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 transient-execution attacks, such as RIDL, Fallout, and ZombieLoad, demonstrated that attackers can leak information while it transits through 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 the 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's 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 processes, virtual machines, user and kernel space, and from SGX enclaves.

I. INTRODUCTION

In 2018 Spectre [31] and Meltdown [33] left an everlasting 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 isolation, many follow-up works have demonstrated the dangers posed by uncontrolled speculation and out-of-order execution. Indeed, these newly-discovered *transient-execution attacks* have been used to violate numerous security domains, such as Intel's Secure Guard Extension (SGX) [46], virtual machine boundaries [50], AES hardware accelerators [45] and others [3, 5, 7, 17, 28, 30, 31, 32, 35, 36].

More recently, the security community uncovered a deeper source of leakage: internal and mostly undocumented CPU buffers. With the advent of Microarchitectural Data Sampling (MDS) attacks [4, 43, 48], it was discovered that the contents of these buffers can be leaked via assisting or faulting load instructions, bypassing the CPU's address and permission checks. Using these techniques, an attacker can siphon off data as it appears in the buffer, bypassing all previous hardware and software countermeasures and again breaking nearly all hardware-backed security domains.

Responding to the threat of unconstrained data extraction, Intel deployed countermeasures for blocking data leakage from internal CPU buffers. For older hardware, Intel augmented a legacy x86 instruction, `verw`, to overwrite the contents of the leaking buffers.

This countermeasure was subsequently deployed by all major operating system vendors, performing buffer overwrite on every security domain change. In parallel, Intel launched the new Whiskey Lake architecture, which is designed to be resistant to MDS attacks [4, 43, 48].

While the intuition behind the buffer overwrite countermeasure is that an attacker cannot recover buffer information that is no longer present, previous works [43, 48] already report observing some residual leakage despite buffer overwriting. Thus, in this paper we ask the following questions:

Are buffer overwrites sufficient to block MDS-type attacks? How can an adversary exploit the the buffers in Intel CPUs despite their content being properly overwritten?

Moreover, for Whiskey Lake machines, we note that the nature of Intel's hardware countermeasures is not documented, requiring users to blindly trust Intel that MDS has been truly mitigated. Thus, we ask the following secondary questions:

Are Whiskey Lake machines truly resistant to MDS attacks? How can an attacker leak data from these machines despite Intel's hardware countermeasures?

A. Our Contribution

Unfortunately, we show that ad-hoc buffer overwrite countermeasures as well as Intel's hardware mitigations are both insufficient to completely mitigate MDS-type attacks. More specifically, we present CacheOut, a transient-execution attack that is capable of bypassing Intel's buffer overwriting countermeasures as well as leak data on MDS-resistant Whiskey Lake CPUs. Moreover, unlike prior MDS works, CacheOut

---

allows the attacker to select which cache sets to read from the CPU's L1 Data cache, as opposed to being limited to data present in the 12 entries of the line fill buffers. Next, because the L1 cache is often not flushed on security domain changes, CacheOut is effective even in the case without hyper-threading, 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 and kernel isolation, virtual machine boundaries, and the confidentiality of SGX enclaves.

A New Type of LFB Leakage. 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 loads to leak information from internal CPU buffers. Instead, Intel's `verw` instruction overwrites all the stale information in these buffers, sanitizing their contents. We further note previous observations by ZombieLoad and RIDL [43, 48], which report residual leakage from the Line Fill Buffers (LFBs) on MDS-vulnerable processors despite the `verw` mitigation.

At a high level, the line fill buffers are intended to provide a non-blocking operation of the L1-D cache by handling data retrieval from lower levels of the memory architecture when a cache miss occurs [1, 2]. Despite their intended role of *fetching* data into the L1-D, we empirically find that on Intel CPUs there exists an undocumented path where data *evicted* from the L1-D cache occasionally ends up inside the LFB.

