essentially, this new class of data sampling attacks is capable of breaking side-channel resistant applications, such as constant-time cryptographic algorithms [25].

Following the classification scheme from Canella et al. [7], ZombieLoad is a Meltdown-type transient-execution attack, and we propose *Meltdown-MCA* as the canonical name for exploiting microcode assists (MCA, explained further) as exception type. We can further classify the different variants of ZombieLoad (cf. Section 5.1). We propose Meltdown-US-LFB for ZombieLoad Variant 1, as it exploits a page fault on a supervisor page to leak from the fill buffer. For ZombieLoad Variant 2, we propose Meltdown-MCA-TAA (microcode assist caused by transactional asynchronous abort), and for ZombieLoad Variant 3 Meltdown-MCA-AD (micorcode assist caused by modifying the accessed or dirty bit). The RIDL attack exploits non-present page faults caused by NULL-pointer accesses [74]. Thus, we propose the canonical name Meltdown-P-LFB for the RIDL attack.

4 ATTACK SCENARIOS & ATTACKER MODEL

Following most side-channel attacks, we assume the attacker can execute unprivileged native code on the target machine. We assume a trusted operating system if not stated otherwise. This relatively weak attacker model is sufficient to mount ZombieLoad. However, we also show that the increased attacker capabilities offered in certain scenarios, e.g., SGX and hypervisor attacks, may amplify the leakage while remaining within the respective threat model.

At the hardware level, we assume a ubiquitous Intel CPU with simultaneous multithreading (SMT, also known as hyperthreading) enabled. Crucially, we do not rely on existing vulnerabilities, such as Meltdown [46], Foreshadow [70, 77], or Fallout [54]. Hence, even the most recent Intel 9th generation processors with silicon-level Meltdown mitigations remain within our threat model.

User-Space Leakage. In the cross-process user-space scenario, an unprivileged attacker leaks values loaded or stored by another concurrently running user-space application. We consider such a cross-process scenario most dangerous for end users. Many secrets are likely to be found in user-space applications such as browsers.

The attacker is co-located with the victim on the same physical but a different logical CPU core, a common case for hyperthreading.

Kernel Leakage. ZombieLoad can also leak across the privilege boundary between user and kernel space. The values of loads and stores executed in kernel space are leaked to an unprivileged attacker, executing either on the same or a sibling logical core.

An unprivileged attacker performs a system call to the kernel, running on the same logical core. Importantly, we found that kernel load leakage may even survive the switch back from the kernel to user space. Hence, hyperthreading is *not* required for this scenario.

Intel SGX Leakage. ZombieLoad can observe loads and stores executed inside an SGX enclave, even if the loads and stores target the encrypted memory region, *i.e.*, the enclave page cache. The attacker is executing outside of an SGX enclave on a sibling logical core, co-located with the victim enclave on the same physical core. In contrast to the kernel leakage, we did not observe leakage on the *same* logical core after exiting the enclave.

Intel [32] suggests that a remote verifier might reject attestations from a hyperthreading-enabled system "if it deems the risk of potential attacks from the sibling logical processor as not acceptable". Hence, hyperthreading can decidedly be enabled safely on recent Intel Cascade Lake CPUs which include hardware mitigations against Foreshadow [32], but even older SGX machines with up-to-date patched microcode may still run with hyperthreading enabled.

Within the SGX threat model, an attacker can, e.g., modify page table entries [73], or precisely execute the victim enclave at most one instruction at a time [71].

Table 2: **Overview of different variants to induce zombie loads in different scenarios.**

Scenario \ Variant	1	2	3
Unprivileged Attacker	○ ◐	◐ ◐	● ○
Privileged Attacker (root)	● ●	◐ ◐	● ●

Symbols indicate whether a variant can be used in the corresponding attack scenario (●), can be used depending on the hardware configuration as discussed in Section 5.1 (◐), or cannot be used (○).

Virtual Machine Leakage. ZombieLoad can leak loaded and stored values across virtual-machine boundaries. An attacker running inside a virtual machine can leak values from a different virtual machine co-located on the same physical but different logical core.

As the attacker is running inside an untrusted virtual machine, the attacker is not restricted to unprivileged code execution. Thus, the attacker can, e.g., modify guest-page-table entries.

Hypervisor Leakage. An attacker inside a virtual machine can use ZombieLoad to leak values of loads and stores executed by the hypervisor.

As the attacker is running inside an untrusted virtual machine, the attacker is not restricted to unprivileged code execution.

5 BUILDING BLOCKS

In this section, we describe the building blocks for the attack.

5.1 Zombie Loads

The main primitive for mounting ZombieLoad is a load which triggers a microcode assist, resulting in a transient load containing wrong data. We refer to such a load as a *zombie load*. Zombie loads are loads which either architecturally or microarchitecturally fault and thus cannot complete, requiring a re-issue of the load at a later point. We identified multiple different scenarios (cf. Appendix B) to create such zombie loads required for a successful attack. Most variants have in common that they abuse the `clflush` instruction to reliably create the conditions required for leaking from a wrong destination (cf. Section 3.2). In this section, we describe 3 different variants that can be used to leak data (cf. Section 5.2) depending on the adversary's capabilities. While there are more variants (cf. Appendix B and Van Schaik et al. [74] for more known variants), these 3 variants are fast, and each has a unique feature. Table 2 overviews which variants are applicable in which scenarios, depending on the operating system and underlying hardware configuration.

Variant 1: Kernel Mapping. The first variant is a ZombieLoad setup which does not rely on any specific CPU feature. We require a kernel virtual address k, *i.e.*, an address where the user-accessible bit is *not* set in the page-table entry. In practice, the kernel is usually mapped with huge pages (*i.e.*, 2 MB pages). Thus k refers to a 2 MB physical page p. Note that although we use such huge pages for our experiments, it is not strictly required, as the setup also works with 4 kB pages. We also require the user to have read access to the content of the physical page through a different virtual address v.

Figure 3 illustrates such a setup. In this setup, accessing the page p via the user-accessible virtual address v provides an architecturally valid way to access the contents of the page. Accessing the same page via the kernel address k results in a zombie load similar to Meltdown [46] requiring a microcode assist. Note that while there are other ways to construct an inaccessible address k, e.g., by clearing the present bit [70], we were only able to exploit zombie loads originating from kernel mappings.

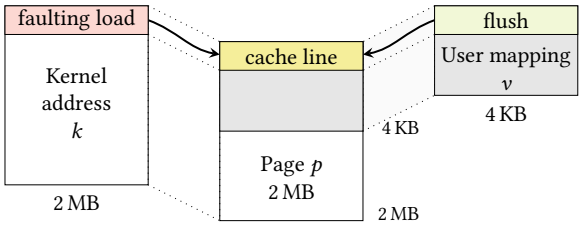

Figure 3: **Variant 1: Using huge kernel pages for ZombieLoad. Page p is mapped using a user-accessible address (v) and a kernel-space huge page (k). Flushing v and then reading from k using Meltdown leaks values from the fill buffer.**

To create precisely the scenario depicted in Figure 3, we allocate a page p in the user space with the virtual address v. Note that p is a regular 4 kB page which is accessible through the virtual address v. We retrieve its physical address through `/proc/pagemap`, or alternatively using a side channel [20, 37, 60]. Using the physical address and the base address of the direct-physical map, we get an inaccessible kernel address k which maps to the allocated page p. If the operating system does not use stronger kernel isolation [19], e.g., KPTI [48], the direct-physical map in the kernel is mapped in the user space and uses huge pages which are marked as not user accessible. A privileged attacker (e.g., for hypervisor or SGX-enclave attacks) can easily create such pages if they do not exist.

