CacheOut: Leaking Data on Intel CPUs via Cache Evictions

Stephan van Schaik*
University of Michigan
stephvs@umich.edu

Marina Minkin
University of Michigan
minkin@umich.edu

Andrew Kwong
University of Michigan
ankwong@umich.edu

Daniel Genkin
University of Michigan
genkin@umich.edu

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

Abstract

Recent speculative execution attacks, such as RIDL, Fallout, and ZombieLoad, demonstrated that attackers can leak information while it transits through various microarchitectural buffers. Named Microarchitectural Data Sampling (MDS) by Intel, these attacks are likened to "drinking from the firehose", as the attacker has little control over what data is observed and from what origin. Unable to prevent these buffers from leaking, Intel issued countermeasures via microcode updates that overwrite the buffers when the CPU changes security domains.

In this work we present CacheOut, a new microarchitectural attack that is capable of bypassing Intel's buffer overwrite countermeasures. We observe that as data is being evicted from the CPU L1 cache, it is often transferred back to the leaky CPU buffers where it can be recovered by the attacker. CacheOut improves over previous MDS attacks by allowing the attacker to choose which data to leak from the CPU's L1 cache, as well as which part of a cache line to leak. We demonstrate that CacheOut can leak information across multiple security boundaries, including those between hyperthreads, processes, and virtual machines, and between user space and the operating system kernel, and from SGX enclaves.

1 Introduction

In 2018 Spectre [29] and Meltdown [31] left an ever lasting impact on the design of modern processors. Speculative and out-of-order execution, which were considered to be harmless and important CPU performance features, were discovered to have severe and dangerous security implications. While the original Meltdown and Spectre works focused on breaking kernel-from-user and process-from-process specifications, many follow-up works have demonstrated the dangers posed by uncontrolled speculation and out-of-order execution. Indeed, these newly-discovered *speculative execution attacks* have been used to violate numerous security domains, such as Intel's Secure Guard Extension (SGX) [46], virtual machine boundaries [48], AES hardware accelerators [44] and others [5, 8, 9, 15, 25, 28, 29, 30, 33, 34]. Recognizing the danger posed by these threats, the computer industry responded with side channel mitigations. For older hardware, Kernel Page Table Isolation (KPTI) [13] as well as Foreshadow [46] and Spectre mitigations [18, 39, 45, 49] were designed and deployed in an attempt to fix leaky hardware isolation features via software means. In parallel, Intel released the Coffee Lake Refresh architecture, which attempted to mitigate Meltdown and Foreshadow in hardware, thereby avoiding the performance overhead induced by software countermeasures.

However, as speculative execution attack research persisted, the security community uncovered a deeper source of leakage: internal and mostly undocumented CPU buffers. With the advent of Microarchitectural Data Sampling (MDS) attacks [7, 42, 47], it was discovered that the contents of these buffers can be dumped via assisting or faulting load instructions, bypassing the CPU's address and permission checks. Using these techniques, an attacker can sample data as it transits through the internal buffers, without the need to match the address of the faulting or assisting load with the address of the data or even its address space. In particular, this allows the attacker to siphon-off data as it appears in the buffer, again breaking nearly all hardware-backed security domains.

Recognizing the danger, Intel deployed countermeasures for blocking data leakage from internal CPU buffer. As modifying the buffer's implementation not to leak information was not possible, Intel instead attempted to mitigate the problem symptomatically by augmenting a legacy x86 instruction, `VERW`, to overwrite the contents of the leaking buffers with constant, data-independent information. This countermeasure was subsequently deployed by all major operating system vendors, performing buffer overwrite on every security domain change. While effective buffer overwriting is often tricky to implement in Intel CPUs [41], the intuition behind the countermeasure is that an attacker cannot recover buffer information that is no longer present. Thus, in this paper we ask

*Work partially done while author was affiliated with Vrije Universiteit Amsterdam.

the following question:

Are buffer overwrites sufficient to block MDS-type attacks? In particular, can an adversary exploit the still leaky CPU buffers in Intel CPUs despite their content being properly overwritten?

Next, we observe that data siphoning via faulting or assisting loads without any address checks by the CPU is a double edged sword. More specifically, while it allows the attacker to obtain *any* information accessed by the victim, it also deprives the attacker of the ability to *select* what information she receives. Previous works [42, 47] had to employ several data analysis and averaging techniques in order to somehow separate useful data from other events triggered by the victim. Thus, in this paper we ask the following secondary question?

Is it possible for the attacker to leak information across arbitrary address spaces, while somehow retaining the capability of selecting which data to leak? If so, how can such attack be implemented in the presence of existing MDS countermeasures?

Finally, while exceptions exist [7], MDS-type attacks [42, 47] are most effective in the HyperThreading scenario, when the attacker and victim reside on the same physical core with the attacker using her logical core to siphon information as the victim accesses it on the sibling core. With HyperThreading disabled on some systems and leaky buffers overwritten, we ask the following final question:

Is HyperThreading essential for mounting MDS type attacks? How can an adversary recover data without the use of HyperThreading in the presence of proper buffer overwrite between security domains?

1.1 Our Contribution

Unfortunately, in this paper we show that ad-hoc buffer overwrite countermeasures are not sufficient to completely mitigate MDS-type attacks. More specifically, we present CacheOut, a transient execution attack that is capable of bypassing Intel's buffer overriding countermeasures as well as allowing the attacker to select which cache sets to read from the CPU's L1 Data cache. Next, as the L1 cache is often not flushed between security domain changes, CacheOut is effective even in the non HyperThreaded scenario, where the victim always runs sequentially to the attacker. Finally, we show that CacheOut is applicable to nearly all hardware-backed security domains, including process to process and process to kernel isolation, virtual machine boundaries, and the confidentiality of SGX enclaves.

CacheOut Overview. We begin by observing that Intel's MDS countermeasures (e.g., the `VERW` instruction) do not address the root cause of MDS. That is, even after Intel's microcode updates, it is still possible to use faulting or assisting load instructions to leak information present in the CPU's line fill, store, and load buffers. Instead, Intel's `VERW` instruction overwrites all the stale information in these buffers, sanitizing their contents.

For CacheOut, we build on the observation [47] that on Intel CPUs, dirty lines from the L1 cache are evicted to the L2 cache via the line fill buffers, where their content remains until overwritten. Thus we can use a faulting or assisting load to recover it. This behavior has two implications. First, using `verw` to flush the CPU buffers on security domain changes does not mitigate CacheOut, because L1 eviction of victim's dirty lines into the fill buffer can occur well after the context switch and the associated `verw` instruction.

Controlling What to Leak. Second, forcing L1 eviction allows us to select the data to leak. Specifically, the attacker can force contention on a specific cache set, causing eviction of victim data from this cache set, and subsequently use the TAA attack [20] to leak this data after it transits through the line fill buffer.

To further control the location of the leaked data, we observe that the line fill buffer seems to have a *read offset* that contols the offset within the buffer that a load instruction reads from. We note that some faulting or assisting loads can use stale offsets from *subsequent* load instructions. Combined with cache evictions, this allows us to control the 12 least significant bits of the address of the data we leak.

Attacking Loads. Cache eviction is useful for leaking data from cache lines *modified* by the victim. This is because the victim's write marks the corresponding cache line as dirty, forcing the CPU to move the data into the line fill buffer. It does not, however, allow us to leak data that is only *read* by the victim, because this data is not written back to memory and does not occupy a buffer when evicted from the L1 cache. We overcome this limitation by evicting the victim's data from the L1 cache before the victim has a chance to read it. This induces an L1 cache miss, which is served via the line fill buffers. Finally, we use an attacker running on the same physical core as the victim to recover the data from the line fill buffers.

Attacking Process Isolation. The leakage of secret data in the manner described thus far has severe implications for OS enforced boundaries. If private data can flow freely across arbitrary address spaces, then the fundamental abstraction of process isolation dissolves completely. We demonstrate the risks concretely by demonstrating an attack for reading private data across processes in different security domains. Targeting an OpenSSL-based victim, we have successfully recovered secret keys and plaintext data in both the hyper-threaded and sequential scenarios.

Attacking the Linux Kernel. Beyond proof-of-concept exploits, we also demonstrate highly practical attacks against the Linux kernel, all mounted from unprivileged user processes. By taking advantage of CacheOut's cache line selection capabilities, we are able to completely de-randomize Kernel Address Space Layout Randomization (KASLR) in under a

second. Furthermore, we demonstrate, to our knowledge, the first attack that extracts stack canaries from the kernel via a micro-architectural side channel.