CacheOut Overview. Exploiting this path, in this paper we show a new technique where we first evict data from the L1-D, and subsequently use a faulting or assisting load to recover it from the LFB. This technique has two important security implications. First, in contrast to prior MDS attacks which can only access information that transits through the CPU's internal buffers, CacheOut can leak information present in the entire L1-D cache by simply evicting it. Next, we demonstrate that this information path between L1-D evictions and the LFB has devastating consequences on countermeasures that are rely on flushing buffers on security domain changes. In particular, using Intel's `verw` instruction on does not protect against CacheOut, because the transfer of evicted data from the L1 cache to the LFB occurs well after the context switch and the completion of the associated `verw` instruction.

Attacking Whiskey Lake Processors. We note that the information path from the L1-D to the LFB exists on Intel's latest Whiskey Lake processors, which protect against MDS attacks via hardware countermeasures as opposed to the `verw` instruction. In addition to not leaking from internal CPU buffers, these machines also contain hardware mitigations against prior Meltdown and Foreshadow/L1TF attacks which leak information from the L1 cache. Thus, to the best of our knowledge, CacheOut is the first demonstration of a successful transient-execution attack on Whiskey Lake CPUs, which do not directly leak either from the LFB nor from the L1 cache.

Leakage Amount. As noted above, the presence of data leakage despite using the `verw` instruction has been previously observed by both the RIDL [48] and the ZombieLoad [43] teams. RIDL does not report any rates but only shows leakage via statistical significance. ZombieLoad reports a troubling but insignificant amount of leakage, around 0.1 B/s [43, Section 7]. In this work we show that the leakage is significantly higher, peaking out at around 2.85 KiB/s.

Controlling What to Leak. Our technique of forcing L1 eviction also allows us to select the data to leak from the victim's address space. 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 [21] to leak this data after it transits through the LFB. To further control the location of the leaked data, we observe that the LFB seems to have a *read offset* that controls the position within the buffer from which a load instruction reads. We observe 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.

Finally, by repeating this technique across all 64 L1-D cache sets, CacheOut is able to dump entire 4 KiB pages from the victim's address space, recoving data as well as the positions of data pieces relative to each other. This significantly improves over previous MDS attacks which can only recover information as it transits through the LFB without its corresponding location; TAA has the additional limitation of being able to read only the first 8 bytes of every cache line present in the LFB, leaving the other 56 bytes inaccessible.

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 out of the cache and to the LFB. It does not, however, allow us to leak data that is only *read* by the victim, since this data is not written back to memory and does not occupy a buffer when evicted from the L1 cache. We overcome this by evicting the victim's data from the L1 before the victim has a chance to read it. This induces an L1 cache miss, which is served via the LFB. Finally, we use an attacker process running on the same physical core as the victim to recover the data from the LFB.

Attacking Process Isolation. We show that CacheOut has severe implications for OS-enforced process security boundaries, as it allows unprivileged users to read information belonging to other victim processes, thereby breaching their confidentiality. We demonstrate this risk by implementing attacks on private data across processes in different security domains. Targeting OpenSSL's AES operations, we successfully recover secret keys and plaintext data in both the scenarios with and without hyper-threading. We also developed attacks for both recovering OpenSSL RSA private keys and stealing the secret weights from a FANN artificial neural network.

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 derandomize Kernel

Address Space Layout Randomization (KASLR) in under a second. Furthermore, we demonstrate extraction of stack canaries from the kernel. To the best of our knowledge, this is the first demonstration of acquiring this information via a MDS-type transient-execution attack.

Attacking Intel's Secure Guard Extensions (SGX). We demonstrate that CacheOut can dump the contents of SGX enclaves. We show the recovery of an image from SGX enclaves as well as of EPID keys in debug mode. These attacks are performed on a fully updated Whiskey Lake CPU, which is resistant to previous MDS attacks, including Fallout [4], ZombieLoad [43] and RIDL [48] and TSX Asynchronous Abort (TAA) [21]. In particular, this implies that Intel's current hardware mitigations for SGX are insufficient, allowing an attacker to breach the confidentiality of SGX enclaves.

Moreover, CacheOut can dump the contents of an enclave without requiring it to perform any operation or even execute at all. Instead, we directly dump the memory content of the victim enclave while the enclave is idle. Thus, our attack bypasses all software-based SGX side-channel defenses such as constant-time coding and others [6, 12, 39, 42, 44] which rely on the enclave executing code for its protection.