The disadvantage of this approach is that it does not work on Meltdown-resistant machines. There, we have to use Variant 2.

Variant 2: Intel TSX. With the second variant of inducing zombie loads, we eliminate the requirement of a kernel mapping. We only require a physical page p which is user accessible via a virtual address v. Any page allocated in user space fulfills this requirement.

Within a TSX transaction, we encode the value of v in a cache covert-channel likewise to Spectre or Meltdown. This ensures that v is in the read set of the transaction [34]. Note that we perform a legitimate load to the user-accessible address v which itself should not cause the TSX transaction to fail. However, by inducing conflicts in the read set (cf. Section 2.3), the TSX transaction "faults" and does not commit. There is no architectural fault but only a transient fault which results in a zombie load.

The main advantage of this approach is that it also works on machines with hardware fixes for Meltdown, which we verified on an i9-9900K and Xeon Gold 5218. However, in contrast to Variant 1, we require the Intel TSX instruction-set extension which is only available in selected CPUs since 2013.

Variant 3: Microcode-Assisted Page-Table Walk. A variant similar to Variant 1 is to trigger a microcode-assisted page-table walk. If a page-table walk requires an update to the access or dirty bit in the page-table entry, it falls back to a microcode assist [11].

In this setup, we require one physical page p which has 2 user-accessible virtual addresses, v and v_2. This can be easily achieved

by using a shared-memory segment or memory-mapped file, which is mapped twice in the application. The virtual address v can be used to access the contents of p architecturally. For v_2, we have to clear the accessed bit in the page-table entry. On Linux, this is not possible in the case of an unprivileged attacker, and can thus only be used in attacks where we assume a privileged attacker (cf. Section 4). However, we experimentally verified that Windows 10 (1803 build 17134.706) periodically clears the accessed bits. We assume that the page-replacement algorithm is responsible for this. Thus, this variant enables the attack on Windows for unprivileged attackers if the CPU does not support Intel TSX.

When accessing the page through the virtual address v_2, the accessed bit of the page-table entry has to be set. This, however, cannot be done by the page-miss handler [11]. Instead, microarchitecturally, the load faults, and a micro-code assist is triggered which repeats the page-table walk and sets the accessed bit [11].

If the access to v_2 is done transiently, *i.e.*, behind a misspeculated branch or after an exception, the accessed bit cannot be set architecturally. Thus, the leakage is not only exploitable once but instead for every access.

5.2 Data Leakage

To leak data with any setup described in Section 5.1, we constantly flush the first cache line of p through the virtual address v. We achieve this by executing the unprivileged `clflush` instruction on the user-accessible virtual address v. For Variant 1, we leverage Meltdown to read from the kernel address k which maps to the cache line flushed before. As with Meltdown-US [46], there are various methods of preventing an architectural exception. We verified that ZombieLoad with Variant 1 works with exception prevention (*i.e.*, speculative execution), handling (*i.e.*, a custom signal handler), and suppression (*i.e.*, Intel TSX).

For Variant 2, the cache-line invalidation of the flush triggers a conflict in the read set of the transaction and aborts the transaction. As there is no architectural exception on a transactional conflict, there is no need to handle exceptions.

For Variant 3, we transiently, *i.e.*, behind a mispredicted branch, read from the address v_2. Similar to Variant 2, there is no architectural exception. Hence, there is no need to handle exceptions.

Counterintuitively, the resulting values leaked for all variants are not coming from page p. Instead, we get access to data which is currently loaded or stored on the current or sibling logical CPU core. Thus, it appears that we reuse fill-buffer entries, and leak the data which the entries references. For Variant 1 and Variant 3, this allowed us to access all bytes from the cache line that the fill-buffer entry references. However, for Variant 2, we are only able to recover the number of bytes of the victim's load or store operation and in contrast to Variant 1, not the entire cache line.

5.3 Data Sampling

Independent of the setup for ZombieLoad, we cannot directly control the address of the data to leak. Both the virtual addresses k and v, as well as the physical address of p is arbitrary and does not correlate with the leaked data. In any case, we simply get the value referenced by one fill-buffer entry which we cannot specify.

Table 3: **Tested environments. A '✓' indicates that the version works, '✗' that it does not work, and '-' that TSX is disabled or not supported on this CPU.**

			Variant		
Setup	**CPU (Stepping)**	**μ-arch.**	**1**	**2**	**3**
Lab	Core i7-3630QM (E1)	Ivy Bridge	✓	-	✓
Lab	Core i7-6700K (R0)	Skylake-S	✓	✓	✓
Lab	Core i5-7300U (H0)	Kaby Lake	✓	✓	✓
Lab	Core i7-7700 (B0)	Kaby Lake	✓	✓	✓
Lab	Core i7-8650U (Y0)	Kaby Lake-R	✓	✓	✓
Lab	Core i7-8565U (W0)	Whiskey Lake	✗	-	✗
Lab	Core i7-8700K (U0)	Coffee Lake-S	✓	✓	✓
Lab	Core i9-9900K (P0)	Coffee Lake-R	✗	✓	✗
Lab	Xeon E5-1630 v4 (R0)	Broadwell-EP	✓	✓	✓
Cloud	Xeon E5-2670 (C2)	Sandy Bridge-EP	✓	-	✓
Cloud	Xeon Gold 5120 (M0)	Skylake-SP	✓	✓	✓
Cloud	Xeon Platinum 8175M (H0)	Skylake-SP	✓	-	✓
Cloud	Xeon Gold 5218 (B1)	Cascade Lake-SP	✗	✓	✗

However, there is at least control within the fill-buffer entry, *i.e.*, we can target specific bytes *within* the 64 B fill-buffer entry. The least-significant 6 bits of the virtual address v refer to the byte within the fill-buffer entry. Hence, we can target a single byte at a specific position from the fill-buffer entry. While at first, this does not sound powerful, it allows leaking sensitive information, such as AES keys, byte-by-byte as shown in Section 6.1.

As described in Section 4, the leakage is not limited to the own process. With ZombieLoad, we observe values from all processes running on the same as well as on the sibling logical CPU core. Furthermore, we also observe leakage across privilege boundaries, *i.e.*, from the kernel, hypervisor, and Intel SGX enclaves. Thus, ZombieLoad allows sampling of all data which is loaded or stored by any application on the current physical CPU core.

5.4 Performance Evaluation

In this section, we evaluate ZombieLoad and the performance of our proof-of-concept implementations[1].

Environment. We evaluated the different variants of ZombieLoad, described in Section 5.1, on different environments listed in Table 3. The tested CPUs range from Sandy Bridge (released 2012) to Cascade Lake (released 2019). While we were able to mount Variant 1 and Variant 3 on different microarchitectures except for Whiskey Lake, Coffee Lake-R, and Cascade Lake-SP, we successfully used Variant 2 on all systems where Intel TSX was available. Thus, Variant 2 also works on microarchitectures with hardware mitigations against Meltdown and Foreshadow.