Attacking Intel's Secure Guard Extensions (SGX). We further demonstrate the effectiveness of CacheOut by dumping the contents of an SGX secure enclave. Even when the victim enclave runs on a fully updated machine that is resistant to all previously known micro-architectural attacks, CacheOut successfully leaks the decrypted contents of the enclave, thereby breaking SGX's confidentiality guarantees.

Attacking Virtual Machines. Another security domain we explore in this paper is the isolation of different virtual machines running on the same physical core. We show that CacheOut is effective at leaking data from both co-resident machines as well as hypervisors. Experimentally evaluating this, we are able to completely de-randomize the Address Space Layout Randomization (ASLR) used by the hypervisor, as well as recover AES keys from another VM.

HyperThreading. While CacheOut is most effective with HyperThreading turned on, it is still possible to use CacheOut to recover information even if the attacker only runs sequentially after the victim, despite Intel's `VERW` mitigation. At a high level, this is because the `VERW` instruction only flushes the internal CPU buffers, but not the L1 cache. Thus, cached data left by the victim can be evicted by the attacker accessing the corresponding eviction set, and subsequently recovered from the leaky line fill buffer.

Summary of Contributions. Our contributions towards advancing the state of the art in speculative execution vulnerabilities are the following:

- We present CacheOut, the first speculative execution attack that can leak across arbitrary address spaces while still retaining fine grained control over what data to leak. Moreover, unlike other MDS-type attacks, CacheOut cannot be mitigated by simply flushing internal CPU buffers between context switches, even when hyperthreading is disabled.
- We demonstrate the effectiveness of CacheOut in violating process isolation by recovering AES keys and plaintexts from an OpenSSL-based victim.
- We demonstrate practical exploits for completely derandomizing Linux's kernel ASLR, and for recovering secret stack canaries from the Linux kernel.
- We demonstrate how CacheOut is effective for violating isolation between two virtual machines running on the same physical core.
- We breach SGX's confidentiality guarantees by reading out the contents of a secure enclave.
- We demonstrate that some of the latest Meltdown-resistant Intel CPUs are still vulnerable, despite all of the most recent patches and mitigations.
- We discuss why current speculative attack mitigations are insufficient, and offer suggestions on what countermeasures would effectively mitigate CacheOut.

1.2 Current Status and Responsible Disclosure

Van Schaik et al. [47] note the relationship between cache evictions and MDS attacks. The first author and researchers from VU Amsterdam notified Intel about the findings contained in this paper during October 2019. Intel acknowledged the issue and has assigned `CVE-2020-0549`, referring to the issue as L1 Data Eviction Sampling (L1DES) with a CVSS score of 6.5 (medium). Intel has also informed that L1DES has been independently reported by researchers from TU Graz KU Leuven.

Current Status. During the course of our research, Intel attempted to mitigate TSX Asynchronous Abort (TAA) [20], a variant of MDS which allows an attacker to leak information from internal CPU buffers. Consequently, Intel recently published microcode updates that enable turning off Transactional Memory Extension (TSX) on its CPUs. These are also effective against CacheOut and we recommend that TSX be turned off across all current Intel platforms. Finally, Intel's security advisory [23] indicates that microcode updates mitigating CacheOut (called L1DES in Intel's terminology) will be published with the public disclosure of our results. We recommend these be installed on affected Intel platforms.

2 Background

2.1 Caches

To bridge the performance gap between the CPU and main memory, processors contain small buffers called *caches*. These exploit locality by storing frequently and recently used data to hide the access latency of main memory. Modern processors typically include multiple caches. In this work we are mainly interested in the L1-D cache, which is a small cache that stores data the program uses. A multi-core processor typically has one L1-D cache in each processor core.

Cache Organization. Caches generally consist of multiple cache sets that can host up to a certain number of cache lines or *ways*. Part of the virtual or physical address of a cache line maps that cache line to its respective cache set, where *congruent* addresses are those that map to the same cache set.

Cache Attacks. An attacker can infer secret information from a victim in a shared physical system such as a virtualized environment by monitoring the victim's cache accesses. Previous work proposed many different techniques to perform cache attacks, the most notable among them being FLUSH+RELOAD and PRIME+PROBE.

FLUSH+RELOAD attacks [14, 50] work with shared memory at the granularity of a cache line. The attacker repeatedly flushes a cache line using a dedicated instruction, such as `clflush`, and then measures how long it takes to reload the cache line. A fast reload time indicates that another process brought the cache line back into the cache.

PRIME+PROBE [24, 27, 32, 35, 40] attacks, on the other hand, work without shared memory, but only at the granularity of a cache set. The attacker repeatedly accesses an *eviction set*—a set of congruent memory addresses that fills up an entire cache set—while measuring how long that takes. As the attacker repeatedly fills up the entire cache set with their own cache lines, the access time is generally low. However, when another process accesses a memory location in the same cache set, the access time becomes higher because the victim's cache line replaces one of the lines in the eviction set.

2.2 Micro-architectural Buffers

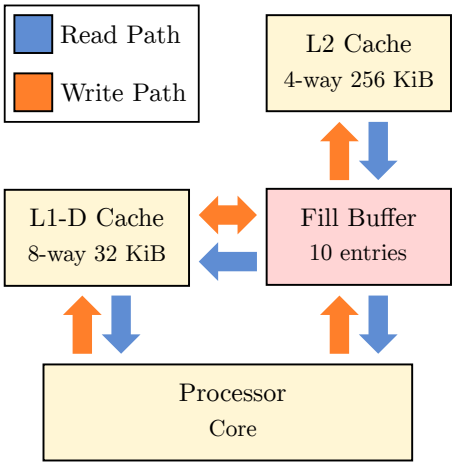

Figure 1: The data paths within the CPU core, marked in blue for loads and marked in orange for stores.

In addition to caches, modern processors contain multiple micro-architectural buffers that are used for storing data in-transit. In this work we are mainly interested in the *Line Fill Buffers*, depicted in Figure 1, which handle data transfer between the L1-D cache, the L2 cache, and the processing core. These buffers serve three main purposes:

Non-Blocking L1-D Cache Misses. One purpose of the line fill buffers is to enable non-blocking operation mode for the L1-D cache [3, 4] by handling the retrieval of data from lower levels of the memory architecture when a cache miss occurs. Specifically, when the processor services a load instruction, it consults both the LFBs and the L1-D cache in parallel. If the data is available in either component, the processor forwards the data to the load instruction. Otherwise, the processor allocates an entry in the LFB to keep track of the address, and issue a request for the data from the L2 cache. When the data arrives, the processor forwards it to all pending loads. The processor may also allocate an entry for the data in the L1-D cache, where it is stored for future use.

Write-Back Buffers. A second purpose of the LFBs is to act as *write-back buffers*. When the program writes to memory, the processor first modifies the copy of the data cached in the L1-D cache. This write is then forwarded down the memory hierarchy, either synchronously, if the memory follows a write-through policy, or asynchronously, if the memory follows a write-back policy. To write the contents to memory, the processor allocates an entry in the line fill buffers, which keeps track of the write operation until it completes.

Non-Temporal Memory Access. When programmers know that caching of data is not required or not desired, they can use *Non-Temporal memory Accesses* to bypass the caching hierarchy. Intel processors use the LFBs for temporarily buffering such data for such memory accesses [37, 38].

Load Squashing and Write Combining. When the program performs multiple memory accesses to the same or to neighbouring addresses, the LFBs can be used to merge these accesses. In the case of *Load Squashing* [1, 2, 6], the processor associates an LFB entry with multiple load instructions. *Write Combining* [12, 26, 36] allows the processor to merge multiple non-temporal writes and reduce the memory traffic.

2.3 Speculative and Out-of-Order Execution

To increase utilization, modern processors diverge from the sequential execution model. Instead, the processor tries to predict future instructions and execute instructions as soon as the data they require is available, rather than following the strict order stipulated by the program. To maintain the illusion of sequential execution, the processor maintains a strict program order on committing the results of instructions to the *architectural state* of the processor, i.e., the state as described by the abstract specifications of the machine language for the processor.

Because the exact sequence of future instructions is not always known in advance, the processor may sometimes execute *transient* instructions that are not part of the nominal program execution. This can occur, for example, when the processor mispredicts the outcome of a branch instruction and executes instructions following the wrong branch. When the processor determines that an instruction is transient, it drops all of the results of the instruction instead of committing them to the architectural state. Consequently, transient instructions do not affect the architectural state of the processor.