Attacking Virtual Machines. Another security domain we explore is the isolation of different virtual machines running on the same physical core. We show that CacheOut is effective at leaking data from both virtual machines and from hypervisors. Experimentally evaluating this, we completely derandomize the Address Space Layout Randomization (ASLR) used by the hypervisor and recover AES keys from another VM.

Avoiding Hyper-Threading. While CacheOut is most effective across hyper-threads, we can nonetheless use it to recover information in a time-shared environment, with hyper-threading being disabled, even in the presence of the verw countermeasure. The core failure is that the verw instruction only flushes the internal CPU buffers, and not the L1 cache. Thus, an attacker can evict cached data left by the victim and subsequently recover it from the leaky line fill buffer. Finally, CacheOut is able to defeat the hardware countermeasures on Whiskey Lake CPUs, both with and without hyper-threading.

Summary of Contributions. In this paper we make the following contributions:

- We present CacheOut, the first transient-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 overwriting the contents of internal CPU buffers between context switches, even when hyper-threading is disabled.
- We demonstrate the effectiveness of CacheOut in violating process isolation by recovering AES and RSA keys as well as 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 violates isolation between two virtual machines running on the same physical core.

- We breach SGX's confidentiality guarantees by reading out the contents of an SGX enclave and recovering the machine's attestation keys from a fully updated system.
- We demonstrate that some of the latest Intel CPUs are still vulnerable, despite all of the most recent patches and mitigations. In particular, to the best of our knowledge, CacheOut is the first transient-execution attack to break Intel's MDS-resistant Whiskey Lake architecture.
- We discuss why current transient-execution attack mitigations are insufficient, and offer suggestions on what countermeasures would effectively mitigate CacheOut.

B. Current Status and Disclosure

Van Schaik et al. [48] 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 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 and KU Leuven.

Current Status. In November 2019, after our initial disclosure of CacheOut, Intel attempted to mitigate TSX Asynchronous Abort (TAA) [21], a variant of MDS which allows an attacker to leak information from internal CPU buffers. Consequently, in November 2019 Intel published microcode updates that enable turning off Transactional Memory Extension (TSX) on CPUs made after Q4 2018. These have been deployed by OS vendors, preventing some variants of CacheOut on these machines. However, for SGX, a malicious OS can always re-enable TSX. As we show in this paper, this results in a loss of confidentiality due to our breach of Intel's TAA countermeasures for protecting SGX.

Next, we note that the majority of deployed Intel hardware is older, and was released prior to Q4 2018. For these systems, TSX is enabled by default at the time of writing, leaving them vulnerable to all variants of CacheOut. Finally, Intel had indicated that microcode updates mitigating the root cause behind CacheOut will be published on June 9th, 2020. We recommend these be installed on all affected Intel platforms to properly mitigate CacheOut.

II. BACKGROUND

A. 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
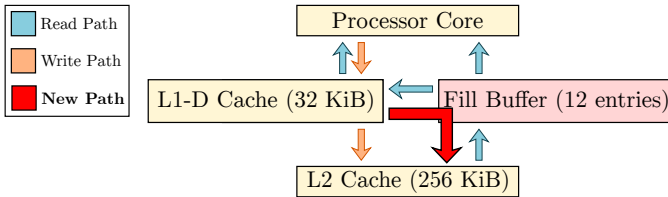
Fig. 1: The data paths within the CPU core, with the paths for loads marked in blue, the path for stores in orange, and the new undocumented path that we uncovered marked in red.

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, 52] 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 [27, 29, 34, 40, 41] 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.

B. Microarchitectural Buffers

In addition to caches, modern processors contain multiple microarchitectural 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 core.
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 [1, 2] 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 issues 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.

A New Data Path. As mentioned above, while the LFB is responsible for handling data coming into the L1-D cache, we empirically demonstrate the existence of an undocumented data path between L1-D evictions and the LFB (marked in red in Figure 1). We then exploit this path by causing L1-D evictions and subsequently leak the evicted data from the LFB. In addition to bypassing the verw instruction by moving data into the LFB after the verw-induced buffer overwrite, we also show that this path exists on MDS-resistant Whiskey Lake machines, making these vulnerable to CacheOut.