Performance. To evaluate the performance of each variant, we performed the following experiment on an i7-8650U. While reading a specific value on one logical core, we performed each variant of ZombieLoad on the sibling logical core for 10 s, recording the number of successful and unsuccessful recoveries. For Variant 1 using TSX to suppress the exception, we achieve an average transmission rate of 5.30 kB/s ($\sigma_{\bar{x}} = 0.076$, $n = 1000$) and a true positive rate of 85.74 % ($\sigma_{\bar{x}} = 0.0046$, $n = 1000$). For Variant 2, we achieved

[1] Our proof-of-concept implementations can be found in a GitHub repository: https://github.com/IAIK/ZombieLoad

an average transmission rate of 39.66 kB/s ($\sigma_{\bar{x}} = 0.048$, $n = 1000$) and a true positive rate of 99.99 % ($\sigma_{\bar{x}} = 6.45^{-9}$, $n = 1000$). With Variant 3 in combination with signal handling, we achieved an average transmission rate of 0.08 kB/s ($\sigma_{\bar{x}} = 0.002$, $n = 1000$) and a true positive rate of 52.7 % ($\sigma_{\bar{x}} = 0.0062$, $n = 1000$). Variant 3 in combination with TSX, achieves an average transmission rate of 7.73 kB/s ($\sigma_{\bar{x}} = 0.21$, $n = 1000$) and a true positive rate of 76.28 % ($\sigma_{\bar{x}} = 0.0055$, $n = 1000$).

6 CASE STUDY ATTACKS

In this section, we present 5 attacks using ZombieLoad in real-world scenarios.

6.1 AES-NI Key Leakage

To demonstrate that data sampling is a powerful side channel, we extract an AES-128 key. The victim application uses AES-NI, which is resistant against timing and cache-based side-channel attacks [25].

However, even with the hardware-assisted AES-NI, the key has to be loaded from memory to a 128-bit XMM register. This is usually the case before invoking `AESKEYGENASSIST`, which is used to derive the AES round keys. The round-key derivation is entirely done in hardware using the XMM registers. Hence, there is no memory load required for the derivation of the 11 round keys used in AES-128. Thus, when the key is loaded from memory before the round-key derivation starts is the point where we can mount ZombieLoad to leak the value of the key. For OpenSSL (v3.0.0), this is in the function `aesni_set_encrypt_key` which is called by `EVP_EncryptInit_ex`. Note that instead of leaking the key, we can also leak the round keys loaded in the encryption process. However, to attack the round keys, an attacker needs to leak (and distinguish) more different values, making the attack more complex.

When leaking the key using ZombieLoad, we have first to detect which load corresponds to the key. Moreover, as we can only leak one byte at a time, we also have to combine the leaked bytes to the full AES-128 key correctly.

Side-Channel Synchronization. For the attack, we assume a shared library implementing the AES encryption, e.g., OpenSSL. Even though OpenSSL (v3.0.0) has a side-channel resistant AES-NI implementation, we can rely on classical memory-based side channels to monitor the control flow. With Flush+Reload, we detect when a specific code part is executed [14, 23]. This does not leak any secrets, but it is a synchronization primitive for ZombieLoad.

We constantly monitor a cache line of the code which is executed right before the key is loaded from memory. In OpenSSL (v3.0.0), this is the second cache line of `aesni_set_encrypt_key`, *i.e.*, 64 B after the start of the function. Similarly to Schwarz et al. [61], we leverage the cache state of the cache line as a trigger for the actual attack. Only if we detect a cache hit on the monitored cache line, we start leaking values using ZombieLoad. Hence, we already filter out most bytes not related to the AES key. Note that the synchronization does not have to be perfect, as independent system noise cancels itself out over multiple measurements. Moreover, the key is always 16 B aligned, and we always leak an entire cache line. Hence, there can be no bytewise shift of the AES key – the first 16 B that we leak are always either from the key or from unrelated noise.

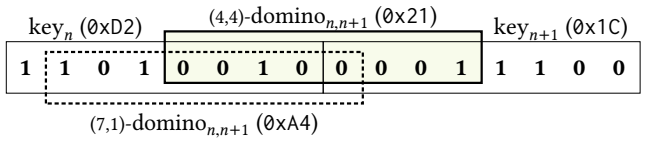

Figure 4: **Additionally leaking domino bytes comprised of bits of different AES-key bytes to filter out unrelated loads.**

Note that if there is no cache line before the load which can be used as a trigger, we can still use a nearby cache line (*i.e.*, a cache line after the load) as a filter. In a parallel thread, we collect the timestamps of cache hits in the nearby cache line. If we also save the timestamps of the values leaked using ZombieLoad, in an offline post-processing step, we can filter out values which were leaked at a different instruction-pointer location.

To further reduce unrelated loads, it is also possible to slow down the victim using performance-degradation techniques such as flushing the code [2, 14]. For OpenSSL, we used performance degradation on the code directly following the load of the key.

Domino Attack. Inevitably, even when synchronizing ZombieLoad by using a cache-based trigger, we also leak values not related to the key. As the bytes in the AES key are independent of each other, we can only assume that the byte which we leak most often per byte position is the correct key byte. Thus, if there is a key byte suffering from noise from unrelated loads, we may assume that the noise is the correct key byte, which leads to a wrong key.

Therefore, we propose the *Domino attack*, an innovative transient error-detection technique for reducing noise when leaking multi-byte loads. In addition to leaking every single key byte, we transmit a specially crafted *domino byte* composed by combining bits from two adjacent key bytes. Note that creating such a domino byte is possible, as the transient domain has access to the full AES key and can use it for arbitrary computations (as also shown with the transient error detection described in Section 6.3). Figure 4 illustrates the idea of the Domino attack. In this case, we leak (4,4) domino bytes consisting of 4 bits of two adjacent key bytes respectively. By combining the lower nibble of one key byte with the higher nibble of the next key byte, we transmit a domino byte which encodes partial information of two key bytes.

In a post-processing step, we consider two adjacent bytes as correct, if we not only leaked both of them often but additionally also the corresponding domino byte. Moreover, we do not look at two key bytes in isolation, but we look at the entire key as a chain of key bytes linked together by domino bytes. If all key bytes and the corresponding domino bytes occurred often in the leaked values, we can assume that the entire key is leaked correctly. Note that the selection of bits can be adapted to the noise measurable before leaking the key, e.g., multiple(7,1) domino bytes can be leaked that are shifted by only a single bit.

Results. We evaluated the attack in a cross-user-space attack (cf. Section 4) using Variant 1. We always ran the attack until the correct key was recovered, *i.e.*, until the key with the highest probability is the correct key. In a practical attack, the number of attacks can even be reduced, as typically it is easy to verify whether a key candidate

is correct. Thus, an attacker can simply test all key candidates with a probability over a certain threshold and does not have to wait until the highest probability corresponds to the correct key.