2.4 Speculative Execution Attacks

Because transient instructions are not part of the nominal program order, they may sometimes process data that is not accessible in nominal program order. In recent years, multiple *speculative execution attacks* have demonstrated the possibility of leaking such data [8, 9, 15, 28, 29, 30, 34]. In a typical attack, the attacker induces speculative execution of transient instructions that access secret data and leak it back to the attacker. Because the instructions are transient, they cannot transmit the secret data via the architectural state of the processor. However, execution of transient instructions can

```
1  ; %rdi = leak source
2  ; %rsi = FLUSH + RELOAD channel
3  taa_sample:
4      ; Cause TSX to abort
       ↪ asynchronously.
5      clflush (%rdi)
6      clflush (%rsi)
7
8      ; Leak a single byte.
9      xbegin abort
10     movq (%rdi), %rax
11     shl $12, %rax
12     andq $0xff000, %rax
13     movq (%rax, %rsi), %rax
14     xend
15 abort:
16     retq
```

Listing 1: the leak primitive using TSX Asynchronous Abort

modulate the state of micro-architectural components based on the secret data. The attacker then probes the state of the micro-architectural component to determine the secret data.

Most published speculative execution attacks use a FLUSH+RELOAD-based covert channel for sending the data. In a typical attack, the attacker maintains a *probing* array consisting of 256 distinct cache lines. The attacker flushes all of these cache lines from the cache before causing speculative execution of the attack *gadget*. Transient instructions in the attack gadget access a secret data byte, and use it to index a specific cache line in the probing array, bringing the line into the cache. The attacker then performs the reload step of the FLUSH+RELOAD attack to identify which of the probing array's cache lines is in the cache, revealing the secret byte.

2.5 TSX Asynchronous Abort

Some contemporary Intel processors implement memory transactions through the *Transactional Synchronization Extensions* (TSX). As part of the extension, TSX offers the `xbegin` and `xend` instructions to mark the start and the end of a transaction, respectively. The processor guarantees that instructions that form a transaction either all execute to completion or none execute at all. The implementation of transactions executes all of the instructions of the transaction speculatively, and only commits them if execution reaches the `xend` instruction. If during the execution of a transaction the processor encounters an `xabort` instruction, which occurs if any instruction in the transaction faults, the transaction is aborted and all of the instructions in the transaction are dropped.

The Intel manual states that there are CPU implementations where the `clflush` instruction may always cause a transactional abort with TSX. TSX Asynchronous Abort (TAA) [20, 42, 47] exploits this behavior in a speculative execution attack by flushing cache lines before running a transaction that attempts to load data from the flushed cache line. Reading from the flushed line aborts the transaction. However, before the transaction aborts, the processor allocates an LFB entry for the load instruction. When the transaction aborts, the load instruction is allowed to proceed speculatively with the data from the LFB. However, because the load does not complete successfully, the data in the LFB is the contents that remained there from a previous memory access, allowing the attacker to sample data from the LFB [19, 20].

Listing 1 shows a code example of TAA, where the attacker simply allocates a 4 KiB page as the leaking source. She then flushes the cache lines that are about to be used by the TSX transaction, as shown in Lines 4–5. The transaction then attempts to read from the leak page (Line 10), and then transmits the least significant byte of the value it reads using a FLUSH+RELOAD channel as shown in Lines 11–13.

3 CPU Mitigations and Threat Model

Since the discovery of Spectre [29] and Meltdown [31], there have been numerous works that exploit speculative and out-of-order execution to violate hardware-backed security domains [8, 9, 15, 28, 29, 30, 34]. In response, recent Intel processor contain hardware-based countermeasures aimed at addressing these attacks. Table 1 presents the availability of such countermeasures in some recent Intel processors. For processors that are not protected, Intel enabled some features that can be used to provide software-based protection. We now describe these software-based countermeasures.

Kernel Page Table Isolation (KPTI). Meltdown [17, 31] shows that an attacker can bypass the protection of kernel memory. The attack requires the virtual address to be present in the address space. and that the data be present in the L1-D cache. Thus, to mitigate Meltdown, operating systems deploy KPTI [10, 13] or similar defenses that separate the kernel address space from the user address space, thereby rendering kernel addresses inaccessible to attackers.

Flushing the L1-D Cache. KPTI alone soon turned out to be ineffective, as Foreshadow / L1TF [16, 46, 48] demonstrates that any data can be leaked from the L1-D cache by speculatively reading from the physical address corresponding with the data in L1-D cache. Since the disclosure of Foreshadow, Intel CPUs introduced `MSR_IA32_FLUSH_CMD` (MSR 0x10b) to flush the L1-D cache upon a VM context switch. When the MSR is unavailable, the Linux KVM resorts to writing 64 KiB of data to 16 pages.*

Flushing MDS Buffers. Fallout [7], RIDL [47], and ZombieLoad [42] show that attackers can leak data transiting through various internal micro-architectural buffers, such as

*https://github.com/torvalds/linux/blob/aedc0650f9135f3b92b39cbed1a8fe98d8088825/arch/x86/kvm/vmx/vmx.c#L5936

CPU	Year	CPUID	Meltdown	Foreshadow	MDS	TAA	CacheOut
Intel Xeon Silver 4214 (Cascade Lake SP)	Q2 '19	50657	✓	✓	✓	✗	✗
Intel Core i7-8665U (Whiskey Lake)	Q2 '19	806EC	✓	✓	✓	✗	✗
Intel Core i9-9900K (Coffee Lake Refresh - Stepping 13)	Q4 '18	906ED	✓	✓	✓	✗	✗
Intel Core i9-9900K (Coffee Lake Refresh - Stepping 12)	Q4 '18	906EC	✓	✓	✗	✗	✗
Intel Core i7-8700K (Coffee Lake)	Q4 '17	906EA	✗	✗	✗	✗	✗
Intel Core i7-7700K (Kaby Lake)	Q1 '17	906E9	✗	✗	✗	✗	✗
Intel Core i7-7800X (Skylake X)	Q2 '17	50654	✗	✗	✗	✗	✗
Intel Core i7-6700K (Skylake)	Q3 '15	506E3	✗	✗	✗	✗	✗
Intel Core i7-6820HQ (Skylake)	Q3 '15	506E3	✗	✗	✗	✗	✗

Table 1: Countermeasures for speculative execution attacks in Intel processors. ✓ and ✗ indicate the existance or absence of in-silicon countermeasure for the attack.

the line fill buffers discussed in Section 2. To address these issues, Intel provided microcode updates [21] that repurpose the `verw` instruction to flush these micro-architectural buffers by overwriting them. The operating system has to issue the `verw` instruction upon every context switch to effectively flush these micro-architectural buffers.

Threat Model. We assume that the attacker is an unprivileged user, such as a VM, or unprivileged user process on the victim's system. For the victim, we assume an Intel-based system that has been fully patched against Meltdown, Foreshadow, and MDS either in hardware or software. We further assume that there are no software bugs or vulnerabilities in the victim software, or in any support software running on the victim machine. We also assume that TSX RTM is present and enabled. Finally, we assume that the attacker can run on the same processor core as the victim.

4 CacheOut

We now start our exposure of CacheOut and show it can bypass Intel's buffer overwriting countermeasures. At a high level, CacheOut forces contention on the L1-D cache to evict the data it targets from the cache. We describe two variants. First, in the case that the cache contains data modified by the victim, the contents of the cache line transits through the LFBs while being written to memory. Second, when the attacker wishes to leak data that the victim does not modify, the attacker first evicts the data from the cache, and then obtains it when it transits through the line fill buffers to satisfy a concurrent victim read. Figure 2 shows a schematic overview of these attacks, which we now describe.