C. Speculative and Out-of-Order Execution

Modern processors try to predict future instructions and execute instructions as soon as the required data is available, rather than following the strict order stipulated by the program. 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.

D. Transient-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 *transient-execution attacks* have demonstrated the possibility of leaking such data [5, 7, 17, 30, 31, 32, 36]. 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 modulate the state of microarchitectural components based on the secret data. The attacker then probes the state of the microarchitectural component to determine the secret data.

Most published transient-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.

E. RIDL, ZombieLoad, Medusa vs. CacheOut

Several prior works explore leakage from internal CPU buffers. These include RIDL [48], ZombieLoad [43], Fallout [4], and Medusa [37], collectively known as *MDS attacks*.

RIDL. RIDL [48] analyzes the Line Fill Buffers (LFBs) and the Load Ports. Focusing mainly on the case with hyper-threading, the work shows that faulting loads can be served from these buffers, bypassing any address and permission checks. This allows an attacker to use the sibling core to siphon off data as it appears in the buffer, compromising the confidentiality of nearly all hardware-backed security domains.

However, while RIDL conjectures that data evicted from the L1 cache is moved into the leaky LFB, and even shows some statistical evidence of such leakage, it does not study the security implications of this issue nor of Intel's MDS buffer flush countermeasures. Moreover, RIDL also lacks control over what data the attacker is leaking, and instead relies on averaging techniques to filter the data from the acquired noise. Finally, the lack of control over what can be leaked from the L1 cache implies that RIDL only demonstrates attacks in the hyper-threaded case, where the attacker siphons off data from the LFB as the victim accesses it.

ZombieLoad. ZombieLoad [43] also analyzes leakage from the LFBs. Extending RIDL's findings to loads that require microcode assists, ZombieLoad shows that leakage exists even without using faulting loads. ZombieLoad also demonstrates LFB leakage from the Cascade Lake architecture, that Intel claims to be the first MDS-resistant architecture.

Similarly to RIDL, ZombieLoad mentions the possibility of leakage via L1 evictions to the LFB. However, ZombieLoad proceeds to argue that the leakage is negligible, limited to 0.1 bytes per second. ZombieLoad also suffers from limitations similar to RIDL with regards to the attacker's ability to control the leakage, resorting to Domino-bytes averaging techniques for data processing with the attacker and victim running on different threads of the same physical core. While the ZombieLoad paper mentions kernel attacks in the case without hyper-threading, Section 6.5 in [43] only demonstrates attacks using artificially inserted kernel gadgets, at a rate of 10 seconds per byte. Finally, while ZombieLoad does mention the possibility of hypervisor and cross-VM leakage, Schwarz et al. [43] only demonstrate a cross-VM covert channel.

Medusa. In concurrent independent work, Moghimi et al. [37] presented Medusa, a variant of ZombieLoad that recovers information from write-combining (WC) operations [8], for which the LFB is responsible on Intel CPUs [22, vol. 3 pg. 6-38]. By focusing on leakage from write combining done in the LFB during `rep mov` and `rep stos` operations, Medusa is able to obtain a cleaner LFB leakage signal, as it avoids recovering values from other memory operations. Finally, as OpenSSL uses fast memory copying to copy RSA keys, and the kernel to transfer data, Medusa demonstrated the recovery of such data across hyper-threads.

However, Medusa is (intentionally) limited to only recovering values during write combining, and is unable to recover leakage from other memory operations. Being a variant of ZombieLoad, the attacker has no knowledge or control over the exact offsets, and can only partially sample the leaked data. This results in slow leakage rates of 12 B/s for kernel data, and the need for Domino-bytes signal averaging. For unstructured

data (e.g., RSA keys), a 400 CPU hour lattice attack [9] is needed to recover the 1024-bit RSA key from the raw leakage, which is obtained during a 7 minutes measurement phase.