On average, we recovered the entire AES-128 key of the victim in under 10 s using the cache-based trigger and the Domino attack. During this time, the victim loaded the key approximately 10 000 times.

6.2 SGX Sealing Key Extraction

In this section, we show that privileged SGX attackers can drastically improve ZombieLoad's temporal resolution and bridge from incidental data sampling in the time domain to the targeted reconstruction of arbitrary enclave secrets (cf. Figure 1). We first explain how state-of-the-art enclave execution control and transient post-processing techniques can be leveraged to reliably leak register values at any point during an enclave invocation. Then we demonstrate the impact of this attack by recovering a full 128-bit SGX sealing key, as used by Intel's trusted provision and quoting enclaves to decrypt the long-term EPID private attestation key.

Leaking Enclave Registers. We consider Intel SGX root attackers that co-locate with a victim enclave on the same physical CPU. As a system attacker, we can increase ZombieLoad's temporal resolution by leveraging previous research results exploiting page faults [73, 80] or interrupts [55, 72] to regulate the victim enclave's execution. We use the SGX-Step [71] framework to precisely single-step the victim enclave one instruction at a time, allowing the attacker to reach a code part where sensitive information is stored in CPU registers. At such a point, we switch to unlimited zero-stepping [70] by either setting the system timer interrupt to a very short interval or revoking code page execute permissions before resuming the victim enclave. This technique provides ZombieLoad attackers with a primitive to repeatedly force-reload CPU registers from the interrupted enclave's SSA frame (cf. Section 2.3). Our experiments show that even though the execution of the enclave instruction never completes, any direct operands plus SSA register file contents are loaded from memory each time. Importantly, since the enclave does not make progress, we can perform unlimited ZombieLoad attack attempts to reconstruct CPU register values from these implicit SSA memory accesses.

We further reduce noise from unrelated non-enclave loads on the victim CPU by opting for timer-based zero-stepping with a user-space interrupt handler [72] to avoid repeatedly invoking the operating system. Furthermore, we found that executing the ZombieLoad attack code in a separate address space avoids unnecessarily slowing down the spy through implicit TLB invalidations on enclave entry/exit [29].

Note that the SSA frame spans multiple cache lines. With ZombieLoad, we do not have explicit address-based control over which cache line is being leaked. Hence, leaked data might come from different saved registers that are at the same offset within a cache line. To filter out such noisy observations, we use the Domino transient error detection technique introduced in Section 6.1. Specifically, we implemented a "sliding window" that transmits 7 different domino bytes for each candidate key byte, stuffed with increasing bits from the next adjacent key byte candidate. Any noisy observations that do not match the overlap can now efficiently be filtered out.

Attack on `sgx_get_key`. The Intel SGX design includes a secure key derivation facility through the `egetkey` instruction (cf. Section 2.3). Enclaves execute this instruction to query a 128-bit cryptographic key from the hardware, based on the calling enclave's code layout or developer identity. This is the underlying primitive used by Intel's trusted prebuilt quoting enclave to unseal a long-term private attestation key from persistent storage securely [11, 70].

The official Intel SGX SDK [29] offers a convenient `sgx_get_key` wrapper procedure that first executes `egetkey` with the necessary parameters, and eventually copies the retrieved key into a provided buffer. We reverse engineered the proprietary `intel_fast_memcpy` function and found that in this case, the key is copied using two 128-bit moves to/from the `xmm0` SSE register. We revert to zero-stepping on the last instruction of `memcpy`. At this point, the attacker-induced zero-step enclave resumptions will repeatedly reload a.o., the `xmm0` register containing the 128-bit key from the memory hierarchy.

Results. We evaluated the attack on a Kaby Lake i7-7700 CPU with an up-to-date Foreshadow-patched microcode revision 0x8e and ZombieLoad Variant 1.

In the first experiment, we implemented a benchmark enclave that uses `sgx_get_key` to generate a new report key with different random key IDs. We performed 100 key-recovery experiments on `sgx_get_key` with different random keys. Our results show that 30 % of the times (in 30 experiments) the full 128-bit key is among the key candidates with average remaining key space entropy of 8.8 bits. This entropy is calculated by averaging the entropy of these 30 cases where the full key is among the 128-bit candidates. Among these cases, 3 % of the times the exact full key has been recovered, and the worst-case entropy is about 14 bits. In the other 70 % of the cases where the full key is not among the key candidates, 31 % of the times, we have partial key bytes among the recovered key candidates. The average correct key bytes are 10 out of 16 bytes. In such cases, where some of the key bytes are part of the candidates, most of the failed key bytes reside in the first few bytes of the key. The reason is that the Domino attack has a stronger effect on key bytes in the middle that are surrounded by more key bytes. In the remaining 39 % of the times where the correct key is not among the key candidates, our attack which uses the Domino technique with a sliding window did not reveal any candidates, which means an attacker can simply repeat the attack in such cases.

In the second experiment, we perform an attack on Intel's trusted quoting enclave. The quoting enclave performs a call to `sgx_get_key` to derive the sealing key which is used to decrypt the EPID provisioning blob. We executed the attack on a quoting enclave that is signed with debug keys, so we can use it as ground truth to easily verify that we have recovered the correct sealing key. We executed the attack multiple times on our setup, and we managed to recover the correct 128-bit sealing key after multiple executions of the attack and checking the candidates against each other. The recovered sealing key matches the correct key, and can indeed successfully decrypt the EPID blob for our debug signed quoting enclave. While we did not yet reproduce this attack on the official quoting enclave image signed by Intel, we believe that this experimental evaluation showcased all the required primitives to break Intel SGX's remote attestation guarantees, as demonstrated before by Foreshadow [70].

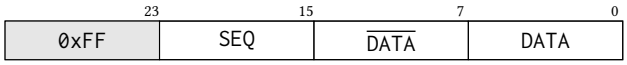

Figure 5: **The packet format used in the covert channel. Every 32-bit packet consists of 8 data bits, 8-bit checksum (two's complement), 8-bit sequence number, and a constant prefix.**

6.3 Cross-VM Covert Channel

To evaluate the performance of ZombieLoad, we implement a covert channel which can be used for all attack scenarios described in Section 4. However, in this section, we focus on the cross-VM covert channel. While covert channels are possible for Intel SGX, the kernel, and the hypervisor, these are somewhat artificial scenarios. Moreover, there are various covert channels available to user-space applications for stealthy inter-process communication [15, 52].

For VMs, however, there are not many known covert channels which can be used between two VMs. So far, all cross-VM covert channels either relied on Prime+Probe [47, 51, 52, 59, 79], DRAMA [58, 63], or bus locking [78]. We show that ZombieLoad can be used as a fast and reliable covert channel between VMs scheduled on the same physical core.

Sender. For the fastest result, the sender repeatedly loads the value to be transmitted from the L1 cache into a register. By not only loading the value from one memory address but instead from multiple memory addresses, the sender ensures that potentially multiple fill-buffer entries are used. In addition, this also thwarts an optimization of Intel CPUs which combines multiple loads from the same cache line to a single load [1].

On a CPU supporting AVX2, the sender can encode up to 256 bits per load (e.g., using the `VMOVAPS` load).