Attacking Reads. The left part of Figure 2 shows our attacks on victim read operations. We assume that the attacker has already constructed an eviction set for the cache set that contains the victim's data. We further assume that the attacker and the victim run on two hyperthreads of the same processor core. For the attack, the attacker reads from all of the addresses in the eviction set (Step 1). This loads the eviction set into the L1-D, evicting the victim's data. Next, the attacker waits for the victim to access his data (Step 2). This victim access

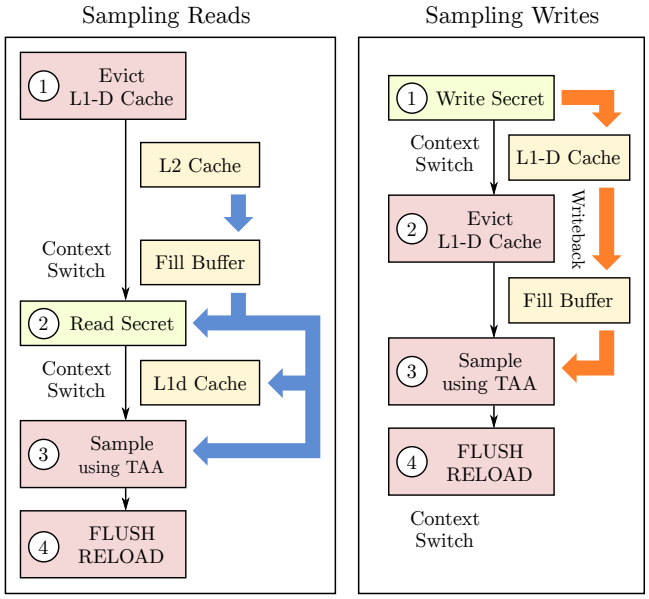

Figure 2: A schematical overview of how we use TSX Asynchronous Abort to leak from loads and stores through *Fill Buffers* on the left and the right respectively. Victim activity, attacker activity and micro-architectural effects are shown in green, red and yellow respectively.

brings the victim's data from the L2 cache into the line fill buffers, and subsequently to the L1 cache. Finally, the attacker uses TAA (Step 3) to sample values from the line fill buffer and transmit them via a FLUSH+RELOAD channel (Step 4).

Attacking Writes. Figure 2 (right) depicts our attack on victim write operations. The attacker first waits until after the victim performs a write operation to a cache line (Step 1). The attacker then accesses the corresponding eviction set, forcing the newly written data out of the L1-D cache and down the memory hierarchy (Step 2). On its way to memory, the victim's data passes through the LFBs. Thus, in Step 3, the attacker samples uses TAA to sample the buffer and subsequently uses FLUSH+RELOAD to recover the value (Step 4).

Having outlined the general structure of CacheOut, we now proceed to describe it in further details.

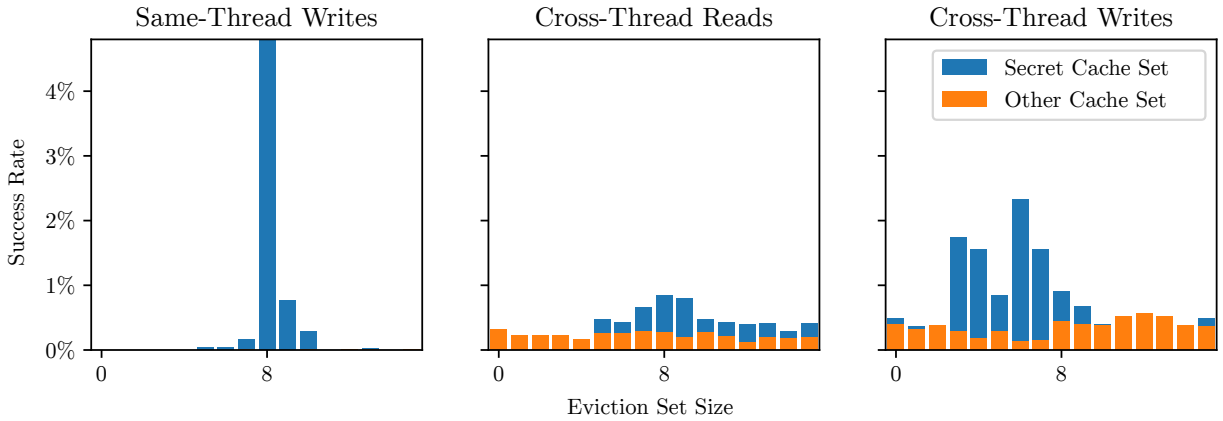

Figure 3: Number of loads/stores required to evict the secret cache line from the victim. The blue bars indicate how often we observe the secret signal from the right cache line, while the orange bars indicate how often we observe it from the strongest wrong cache line. We ran 10,000 iterations per tested value.

4.1 L1-D Eviction

Eviction Set Construction. A precondition for CacheOut is that the attacker is able to construct an eviction set for L1-D cache sets. Recall that an eviction set is a collection of congruent addresses that all map to the same cache set. On contemporary Intel processors, virtual addresses are used for addressing the L1-D cache. Specifically, bits 6–11 of the virtual address are used to identify the cache set used. Consequently, by allocating eight 4 KiB memory pages, the attacker can cover the whole cache, and eviction sets can be constructed from memory addresses that have the same offsets in each of these pages.

Measuring L1-D Eviction. To measure the number of accesses we need to make in order to evict the victim's line from the cache, we use a synthetic victim that repeatedly accesses the same cache set. We test CacheOut with varying sized for the eviction set, and under three different attack scenarios. Figure 3 shows the result of the three scenarios. In the first scenario (left) the victim and the attacker time-share the same hyperthread. As expected, when the eviction set contains eight addresses we get the best results, recovering the contents of the victim's cache line in 4.8% of the cases.

The other two scenarios test cross-hyperthread scenarios, first targeting the victim's memory reads (middle) and then victim writes (right). As we can see, the results here are not as strong, but the likelihood of getting the victim's data is still higher than getting data from cache lines other than the targeted. For victim reads, we still get the best results with an eviction set of size eight. For victim writes, best results are obtained with an eviction set of six addresses. We suspect that the cause is the increased contention on the L1-D due to having two active hyperthreads.

Measuring Data Selection. To demonstrate that we can select the cache set we want to monitor, we repeat the experiments, this time varying the "secret" cache set the victim uses and the cache set the attacker evicts. the results, summarized in Figure 4, show that in all scenarios the attacker can target the victim's cache set, albeit with some noise for the case of cross-thread victim writes.

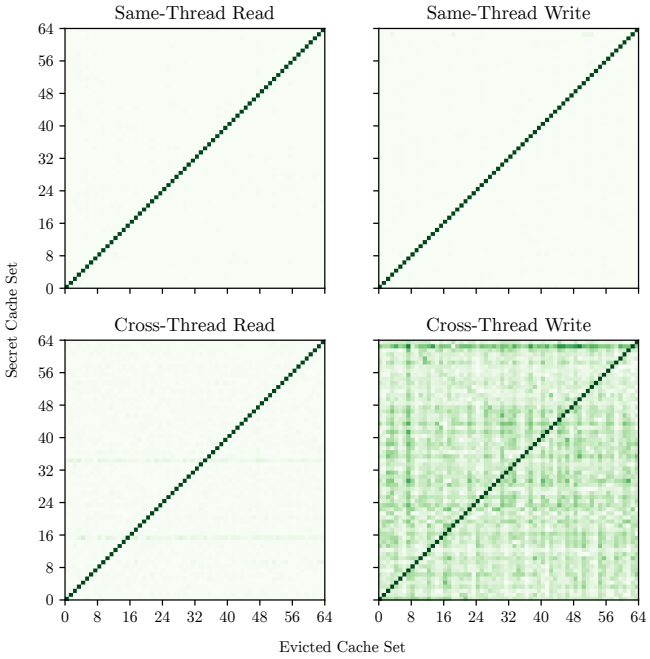

Figure 4: The victim loads/stores a secret to every possible cache line (y-axis), while the attacker tries to evict every possible cache line (x-axis) to leak it. We ran 10,000 iterations per tested value.

4.2 Selecting Cache Line Offsets

So far we have shown how to control the cache set that CacheOut leaks data from. However, we note that like previous TAA attacks [47], we still do not have control of the offset within the cache line we read from. In fact, our experiments showed that the attack retrieves data from arbitrary positions in the

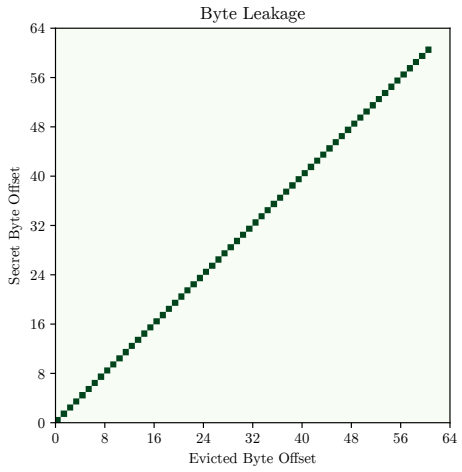

Figure 5: The victim loads/stores a secret byte to every possible offset within a fixed cache line (y-axis), while the attacker tries to leak from every possible byte offset (x-axis) to leak it.

cache line we target. However, upon further experimentation we determined that if the offset of load instructions that *follow* the TAA attack also controls the cache line offset from which the TAA attack reads from.