CacheOut. In this work, we also focus on leakage from Intel's LFB. However, unlike RIDL, ZombieLoad, and Medusa we do not wait for the information to become available in the LFB, and instead use cache evictions to actively move it to the leaky LFB. We show that the leakage is far greater than the 0.1B/s conjectured by ZombieLoad, peaking out at 2.85KiB/s. Next, we show that by using cache evictions the attacker can choose what information he is interested in leaking, thus avoiding the need to use noise-averaging techniques. Ironically, in addition to bypassing Intel's `verw` countermeasure that overwrites the contents of the leaky buffers, we go a step further and show how `verw` can actually be used to improve our attack's leakage rate. Furthermore, we show the effectiveness of CacheOut in breaking the isolation between processes, VMs, hypervisors, and SGX enclaves. Finally, we show that MDS attacks are still effective on Intel's latest MDS-resistant Whiskey Lake CPUs.

F. 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 end of a transaction, respectively. These instructions form a transaction where either all of them execute to completion or none of them at all. All of the transaction's instructions execute speculatively, but are only committed if execution reaches the `xend` instruction. If during the execution of a transaction 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 [22, vol. 2A, pg. 3-139–3-142]. TSX Asynchronous Abort (TAA) [21, 43, 48] exploits this behavior in a transient-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 data from the LFB. Since the load does not complete successfully, the load proceeds with remnants from a previous memory access, allowing the attacker to sample LFB data [19, 21]. We refer the reader to Appendix A for a TAA code example.

III. CPU MITIGATIONS AND THREAT MODEL

Since the discovery of Spectre [31] and Meltdown [33], there have been numerous works that exploit speculative and out-of-order execution to violate hardware-backed security domains [5, 7, 17, 30, 31, 32, 36]. In response, recent Intel processor contain hardware-based countermeasures aimed at addressing these attacks. Table I summarizes these counter-measures in some recent Intel processors. For processors that

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 I: Countermeasures for transient execution attacks in Intel processors. ✓ and ✗ indicate the existence or absence of in-silicon countermeasure for the attack.

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 [20, 33] shows that an attacker can bypass the protection of kernel memory. The attack requires that the virtual address is present in the address space and that the data it refers to is 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 [18, 46, 50] 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` 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 [4], RIDL [48], ZombieLoad [43], and Medusa [37] show that attackers can leak data transiting through various internal microarchitectural buffers, such as the LFBs discussed in Section II. To address these issues for older hardware, Intel provided microcode updates [24] that repurpose the `verw` instruction to flush these microarchitectural buffers by overwriting them. The operating system has to issue the `verw` instruction upon every context switch to effectively flush these microarchitectural buffers.

The Whiskey Lake Architecture. In an attempt to mitigate MDS attacks in hardware, Intel also released the Whiskey Lake architecture, which contains hardware mitigations to MDS attacks (i.e., RIDL, Fallout, and ZombieLoad) as well as to Meltdown and Foreshadow/L1TF. In particular, Whiskey Lake machines are not vulnerable to previous MDS techniques that leak from internal buffers or to older generation Meltdown/Foreshadow attacks which leak the contents of the L1-D cache. As we show however, these machines are vulnerable to CacheOut, making our attack the only attack currently capable of leaking the contents of L1-D on these machines.

Threat Model. We assume that the attacker is an unprivileged user, such as a VM, or an 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.