Receiver. The receiver mounts ZombieLoad to leak the values loaded by the sender. However, as the receiver leaks the loads only in the transient domain, the leaked value have to be transferred into the architectural domain. We encode the leaked values into the cache and recover them using Flush+Reload. When encoding values in the cache, we require at least 2 cache lines, *i.e.*, 128 B, per bit to prevent the adjacent-cache-line prefetcher from interfering with the encoding. In practice, we require one physical page per possible value to prevent prefetcher interference. To reduce the bottleneck, we transfer single bytes from the transient to the architectural domain which already requires 256 runs of Flush+Reload.

As a result, our proof-of-concept limits the transmission of data to a single byte per leaked load. However, we can use the remaining bits in the load to ensure that the channel is free of errors.

Transient Error Detection. The transmission of the data between sender and receiver is free of any noise. However, the receiver does not only recover values from the sender, but also other loads from the current and sibling logical core. Hence, to get rid of this noise, we encode the data as shown in Figure 5. This allows the receiver to filter out data not originating from the sender.

Although we cannot transfer the entire packet into the architectural domain, we can compute on the packet in the transient domain. Thus, we run the error detection in the transient domain and only transmit valid packets to the architectural domain.

The challenge to run the error detection in the transient domain is that the number of instructions is limited, and not all instructions can be used. For reliable results, we cannot use instructions which speculate on either control or data flow. Hence, the error-detection code has to be as short as possible and branch free.

Our packet structure allows for extremely efficient error detection. We encode the data in the first byte and the two's complement of the data in the second byte as a checksum. To detect errors, we `XOR` the value of the first byte (*i.e.*, the data) onto the second byte (*i.e.*, the two's complement of the data). If both values are received correctly, the `XOR` ensures that the bits 8 to 15 of the packet are zero. Thus, for a correct packet, the least-significant 16 bits of the packet represent a value between 0 and 255, and for a wrong packet, these bits represent a value which is larger than 255. We use these resulting 16-bit value as an index into our oracle array, *i.e.*, an array consisting of 256 pages. Therefore, any value which is not a correct byte is out of bounds and has thus no effect on the cache state of the array. A correct byte is also a valid index into the oracle array and ensures that the first cache line of the corresponding page is cached. Finally, by applying a cache-based side-channel attack, such as Flush+Reload, we can recover the byte from the cache state of the oracle array [43, 46].

The error detection in the transient domain has the advantage that we do not require computation time in the architectural domain. Instead of waiting for the exception to become architecturally visible by doing nothing, we already use this time to perform the required computation. An additional advantage is that while we are still in the transient domain, we can work on noise-free data. Thus, we do not require complex error correction [52].

Additionally, we also encode a sequence number into the packet. The sequence number allows ordering the received packets and is also recovered using the same method as the data value.

Results. We evaluate the covert channel in a lab environment and a public cloud. In the lab environment, we used 2 VMs running inside QEMU KVM on an i7-8650U. For the cloud scenario[2], we used 2 co-located virtual machines running CentOS 7.6.1810 with a Linux kernel version of 3.10.0-957 on a Xeon E5-2670 CPU.

Both on the cloud, as well as on our lab machine, we achieved an error-free transmission. On our lab machine, we observed transmission rates of up to 26.8 kbit/s with Variant 1. As TSX was not available in the cloud scenario, we achieved a transmission rate of 1.99 kbit/s ($\sigma_{\bar{x}} = 2.5\,\%$, $n = 1000$) with Variant 1 and signal handling.

Table 4 shows a comparison to the transmission rates of state-of-the-art cross-VM covert channels.

6.4 Browsing-Behavior Monitoring

ZombieLoad is also well suited for detecting specific byte sequences within loaded data. We demonstrate an attack for which we leverage ZombieLoad to fingerprint a web browser session. For this attack, we assume an unprivileged attacker running on one logical core and a web browser running on the sibling logical core. In this scenario, it is irrelevant whether the attacker and victim run on a native machine or whether they are in (different) virtual machines.

[2] The cloud provider asked us not to disclose its name at this point.

We present two different attacks, a keyword detection attack which can fingerprint website content, and an URL recovery attack to monitor a victim's browsing behavior.

Keyword Detection. The keyword detection allows an attacker to gain information on the type of content the victim is consuming. For this attack, we constantly sample data using ZombieLoad and match leaked values against a list of keywords defined by the attacker.

We leverage the fact that we have access to a full cache line and can do arbitrary computations in the transient domain (cf. Section 6.3). As a result, we only externalize a small integer indicating which keyword has matched via a cache side channel.

One limitation is the length of the keyword list, as in the transient domain, only a limited number of memory accesses are possible before the transient execution aborts. The most reliable solution is to store the keyword list entirely in CPU registers. Hence, the length of the keyword list is limited by the available registers. Moreover, the length is also limited by the amount of code that is transiently executed to compare leaked values to the keyword list.

URL Recovery. In the second attack, we recover accessed websites from browser sessions without prior selection of interesting keywords. We take a more indirect approach that relies on modern websites performing many individual HTTP requests to the same domain, e.g., to load additional resources such as scripts and images.

In the transient domain, we again sample data using ZombieLoad. While still in the transient domain, we detect the substring "`www.`" inside the leaked data. When we discover a match, we leak the character following "`www.`" to the architectural domain using a cache side channel. This already results in a set of first characters of domain names which we refer to as the candidate set.

In the next iteration, for every domain in the candidate set, we take the last four leaked characters (e.g., "`ww.X`"). We use this string in the transient domain to filter leaked values, similar to the "`www.`" substring in the first iteration. If a match is found, we leak the next character, until the string ends with a top-level domain.

Note that this attack is not limited to URLs. Potentially all data which follows a predictable pattern, such as session cookies or credit-card numbers, can be leaked with this variant.

Results. We evaluated both attacks running an unmodified Firefox browser version 66.0.2 on the same physical core as the attacker.

Covert channel	Speed	Error rate
Pessl et al. [58]	411 kbit/s	4.11 %
Liu et al. [47]	600 kbit/s	1 %
Maurice et al. [52]	362 kbit/s	0 %
ZombieLoad (this)	26.8 kbit/s	0 %
Maurice et al. [51]	751.2 bit/s	5.7 %
Wu et al. [78]	746.8 bit/s	0.09 %
Xu et al. [79]	215 bit/s	5.12 %
Schwarz et al. [63]	11 bit/s	0 %
Ristenpart et al. [59]	0.2 bit/s	-

Table 4: **Transmission rates of state-of-the-art cross-VM covert channels ordered by their transmission speed.**

Table 5: **Number of accesses required to recover a website name. The experiment was repeated 100 times per website.**

Website	Minimal	Average	Maximum
nytimes.com	1	1	3
facebook.com	1	2	4
kernel.org	2	6	13
gnupg.org	2	10	34

```
1 if (x < array_len) {
2     y = array[x];
3 }
```

Listing 1: A simple Spectre-PHT [43] prefetch gadget.