Listing 2 shows our amended leakage primitive that allows us to control the offset we read data from. As we can see, the code is basically the same as the TAA leak primitive, but we added four `movq` instructions in Lines 16–19.

Analysing the CacheOut Primitive. We note that the leakage in the CacheOut primitive occurs in Line 11. At a first glance it seems odd that later `movq` instruction can affect the outcome of this instruction. However, we note that the `movq` instructions we add do not depend on the outcome of the leaking `movq` at Line 11. Consequently, due to out-of-order execution they can execute *before* instructions that precede them in program order. We hypothesize that the line fill buffer has a *read offset*, some internal state that determines the offset within buffer entries to read data from. This read offset gets reused by the leaking `movq` when the transaction aborts, thereby allowing us to select the desired cache line offset.

Evaluating Offset Selection. To evaluate our offset selection method, we performed an experiment where the victim chooses a byte offset and writes some "secret" value to this byte, setting the rest of the bytes in the same cache line to zero. The attacker then tries to leak the secret from every possible byte offset from the victim's cache line. As we can see in Figure 5, we can successfully select the offset in the cache line we leak from. Furthermore, combining this behavior with the L1-D cache set eviction that we previously described, we have full control over the 12 least significant bits of the address we leak from, allowing the attacker to choose exactly which page offset to read.

```
1  ; %rdi = leak source
2  ; %rsi = FLUSH + RELOAD channel
3  ; %rcx = offset-control address
4  taa_sample:
5      ; Cause TSX to abort
       ↪ asynchronously.
6      clflush (%rdi)
7      clflush (%rsi)
8
9      ; Leak a single byte.
10     xbegin abort
11     movq (%rdi), %rax
12     shl $12, %rax
13     andq $0xff000, %rax
14     movq (%rax, %rsi), %rax
15
16     movq (%rcx), %rax
17     movq (%rcx), %rax
18     movq (%rcx), %rax
19     movq (%rcx), %rax
20     xend
21 abort:
22     retq
```

Listing 2: CacheOut leak primitive.

4.3 Determining the Leakage Source

To determine the source of our leakage, we study the effects of flushing internal micro-architectural structures as well as the L1-D Cache.

Flushing the MDS Buffers. To see the effects of flushing the micro-architectural buffers, we issue the `verw` instruction after evicting from the cache but before executing the leakage primitive. When the victim and attacker execute on the same hyperthread, this completely remove the signal. Thus, we conclude that the actual leakage indeed happens from one of the MDS buffers. Next, when we move the `verw` instruction before evicting from the cache, the attack leaks data from cache lines modified by the victim, but does not leak victim reads. This supports the hypothesis that the L1-D cache eviction transfers the data into the leaky LFB when it is written back to the L2 cache.

Finally, we note that issuing the `verw` instruction before evicting from the cache significantly improves the signal for victim writes, both in the cross-hyperthread and in the same-hyperthread scenarios. We conjecture that this `verw` removes all values but the leaked one from the LFB, thereby increasing the probability of the leaked value be successfully recovered by TAA.

Flushing the L1-D Cache. To confirm that it is the eviction from the L1-D that causes modified data to transit through the LFB, we try to flush the L1-D using `MSR_IA32_FLUSH_CMD` (MSR 0x10b) between the victim access and the cache

eviction. We find that in the same-hyperthread case, this completely removes the signal. This again supports the hypothesis that evictions of modified data from the L1-D transit through the LFB from where CacheOut is able to leak the data.

5 Cross Process Attacks

To demonstrate the practical risk that CacheOut poses even in the case where HyperThreading is disabled, we developed a proof of concept attack wherein an unprivileged user process leaks confidential data from another process. Moreover, in our example we demonstrate the feasibility of address selection, which allows the attacker to choose what locations she wants to read in the victim's address space. This capability allows us to lift the known-prefix or known-suffix restriction of [47], where in order to leak the data that attack must know some prefix or suffix of it.

Experimental Setup. We run the attack presented in this section on two machines. The first is equipped with an Intel i7-8665 CPU (Whiskey Lake), running Linux Ubuntu 18.04.3 LTS with a 5.0.0-37 generic kernel. Our second machine is equipped with an Intel Core i9-9900K (Coffee Lake Refresh, Stepping 13) running Linux Ubuntu 18.04.1 LTS with a 5.3.0-26 generic kernel. The former machines uses microcode version `0xca`, while the latter uses `0xb8`.

Leaking from OpenSSL AES. Our cross-process attack aims to leak the AES key as well as the plaintext message that results from the victim decrypting AES. To that aim, we constructed a victim process that repeatedly decrypts an encrypted message, followed by issuing the `sched_yield()` system call. The attacking process runs sequentially on the same virtual core, and repeatedly calls `sched_yield()` to allow the victim to run and decrypt the ciphertext. After the victim finishes running, the attacker evicts the L1-D cache set of interest with the goal of leaking interesting information from the victim process into the line fill buffer. Then the attacker uses TAA to sample the AES key as well as the decrypted message from the line fill buffer. Figure 6 illustrates this process. Finally, even though we artificially instrumented the victim process to yield the CPU to simplify the synchronization problem, [14] demonstrate that this is not a fundamental limitation and can be overcome with an attack on the Linux scheduler.

Experimental Results. Compared to previous techniques [42, 47], our attack benefits from CacheOut's cache line selection capabilities. By targeting specific cache lines to leak from, our attack classifies plaintext bytes with 96.8% accuracy and AES keys with 90.0% accuracy, taking 15 seconds on average to recover a single 64 bytes cache line.

6 Attacks on Linux Kernel

In addition to the cross process attack described in Section 5, we also demonstrate how CacheOut can leak sensitive data from the unmodified Linux kernel. More specifically, we demonstrate attacks for breaking kernel address space layout randomization (KASLR) and recovering secret kernel stack canaries.

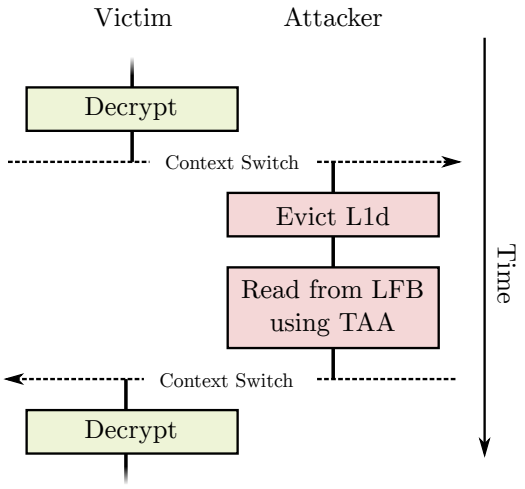

Figure 6: After decrypting, the victim writes the plaintext, bringing it into the L1-D cache. The attacker can then evict it from the L1-D cache and use TAA to read it from the LFB.

6.1 Derandomizing Kernel ASLR

ASLR Overview. Address Space Layout Randomization (ASLR) is a defense in depth countermeasure to common binary exploits. In the event that an attacker redirects the victim's control flow (using return oriented programing (ROP [43]) , overwriting a function pointer, etc), mounting a code reuse attack still requires the attacker to know the locations of targeted code pieces. To avoid having a predictable code layout in the victim's memory, ASLR attempts to limit the attacker's knowledge of the program's code section by offsetting the entire code section by a randomized value, known as the ASLR slide. However, since the entire code section shares the same ASLR slide, uncovering a single virtual address from the victim program (such as a function pointer) suffices to completely derandomize the code section's location. ASLR is commonly deployed in both user space applications and the kernel, where is it know as Kernel ASLR (KASLR).