For both attacks, we used ZombieLoad Variant 2. Our proof-of-concept implementation of the keyword-checking attack can check four up to 8-byte long keywords. Due to excessive precomputations of browsers when entering an URL, a keyword is sometimes already matched during the autocompletion of the URL. For highly dynamic websites, such as *nytimes.com*, keywords reliably match on the first access of the website. Accessing mostly static websites, such as *gnupg.org*, have a 60 % probability of matching a keyword in this setup. We observed false positives after the first website access when continuing to use the browser. We hypothesize that memory locations containing the keywords get re-used and may thus leak at a later time again.

For the URL recovery attack, we simulated user behavior by accessing popular websites and refreshing them in a defined time interval. We counted the number of refreshes necessary until we recovered the entire URL, including top-level domain. For each website, the experiment was repeated 100 times.

The actual number of refreshes needed depends on the nature of the website that is visited. If it is a highly dynamic page, such as *facebook.com* or *nytimes.com*, a small number of reloads is sufficient to recover the entire name. For static pages, such as *gnupg.org* or *kernel.org*, the necessary reloads increase by approximately a factor of 10. See Table 5 for a detailed overview of required reloads.

6.5 Targeted Data Leakage

Inherently, ZombieLoad is a 1-dimensional side channel, *i.e.*, the leakage is only controlled by the time. Hence, leakage cannot be steered using specific addresses as is the case, e.g., for Meltdown [46]. While this data sampling is still sufficient for several real-world attacks, it is still a limiting factor for general attacks.

In this section, we show how ZombieLoad can be combined with *prefetch gadgets* [7] for targeted data leakage.

Speculative Data Leakage. Listing 1 illustrates such a gadget, which is a common pattern for accessing an array element [7]. First, the code checks whether the index lies within the bounds of the array. Only if this is the case, the element is accessed, *i.e.*, loaded. While it is evident that for a user-controlled index the corresponding array element can be loaded, such a gadget is more powerful.

On a CPU vulnerable to Spectre, an attacker can mistrain the branch predictor, e.g., by providing several valid values for the array index. Then, by providing an out-of-bounds index, the branch is

misspeculated and speculatively accesses an out-of-bounds value. Alternatively, the attacker can alternate between valid and out-of-bounds indices randomly to achieve a high percentage of mispredictions without any prior branch predictor mistraining.

ZombieLoad cannot only leak architecturally accessed data but also speculatively accessed data. Hence, ZombieLoad can even see the value of loads which are never architecturally visible. Such loads include, among others, speculative memory loads and prefetches. Thus, any Spectre gadget which is not hardened, e.g., using a fence [3, 4, 7, 30] or a mask [7, 8], can be used to leak data.

Moreover, ZombieLoad does not require classic Spectre gadgets containing an indirect array access [43]. A simple out-of-bounds access (cf. Listing 1) is sufficient. While such gadgets have been demonstrated for breaking KASLR [64], they were considered as relatively harmless as they do not leak data [7]. Hence, most approaches for finding gadgets do not consider such gadgets [24, 75]. In the Linux kernel, however, such gadgets are patched if they are discovered, mainly as they can be used together with Foreshadow to leak arbitrary kernel memory [10, 68]. So far, 172 such gadgets were fixed in kernel 5.0 [7]. With ZombieLoad, we show that such gadgets are indeed powerful and require patching.

A huge advantage of ZombieLoad over Meltdown is that it circumvents KPTI. The targeted data is legitimately accessed in the kernel space by the prefetch gadget. Thus, in contrast to Meltdown, stronger kernel isolation [19] does not have any effect on the attack.

Potential Incompleteness of Countermeasures. Mainly, there are 2 methods to prevent exploitation of Spectre-PHT: memory fences after branches [3, 4, 7, 30], or constraining the index to a valid range using a bitmask [7, 8]. The variant using fences is implemented in the Microsoft compiler [42, 43], whereas the variant using bitmasks is implemented in GCC [49] and LLVM [8], and also used in the Linux kernel [49].

Both prevent exploitation of Spectre-PHT as the misspeculation cannot load any data, making it also effective against ZombieLoad.

However, even with these countermeasures in place, there is a remaining leakage which can be exploited using ZombieLoad. When architecturally loading an in-bounds value, ZombieLoad can leak up to 64 bytes of the load. Hence, with ZombieLoad, there is a potential leakage of up to 63 bytes which are out of bounds if the last in-bounds value is at the beginning of a cache line or the base of the array is at the end of a cache line.

Data Leakage. To demonstrate the feasibility of prefetch gadgets for targeted data leakage, we use an artificial prefetch gadget as given in Listing 1. For our evaluation, we used such a gadget in the system-call path of the Linux kernel 5.0.7. We execute ZombieLoad Variant 1 on one logical core and on the other, we execute system calls switching between out-of-bounds and in-bounds array indices to achieve a high frequency of mispredictions in the gadget.

This approach yields leaked values with a large noise component from unrelated loads. We repeat this setup without trying to generate mispredictions to generate a baseline of noise values. We generate frequency distributions for both runs and subtract the noise frequency from the misprediction run. We then choose the byte value that was seen most frequently. leverage We recover kernel memory at one byte per 10 s with 38 % accuracy. Probing bytes for 20 s improves the accuracy to 46 %.

As with Meltdown [46], common byte values such as `0x00` and `0xFF` occur too often and have to be removed from the leaked data for the recovery to work. Our approach is thus blind to these values.

The speed and accuracy can be improved if there is a priori knowledge of the target data. For example, a 7-bit ASCII string can be leaked with a probing time of 10 s per byte with 72 % accuracy.

7 COUNTERMEASURES

As ZombieLoad leaks loaded and stored values across logical cores, a straight-forward mitigation is disabling hyperthreading. Hyperthreading improves performance for certain workloads by 30 % to 40 % [6, 53], and as such disabling it may incur a significant performance impact.

Co-Scheduling. Depending on the workload, a more efficient mitigation is the use of co-scheduling [56]. Co-scheduling can be configured to prevent the execution of code from different protection domains on a hyperthread pair. Current topology-aware co-scheduling algorithms [66] are not concerned with preventing kernel code from running concurrently with user-space code. With such a scheduling strategy, leaks between user processes can be prevented but leaks between kernel and user space cannot. To prevent leakage between kernel and user space, the kernel must additionally ensure that kernel entries on one logical core force the sibling logical core into the kernel as well [33]. This discussion applies in an analogous way to hypervisors and virtual machines.

Flushing Buffers. As ZombieLoad also works across protection boundaries on a single logical core, disabling hyperthreading or co-scheduling are not fully effective as mitigation. Flushing the L1 cache (using `MSR_IA32_FLUSH_CMD`) and issuing as many dummy loads as there are fill-buffer entries is not sufficient. Intel provided a microcode update [33] which added a side effect to the rarely used `VERW` instruction. Operating systems have to issue a dummy `VERW` instruction on every context switch. If the microcode update is installed, this clears the fill buffers and store buffer. Otherwise, the instruction has no side effect. While the microcode update (microcode 0xB4 on i7-8650U), in combination with a correct usage of the `VERW` instruction does reduce the leakage, it does not fully prevent it. We can still observe leakage from kernel values accessed on the same logical core. However, the leakage rate drops from multiple kilobytes per second to less than 0.1 B/s. Our hypothesis is that we can leak data which is evicted from L1 to L2 after issuing the `VERW` instruction. As the `VERW` instruction does not flush dirty L1-cache lines, these can be easily leaked if the attacker partly evicts the L1. Evicting the L1 cache forces the dirty L1-cache lines to go through the fill buffer to L2. Hence, to fully mitigate ZombieLoad, the operating system has to additionally flush the L1 cache. Our performance measurement showed that only flushing the L1 takes on average 1070 cycles (i7-8650U, $n = 1000$, $\sigma_{\bar{x}} = 1.08$). Therefore, we expect that flushing the L1 on every context switch would have a considerable performance impact.

If the microcode update is not available for a specific CPU, Intel provides code sequences to emulate that behaviour [33]. However, these code sequences do not fully work on all CPUs. For example, on the i7-8650U, we still observe leakage which we assume is caused by the replacement policy of the line-fill buffer.

Selective Feature Deactivation. Weaker countermeasures target individual building blocks (cf. Section 5). Intel SGX can be disabled if not required to disable the use of Variant 4 (cf. Appendix B) permanently. The operating system kernel can make sure always to set the accessed and dirty bits in page tables to impair Variant 3. To prevent Variant 2, Intel may offer a microcode update to disable TSX. Such a microcode update already exists for older microarchitectures with a faulty TSX implementation [31]. On the Amazon EC2 cloud, we observed that all TSX transactions always fail, which indicates that such a microcode update might already be deployed there. Unfortunately, Variant 1 is always possible, if the attacker can identify an alias mapping of any accessible user page in the kernel. This is especially true if the attacker is running in or can create a virtual machine. Hence, we also recommend disabling VT-x on systems that do not need to run virtual machines.

Removing Prefetch Gadgets. To prevent targeted data leakage, prefetch gadgets need to be neutralized, e.g., using *array_index_nospec* in the Linux kernel. This function clamps array indices into valid values and prevents arbitrary virtual memory to be prefetched. Placing these functions is currently a manual task and due to the incomplete documentation of how Intel CPUs prefetch data, these mitigations cannot be complete. Note that Spectre mitigations might be incomplete against ZombieLoad (cf. Section 6.5).

Another way to prevent prefetch gadgets from reaching sensitive data is to unmap data from the address space of the prefetch gadget. Exclusive Page-Frame Ownership [40] (XPFO) partially achieves this for the Linux kernel's mapping of physical memory.

Instruction Filtering. For attacks inside of a single process (e.g., JavaScript sandbox), the sandbox implementation must make sure that the requirements for mounting ZombieLoad are not met. One example is to prevent generation and execution of the `clflush` instructions, which so far is a crucial part of the attack.

Secret Sharing. On the software side, we can also rely on secret sharing techniques used to protect against physical side-channel attacks [67]. We can ensure that a secret is never directly loaded from memory but instead only combined in registers before being used. As a consequence, observing the data of a load does not reveal the secret. For a successful attack, an attacker has to leak all shares of the secret. This mitigation is, of course, incomplete if register values are written to and subsequently loaded from memory as part of context switching.

8 CONCLUSION

With ZombieLoad, we showed a novel Meltdown-type attack targeting the processor's fill-buffer logic. ZombieLoad enables an attacker to leak values recently loaded by the current or sibling logical CPU. We show that ZombieLoad allows leaking across user-space processes, CPU protection rings, virtual machines, and SGX enclaves. Furthermore, we show that ZombieLoad even works on MDS- and Meltdown-resistant processors, *i.e.*, even on the newest Cascade Lake microarchitecture. We demonstrated the immense attack potential by monitoring browser behaviour, extracting AES keys, establishing cross-VM covert channels or recovering SGX sealing keys. Finally, we conclude that disabling hyperthreading is necessary to fully mitigate ZombieLoad on current processors.

ACKNOWLEDGMENTS

We thank Werner Haas (Cyberus Technology), Claudio Canella (Graz University of Technology), Jon Masters (Red Hat), Alex Ionescu (CrowdStrike), and Martin Schwarzl (Graz University of Technology). We would like to thank our anonymous reviewers and especially our shepherd, Yinqian Zhang, for their comments and suggestions that helped improving the paper. The research presented in this paper was partially supported by the Research Fund KU Leuven. Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO). Daniel Moghimi is supported by the National Science Foundation, under grant CNS-1814406. The project was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). It was also supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] 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, 1998. US Patent 5,717,882.

[2] Allan, T., Brumley, B. B., Falkner, K., Van de Pol, J., and Yarom, Y. Amplifying side channels through performance degradation. In *ACSAC* (2016).

[3] AMD. Software Techniques for Managing Speculation on AMD Processors, 2018. Revison 7.10.18.

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

[5] Boggs, D. D., and Rodgers, S. D. Microprocessor with novel instruction for signaling event occurrence and for providing event handling information in response thereto, Apr. 1997. US Patent 5,625,788.

[6] Bulpin, J. R., and Pratt, I. A. Multiprogramming performance of the Pentium 4 with Hyper-Threading. In *Second Annual Workshop on Duplicating, Deconstruction and Debunking (WDDD)* (2004).

[7] 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).

[8] Carruth, C. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation), Mar. 2018.

[9] 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).

[10] Corbet, J. Finding Spectre vulnerabilities with smatch, https://lwn.net/Articles/752408/ Apr. 2018.

[11] Costan, V., and Devadas, S. Intel SGX explained. *Cryptology ePrint Archive, Report 2016/086* (2016).

[12] 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).

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

[14] García, C. P., and Brumley, B. B. Constant-time callees with variable-time callers. In *USENIX Security Symposium* (2017).

[15] Ge, Q., Yarom, Y., Cock, D., and Heiser, G. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* (2016).

[16] Glew, A. F., Akkary, H., Colwell, R. P., Hinton, G. J., Papworth, D. B., and Fetterman, M. A. Method and apparatus for implementing a non-blocking translation lookaside buffer, Oct. 1996. US Patent 5,564,111.

[17] Glew, A. F., Akkary, H., and Hinton, G. J. Translation lookaside buffer that is non-blocking in response to a miss for use within a microprocessor capable of processing speculative instructions, 1997. US Patent 5,613,083.

[18] 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).

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

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

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

[22] Gruss, D., Schwarz, M., Wübbeling, M., Guggi, S., Malderle, T., More, S., and Lipp, M. Use-after-freemail: Generalizing the use-after-free problem and applying it to email services. In *AsiaCCS* (2018).

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

[24] Guarnieri, M., Köpf, B., Morales, J. F., Reineke, J., and Sánchez, A. SPECTECTOR: Principled Detection of Speculative Information Flows. *arXiv:1812.08639* (2018).

[25] Gueron, S. Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01, 2012.

[26] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 6 ed. Morgan Kaufmann, 2017.

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

[28] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2016.

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

[30] 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.

[31] Intel. Intel Xeon Processor E3-1200 v3 Product Family Specification Update, https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf Aug. 2018.

[32] Intel. L1 Terminal Fault SA-00161, https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault Aug. 2018.

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

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

[35] Intel. Intel® C++ Compiler 19.0 Developer Guide and Reference, Apr. 2019.

[36] Intel. Side Channel Vulnerability MDS, https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html May 2019.

[37] 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).