Attacking Kernel ASLR. We now show how the cache line selection capabilities of CacheOut enable an attacker to reliably leak a kernel function pointer and breach KASLR in under a second. In our attack, the attacking process binds itself to a single core and repeatedly executes a loop composed of just a `sched_yield()` followed by TAA. When `sched_yield()` returns to the attacking process from the kernel, the attacker uses TAA to leak stale data in the L1-D cache leftover from the kernel during the context switch. We first used TAA to leak data from all 64 cache lines at all byte offsets. Upon inspection, we found that a pointer corresponding to

the `hrtick` kernel symbol could be consistently recovered from the same cache line at the same byte offset. We then verified that this location remains static across both reboots and different machines running the same kernel version.

Attack Evaluation. An attacker can exploit this by first conducting offline analysis by running the attack code on a machine running the same kernel version as the victim's. Then, after learning the location, the attacker can conduct the online attack against the victim; the difference is that the attacker needs only leak the single cache line and eight byte offsets that contain the kernel pointer, as opposed to an entire 4KiB of data. Thus, the cache-line selection capabilities of CacheOut result in a running time of 14 seconds for the offline analysis phase, and under a single second for the online attack phase.

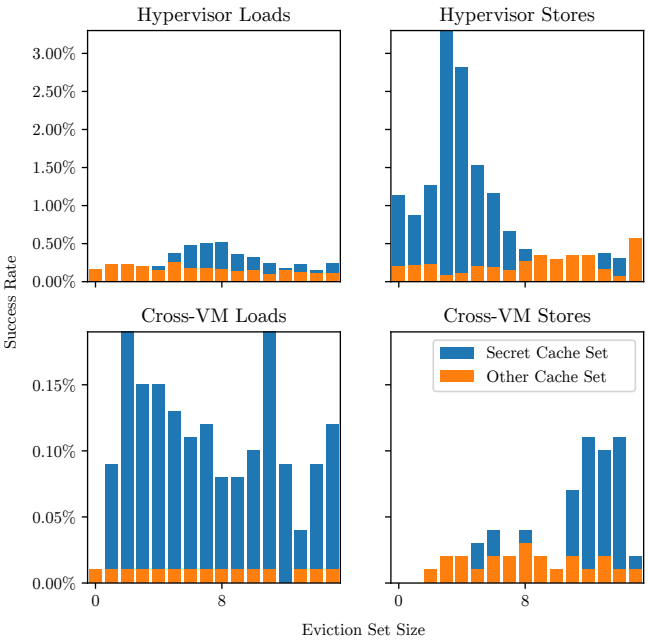

Figure 7: The number of loads/stores required to evict the L1-D cache sets for loads (left), stores (right), against the hypervisor (top) and across VMs (bottom).

6.2 Defeating Kernel Stack Canaries

Stack Canaries Overview. Stack Canaries [11] are another widely deployed defense-in-depth countermeasure to binary exploits. More specifically, stack canaries are designed to protect against stack-based buffer overflow attacks, where an attacker writes beyond the end of a buffer on the stack and overwrites data used for control flow (e.g. function pointers and return addresses). When the victim subsequently attempts to use this information to determine its control flow, the attacker is able to redirect the victim to execute arbitrary code inside the victim process's address space.

Stack-based buffer overflow attacks are particularly concerning when the victim is the kernel, as a successful attack against the OS kernel will result in the attacker gaining root over the entire system. As a mitigation, modern compilers employ the use of stack canaries. These 64-bit randomized values are stored at the bottom of each stack frame at the beginning of each function. Before unwinding the stack function and returning, the canary value is checked for modification. If the value has changed, the program exits gracefully. Since the stack canary is located between the local variables and the return address, any attempt to overwrite a function's return address via a buffer overflow necessarily overwrites the stack canary. This causes the victim program to fail quickly and crash before the attacker can gain arbitrary code execution, thereby limiting the attacker to a Denial-of-Service attack.

Extracting Kernel Canary Values. We used CacheOut to leak the Linux kernel's stack canary value, which is shared for all kernel functions running on the same core in the context of the same user process. The attacking code is similar to the KASLR break, but instead of repeatedly calling `sched_yield()`, the attacker repeats a loop with a write to /dev/null, followed by performing TAA to leak from the L1-D cache. We found three different locations (cache line and byte offset) where the kernel's stack canary can be leaked. On average, the attack succeeds in 23 seconds when evaluated on an i9-9900K stepping 12 CPU with microcode version 0xca running Ubuntu 16.04.

To our knowledge, CacheOut is the first micro-architectural side-channel that manages to recover stack canaries from the kernel. This is made possible by the address selection capabilities, as a completely random 64-bit value is extremely difficult to detect without targeting a particular cache line.

7 Breaking Virtualization

Infrastructure-as-a-Service (IaaS) cloud-computer services provide their end-users virtualized system resources, where each tenant runs in a separate virtual machine (VM). Modern processors support virtualization by means of extensions where the hypervisor can create and manage these VMs that each run their own operating system and applications in an isolated environment, analogous to how an operating system creates and manages processes. In this section, we demonstrate that CacheOut can break this isolation, showing how to leak both from the hypervisor as well as VMs that are co-resident on the same physical CPU core.

Experimental Setup. We run the attacks presented in this section on an Intel Core i7-8665U (Whiskey Lake) running Linux Ubuntu 18.04.3 LTS with a 5.0.0-37 generic kernel and microcode update `0xca`.

Evicting L1-D Cache Sets. We perform the same experiment as in Section 4.1 to determine the number of loads and stores necessary to evict any L1-D cache set across VMs and

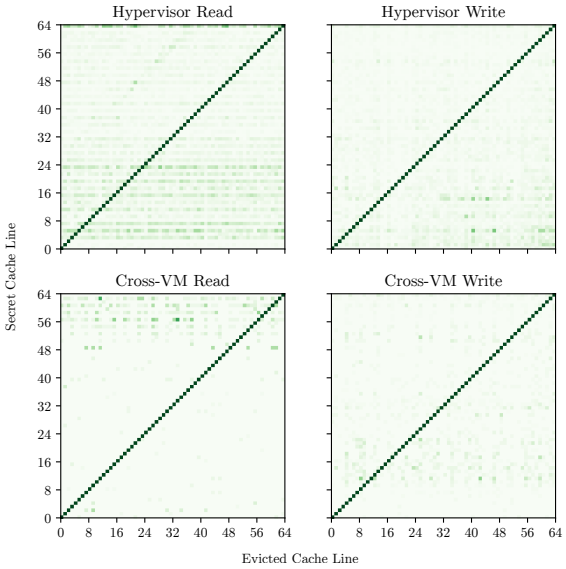

Figure 8: The hypervisor and victim VM loads/stores a secret from/to every possible cache line (y-axis), while the attacker VM tries to evict every possible cache line (x-axis) to leak it.

to attack the hypervisor across CPU threads. Figure 7 shows that we can successfully leak data from the hypervisor as well as across VMs using 8 loads and 3 to 4 stores against the hypervisor and 10 loads and stores across VMs.

Selecting L1-D Cache Sets. After establishing the ideal number of loads and stores required to evict the L1-D cache set, we now proceed with the second experiment as outlined in Section 4.1. We set up our hypervisor and victim VM to write a secret to every possible cache set, and then try to leak from every possible cache set using our attacker VM. We present our results in Figure 8, which clearly shows that we are able to select and evict any L1-D cache set and leak secrets from either the hypervisor or a co-resident victim VM.

Leaking AES keys across VMs. We run a setup with an attacker and a victim VM across Intel Hyper-Threads, where the victim is running a program that repeatedly performs AES decryptions using OpenSSL 1.1.1. The attacker VM evicts the L1-D cache set of interest in an attempt to leak interesting information from the victim process through the line fill buffer. Once the attacker manages to evict the data from the L1-D cache into the line fill buffer, the attacker uses TAA to sample the AES key.

Experimental Results. We target a specific cache line to leak from, and run our experiment three times. For each run, we attempt to leak each key byte 10000 times. During all three runs, the bytes corresponding to the victim's AES key were observed during 20 out of the 10000 attempts. In order to improve our signal, we run `sched_yield()` in a loop in an attempt to capture baseline noise that we can later subtract from the AES signal. After subtraction, we were able to recover 75% of the key bits on average across the three runs. Finally, as in the case of Section 5, it takes about 15 seconds on average to leak a single 64-byte cache line.

Breaking Hypervisor ASLR. Similar to kernels, hypervisors deploy Address Space Layout Randomization (ASLR) as a countermeasure against attackers trying to exploit buffer overflows as explained in Section 6.1. To be able to leak any information regarding ASLR from the hypervisor, we first have to find a controlled way to trap into the hypervisor. One way of trapping into the hypervisor is by issuing `cpuid` from our program, as the hypervisor hides or represents CPU information in a different way. We assume an attacker VM with full access over at least a single CPU core with Intel Hyper-Threading. On one of the threads, the attacker runs a loop issuing `cpuid`, while on the other thread it runs the attacker program.

Disambiguating Guest and Host. In addition to the hypervisor, our attacker VM is also running its own kernel that we leak kernel pointers from. In order to disambiguate any of the kernel pointers we find from actual hypervisor pointers, we simply reboot our VM. Rebooting our VM ensures that the guest kernel has to choose random values again to use for KASLR, while the hypervisor keeps using the same random value. This way we can tell apart the pointers we leak from our hypervisor, as the kernel pointers are likely to change after a reboot.

Attack Evaluation. We first perform an offline phase to determine whether there are static locations that we can leak hypervisor addresses from. We found that there are indeed various locations that leak a hypervisor pointer to `x86_vm_ops`. After establishing the fixed locations for a known kernel, we can mount an online attack on the hypervisor. This reduces the time from roughly 17 minutes in the offline phase to 1.8 seconds.

8 Breaching SGX Enclaves

Intel's Software Guard Extensions (SGX) is a set of CPU features that offer hardware backed confidentiality and integrity to user space programs, even in the presence of a root-level adversary. This enables users to execute a program securely even on a system where the OS and all of the hardware, except for the CPU, are untrusted. CacheOut's ability to leak across arbitrary address spaces while retaining address selection capability enable it to breach SGX's confidentiality guarantees. To demonstrate this, in this section we present an attack that dumps the contents of an SGX enclave. In agreement with the SGX threat model, we assume a malicious kernel that aims to breach the victim enclave's confidentiality. We also assume that hyperthreading is enabled and that the kernel runs in parallel on the same physical core as the victim enclave. We show an overview of the attack in Figure 9.

Loading Secret Data into the Cache. Even though the malicious kernel cannot directly read the contents of the en-

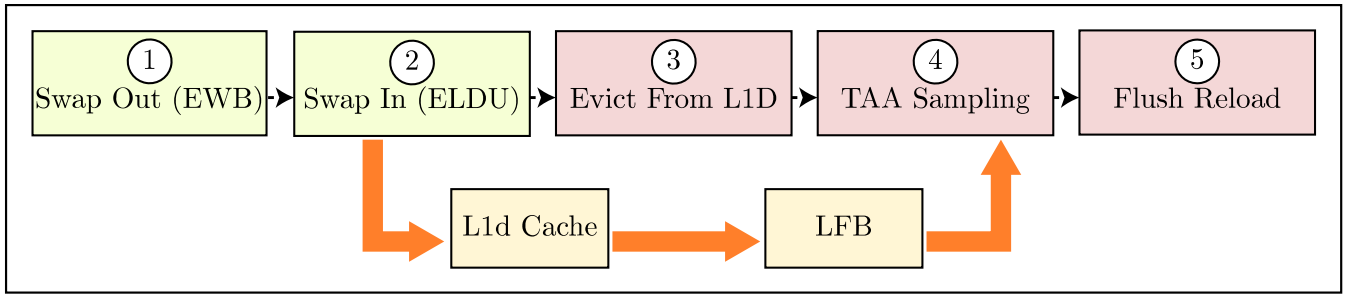

Figure 9: A schematical overview of how the SGX paging mechanism, in combination with TSX Asynchronous Abort, leaks arbitrary SGX data.

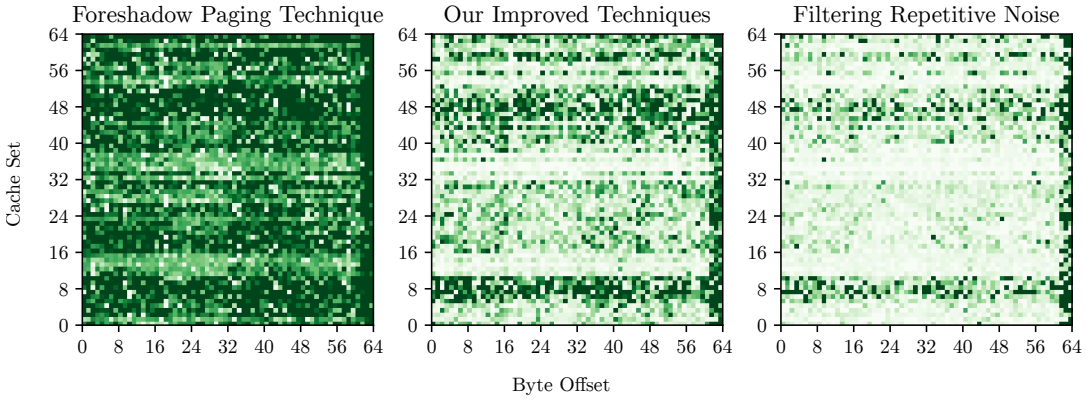

Figure 10: Accuracy heatmap of the attack against SGX, where a darker color indicates a higher ranking in the classification and all rankings after 20 are the same color. The left figure displays the accuracy obtained from using the technique from [46], the middle figure from using our improved paging technique, and the right figure from our most advanced technique for filtering repetitive noise.

clave, the kernel is still responsible for paging the victim enclave's pages using the special SGX instructions `ewb` and `eldu`. Van Bulck et al. [46] discovered that using these instructions, an attacker can load the data into the L1-D cache, even in case the victim enclave is not running at all. Similarly to Van Bulck et al. [46], we used the `ewb` and `eldu` instructions to load the victim's decrypted page into the L1-D cache. See Steps 1 and 2 in Figure 9. Next, since the `ewb` and `eldu` instructions operate at page granularity, an attacker using these instructions can choose which pages to load into the L1 cache. This gives the attacker even more control over the leaked data, compared to the other attacks in this paper, which only have control over the page offset.

Reading Secret Enclave Data. After loading the secret data into the L1-D (Figure 9 Steps 1-2), the attacker can mount a CacheOut attack from a parallel hyperthread that will perform Steps 3-5. When the attacker evicts the targeted cache line in Step 3, it is evicted into a leaky microarchitectural buffer. The attacker then leaks the data in the chosen eviction set via TAA (Step 4) and retrieves it using FLUSH+RELOAD in Step 5. Finally, the entire process is repeated several times in order to improve accuracy.

Improving Leakage Probability Figure 10 displays heatmaps depicting the classification ranking of each byte as a function of cache line and byte offset, where the lighter colors indicate bytes that are classified more accurately. Amongst other sources, a substantial amount of noise stems from background system noise, which eclipses the signal from `eldu`. We discovered that having multiple copies of the plain-text in the cache improves the accuracy of the attack. To achieve this, each time the attacker executes `ewb` and `eldu`, she allocates a different physical frame for the SGX enclave. Since writing to different physical addresses puts the data in different cache ways, we were able to fill the entire cache with the victim enclave's secret page, thereby improving the probability of evicting the correct data. We can see the improved results in Figure 10 (middle).

Noise Reduction. Even after employing our improved technique, however, substantial noise remains. Since the noise is repetitive in nature, we cleaned the noise further by first conducting the attack with a separate enclave's page set to all zeros to determine the baseline noise level. Then, we subtracted the corresponding baseline noise scores from each of the byte values obtained for Figure 10 (middle), and obtained the further improved results shown in Figure 10 (right).

Comparison to State-of-the-Art. We demonstrated our

attack on a machine with an i9-9900K stepping 12 CPU with the latest microcode version, 0xCA. Our breach of an SGX enclave on this particular machine exemplifies how CacheOut's advancement over the state of the art in speculative attacks enables it to compromise a system that is resistant to previously known attacks. The i9-9900K contains in-silicon Foreshadow mitigations, which prevents an attacker from directly leaking from the L1-D cache. Fallout cannot target SGX, as the store-buffer is flushed upon swapping to the enclave's page tables, and RIDL's inability to choose which cache line to read from makes it extremely difficult to dump an entire page. As such, CacheOut's unique capabilities for both crossing arbitrary address spaces and selecting cache lines enable another vector for exploiting SGX.

Evaluation. We ran this evaluation on an i9-9900K stepping 12 CPU with the latest microcode version, 0xCA. Leaking an entire page containing ASCII text took 17 minutes and 54 seconds, while performing the attack with noise cancellation took twice as long.