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

[39] Johnson, S. P., Savagaonkar, U. R., Scarlata, V. R., McKeen, F. X., and Rozas, C. V. Technique for supporting multiple secure enclaves, June 2012. US Patent 2012/0159184 A1.

[40] Kemerlis, V. P., Polychronakis, M., and Keromytis, A. D. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium* (2014).

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

[42] Kocher, P. Spectre mitigations in Microsoft's C/C++ compiler, 2018.

[43] 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).

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

[45] 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).

[46] 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).

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

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

[49] LWN. Spectre v1 defense in gcc, https://lwn.net/Articles/759423/ July 2018.

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

[51] Maurice, C., Neumann, C., Heen, O., and Francillon, A. C5: Cross-Cores Cache Covert Channel. In *DIMVA* (2015).

[52] 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).

[53] Michael Larabel. Intel Hyper Threading Performance With A Core i7 On Ubuntu 18.04 LTS, https://www.phoronix.com/scan.php?page=article&item=intel-ht-2018&num=4 June 2018.

[54] Minkin, M., Moghimi, D., Lipp, M., Schwarz, M., Van Bulck, J., Genkin, D., Gruss, D., Piessens, F., Sunar, B., and Yarom, Y. Fallout: Reading Kernel Writes From User Space. *arXiv:1905.12701* (2019).

[55] Moghimi, A., Irazoqui, G., and Eisenbarth, T. Cachezoom: How sgx amplifies the power of cache attacks. In *CHES* (2017).

[56] Ousterhout, J. K., et al. Scheduling techniques for concurrent systems. In *ICDCS* (1982).

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

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

[59] 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).

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

[61] Schwarz, M., Gruss, D., Lipp, M., Maurice, C., Schuster, T., Fogh, A., and Mangard, S. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. *AsiaCCS* (2018).

[62] 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).

[63] 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).

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

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

[66] Schönherr, J. H., Juurlink, B., and Richling, J. Topology-aware equipartitioning with coscheduling on multicore systems. In *6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)* (2013).

[67] Shamir, A. How to share a secret. *Communications of the ACM* (1979).

[68] Stecklina, J. [RFC] x86/speculation: add L1 Terminal Fault / Foreshadow demo, https://lkml.org/lkml/2019/1/21/606 Jan. 2019.

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

[70] 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).

[71] Van Bulck, J., Piessens, F., and Strackx, R. SGX-Step: A practical attack framework for precise enclave execution control. In *Workshop on System Software for Trusted Execution* (2017).

[72] Van Bulck, J., Piessens, F., and Strackx, R. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS* (2018).

[73] Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., and Strackx, R. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security Symposium* (2017).

[74] van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. RIDL: Rogue in-flight data load. In *S&P* (2019).

[75] Wang, G., Chattopadhyay, S., Gotovchits, I., Mitra, T., and Roychoudhury, A. oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis. *arXiv:1807.05843* (2018).

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

[77] 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, 2018.

[78] Wu, Z., Xu, Z., and Wang, H. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security Symposium* (2012).

[79] 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).

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

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

A FILL-BUFFER SIZE

In this section, we analyze the size of the fill buffer in terms of fill-buffer entries usable per logical core. Intel describes the fill buffer as a "competitively-shared resource during HT operation" [28]. Hence,

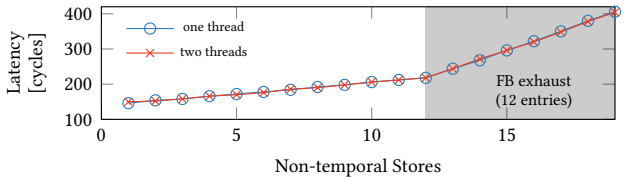

Figure 6: **One logical core can leverage the entire fill buffer (12 entries). If both logical cores execute stores, the fill buffer is competitively shared, leading to an increased latency for both logical cores.**

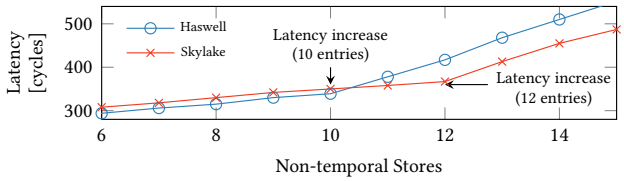

Figure 7: **One pre-Skylake, we measure 10 fill-buffer entries, matching Intel's documentation. On Skylake and newer, we measure 12 fill-buffer entries.**

with 10 fill-buffer entries (Sandy Bridge and newer microarchitectures) [28], we expect that when hyperthreading is enabled, every logical core can use up to 10 entries.

Our experimental setup measures the time it takes to execute n stores to DRAM, for $n = 1, \ldots, 20$. We expect that the time increases linearly with the number of stores n as long as there are unused fill-buffer entries. To ensure that the stores occupy the fill buffer, we leverage non-temporal stores which bypass the cache and directly go to DRAM. We repeated our experiments 1 000 000 times, and we always measured the best case, *i.e.*, the minimum latency, to get rid of any noise.

Figure 6 shows that both logical cores can indeed leverage the entire fill buffer. When running the experiment on one (isolated) logical core, while the other (isolated) logical core does nothing, we get a latency increase when executing more than 12 stores. When we run the experiment on both logical cores in parallel, the latency increase is still after 12 stores.

Interestingly, the documented number of fill buffers does not match our experiments for Skylake and newer microarchitectures. While we measure 10 entries on pre-Skylake CPUs as it is documented, we measure 12 entries on Skylake and newer (cf. Figure 7).

From our experiments we conclude that both logical cores can leverage the entire fill buffer Therefore, every logical core can potentially use any entry in the fill buffer.

B FURTHER VARIANTS

As explained above, we hypothesized that load operations which require a microcode assist might first transiently dereference unauthorized fill buffer entries. Apart from the 3 main variants described in Section 5.1, we experimentally verified multiple approaches to provoke a microcode assist on attacker-controlled load operations.

Variant 4: SGX Abort Page Semantics. SGX-enabled processors trigger a microcode assist whenever an address translation resolves into SGX's "processor reserved memory" area and the CPU is outside enclave mode [11]. Next, the microcode assist replaces the address translation result with the address of the abort page which yields `0xff` for reads and silently ignores writes.

For this attack variant, we require a virtual address v mapping to a physical enclave page p. Whenever accessing v outside the enclave, abort page semantics apply, and a microcode assist will be invoked. While this ensures that the load instruction always reads `0xff` at the architectural level, we found however that unauthorized fill buffer entries accessed by the sibling logical core may still be transiently dereferenced before abort page semantics are applied.

In our experimental setup, much like Variant 2, we access v inside a TSX transaction and encode it in a cache-based covert channel. Interestingly, however, we found that for Variant 4 instead of flushing the first cache line of p, it suffices to simply *access* it before the TSX transaction. We conjecture that this is because abort page values never end up in the cache hierarchy.

Variant 5: Uncachable Memory. A variant closely-related to Variant 4 and CVE-2019-11091, yielding the same effect is to use a memory page that is marked as *uncacheable* instead of an enclave page. As the page miss handler issues a microcode assist when page tables are in uncacheable memory, we can leak data similar to the described SGX scenario where memory can also be marked as write-back [11].