We can see accuracy evaluation of the attack in Figure 10. The left figure displays the accuracy when using the technique from Van Bulck et al. [46], and gives a 2.15% probability of the correct value being in the top 3, and a 4.57% chance of being in the top 5. The middle figure uses our improved paging, and gives a 20.95% probability for top 3 and a 42.26% probability for top 5. The right figure has our most advanced technique for noise filtering, with a probability of 57.25% for top 3 and a probability of 73.80% for top 5.

9 Mitigations

We will now discuss various ways to mitigate CacheOut: disabling Intel Hyper-Threading, flushing the L1-D cache, and disabling TSX.

Disabling Intel Hyper-Threading. Similar to MDS, CacheOut can be used to leak across Intel Hyper Threads. As synchronizing different threads is not straight-forward, one way of mitigating the cross-thread leakage is to disable Intel Hyper-Threading for workloads where security is more important. Unfortunately, disabling Intel Hyper-Threading does not cover the case where the attacker and the victim run on the same CPU thread.

Flushing the L1-D cache. As discussed in Section 4.3, flushing the L1-D cache appears to be an effective mitigation against CacheOut. To offer a complete mitigation on machines affected by both MDS and CacheOut, the kernel would have to issue an `MSR_IA32_FLUSH_CMD` followed by `verw` for each context switch, and for machines that are not affected by MDS, issuing just the `MSR_IA32_FLUSH_CMD` would be sufficient. Unfortunately, flushing the L1-D cache adds significant overhead and only covers the case without Intel Hyper-Threading.

Disabling TSX. To address TSX Asynchronous Abort (TAA) [20] on the newest platforms released after Coffee Lake Refresh (i.e., released after Q4 of 2018), Intel released a series of microcode updates between September and December 2019 that disable transactional memory. These microcode updates introduce `MSR_IA32_TSX_CTRL` (MSR 0x122), where setting the first bit in the MSR disables TSX, and setting the second bit disables CPUID enumeration for TSX capability. Concurrently to our work, after the disclosure of TAA, operating system vendors started disabling TSX through `MSR_IA32_TSX_CTRL` by default on all Intel machines released after Q4 of 2018.

For the majority of Intel machines (i.e., all machines released before 2018-Q4), Intel started rolling out microcode updates to address CPU errata regarding TSX [22]. These microcode updates introduce `MSR_TSX_FORCE_ABORT` (MSR 0x10f), where setting the first bit in the MSR forces any TSX transaction to always abort. However, as of writing, this behavior is not enabled by default, leaving these machines exposed to CacheOut. Given that TSX is not widely used at the time of writing, we thus recommend that TSX be disabled by default on these machines as well.

Microcode Updates. Intel's security advisory [23] indicates that CacheOut (called L1Des in Intel's terminology) will be mitigated via additional microcode updates. We recommend that these be Installed by users of affected systems.

10 Acknowledgments

This research was supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL) under contract FA8750-19-C-0531, and by generous gifts from Intel and AMD.

References

[1] Jeffery M. Abramson, Haitham Akkary, Andrew F. Glew, Glenn J. Hinton, Kris G. Koningsfeld, and Paul D. Madland. Method and apparatus for performing load operations in a computer system. US Patent 5,694,574, Dec 1997.

[2] Jeffery M. Abramson, Haitham Akkary, Andrew F. Glew, Glenn J. Hinton, Kris G. Konigsfeld, and Paul D. Madland. Method and apparatus for blocking execution of and storing load operations during their execution. US Patent 5,881,262, Mar 1999.

[3] Haitham Akkary, Jeffrey M. Abramson, Andrew F. Glew, Glenn J. Hinton, Kris G. Konigsfeld, Paul D. Madland, Mandar S. Joshi, and Brent E. Lince. Methods and apparatus for caching data in a non-blocking manner using a plurality of fill buffers. US Patent 5,671,444, Oct 1996.

[4] Haitham Akkary, Jeffrey M. Abramson, Andrew F. Glew, Glenn J. Hinton, Kris G. Konigsfeld, Paul D. Madland,

Mandar S. Joshi, and Brent E. Lince. Cache memory system having data and tag arrays and multi-purpose buffer assembly with multiple line buffers. US Patent 5,680,572, Jul 1996.

[5] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, 2019.

[6] Milind Bodas, Glenn J. Hinton, and Andrew J. Glew. Mechanism to improved execution of misaligned loads. US Patent 5,854,914, Dec 1998.

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

[8] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, pages 249–266, 2019.

[9] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *Euro S&P*, pages 142–157, 2019.

[10] Jonathan Corbet. The current state of kernel page-table isolation. https://lwn.net/Articles/741878/, 2017.

[11] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security*, 1998.

[12] Andy Glew, Nitin Sarangdhar, and Mandar Joshi. Method and apparatus for combining uncacheable write data into cache-line-sized write buffers. US Patent 5,561,780, Dec 1993.

[13] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: Long live KASLR. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176, 2017.

[14] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *IEEE SP*, pages 490–505, 2011.

[15] Jann Horn. Speculative execution, variant 4: Speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[16] Intel. Deep dive: Intel analysis of L1 terminal fault. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault, Aug 2018.

[17] Intel. Deep dive: Mitigation overview for side channel exploits in Linux. https://software.intel.com/security-software-guidance/insights/deep-dive-mitigation-overview-side-channel-exploits-linux, Jan 2018.

[18] Intel. Speculative execution side channel mitigations. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, May 2018.

[19] 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.

[20] Intel. Deep dive: Intel transactional synchronization extensions (Intel TSX) asynchronous abort. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort, Nov 2019.

[21] Intel. Microcode revision guidance. https://www.intel.com/content/dam/www/public/us/en/documents/corporate-information/SA00233-microcode-update-guidance.pdf, Aug 2019.

[22] Intel. Performance monitoring impact of Intel transactional synchronization extension memory. https://cdrdv2.intel.com/v1/dl/getContent/604224, Mar 2019.

[23] Intel. L1d eviction sampling. https://software.intel.com/security-software-guidance/software-guidance/l1d-eviction-sampling, Jan 2020.

[24] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *IEEE SP*, 2015.

[25] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative load hazards boost

Rowhammer and cache attacks. In *USENIX Security*, pages 621–637, 2019.

[26] Mandar S. Joshi, Andrew F. Glew, and Nitin V. Sarangdhar. Write combining buffer for sequentially addressed partial line operations originating from a single instruction. US Patent 5,630,075, May 1995.

[27] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *DAC*, 2016.

[28] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[29] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, 2019.

[30] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT*, 2018.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[32] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, 2015.

[33] Andrei Lutas and Dan Lutas. Security implications of speculatively executing segmentation related instructions on Intel CPUs. https://businessresources.bitdefender.com/hubfs/noindex/Bitdefender-WhitePaper-INTEL-CPUs.pdf, Aug 2019.

[34] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, pages 2109–2122, 2018.

[35] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, 2006.

[36] Salvador Palanca, Vladimir Pentkovski, Niranjan L. Cooray, Subramaniam Maiyuran, and Angad Narang. Method and system for optimizing write combining performance in a shared buffer structure. US Patent 6,122,715, Mar 1998.

[37] Salvador Palanca, Vladimir Pentkovski, and Steve Tsai. Method and apparatus for implementing non-temporal loads. US Patent 6,223,258, Mar 1998.

[38] Salvador Palanca, Vladimir Pentkovski, Steve Tsai, and Subramaniam Maiyuran. Method and apparatus for implementing non-temporal stores. US Patent 6,205,520, Mar 1998.

[39] Andrew Pardoe. Spectre Mitigations in MSVC. https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/, Jan 2018.

[40] Colin Percival. Cache missing for fun and profit, 2005.

[41] RedHat. Intel November 2019 microcode update. https://access.redhat.com/solutions/2019-microcode-nov, Nov 2019.

[42] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[43] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, pages 552–561, 2007.

[44] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[45] Paul Turner. Retpoline: a Software Construct for Preventing Branch Target Injection. https://support.google.com/faqs/answer/7625886, Jan 2018.

[46] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[47] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Rogue in-flight data load. In *IEEE SP*, 2019.

[48] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. https://foreshadowattack.eu/foreshadow-NG.pdf, 2018.

[49] David Woodhouse. x86/retpoline: Fill RSB on Context Switch for Affected CPUs.

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c995efd5a740d9cbafbf58bde4973e8b50b4d761, Jan 2018.

[50] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.