IV. CACHEOUT: EXPLOITING CACHE EVICTIONS

We now start our exposure of CacheOut and show that 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 hyper-threads of the same physical 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 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 writes 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
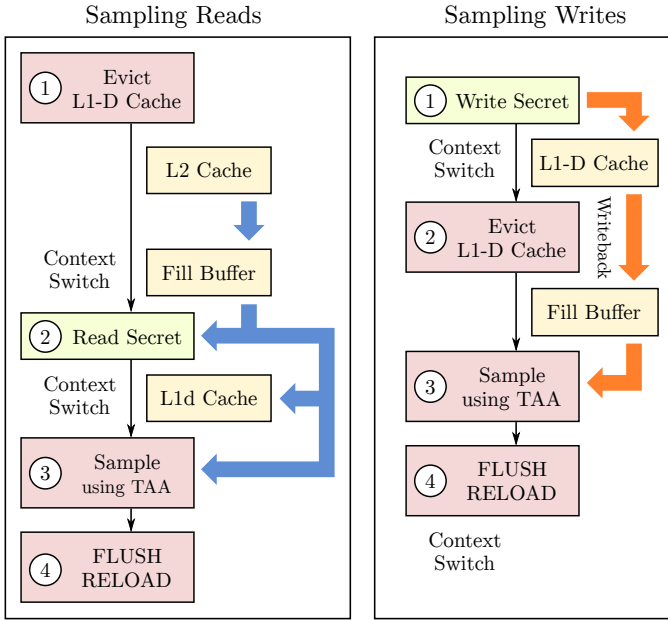
Fig. 2: Overview of how we use TAA to leak from loads and stores through Fill Buffers. Victim activity, attacker activity, and microarchitectural effects are shown in green, red, and yellow respectively. The context switches both illustrate the OS flushing the MDS buffers before switching to the other process as well as switching between actual hyper-threads.
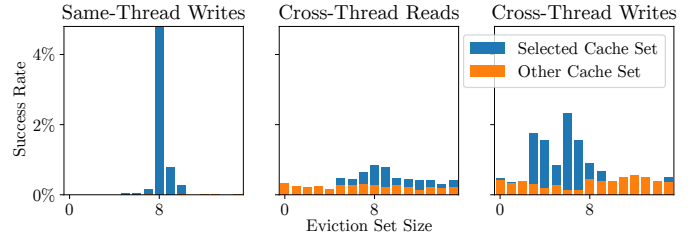

Fig. 3: Number of loads/stores required to evict the victim's cache line containing a fixed value. The blue bars indicate how often we observe the correct value from the selected cache-line, while the orange bars indicate how often we observe data present in a different cache (due to noise). Finally, we ran 10,000 iterations per tested set.

passes through the LFBs. Thus, in Step 3, the attacker uses TAA to sample the buffer and subsequently uses FLUSH+RELOAD to recover the value (Step 4). Finally, unlike the case of reads, this attack can be performed both with and without hyper-threading.

A. Exploiting 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. Consequently, the attacker can allocate eight 4 KiB memory pages to cover the whole cache. The attacker then constructs eviction sets from memory addresses with the same page offset.

Measuring L1-D Eviction. To measure the number of accesses needed 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 eviction set sizes, and under three different attack scenarios. Figure 3 contains a summary of our results. In the first scenario (left) the victim and the attacker time-share the same hyper-thread. 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. Next, we note the decreased performance of the attack with other eviction set sizes (both

smaller and larger sets). We conjecture that small sets cannot evict the victim's element due to the cache's LRU replacement policy while larger eviction sets increase noise due to cache pressure.

We also test the cases across hyper-threads, targeting the victim's memory reads (middle) and then victim writes (right). While the results here are not as strong, the likelihood of getting the victim's data from the correct cache line is still higher than getting data from other cache lines. For victim reads, we still get the best results with an eviction set of size eight while an eviction set of size six works best for writes. We suspect that the cause is the increased L1-D contention due to having two active hyper-threads.

Measuring Data Selection. Demonstrating our ability to select which cache set to leak, we repeat the experiments of Figure 3, this time varying the cache set the victim uses and the cache set the attacker evicts. As can be seen in Figure 4, in all scenarios the attacker can target a specific cache set, correctly leaking its values, albeit with some noise for the case of cross-thread victim writes. Finally, we note that this is a qualitative improvement over prior works such as RIDL [48], ZombieLoad [43] and Medusa [37], as these are limited to leaking data already present in the 12 entries of the LFB, as opposed to leaking data from the entire L1-D cache.

B. Selecting Cache Line Offsets

So far we have shown how to control the cache set from which CacheOut leaks data. However, we note that like previous TAA attacks [48], we still do not have control over the offset within the 64-bytes cache line from which we read. In particular, TAA [48] is only able to leak the first 8 bytes out of every 64-byte cache line, leaving the other 56 bytes unreachable.

Tackling this limitation, we discovered that that the offset of load instructions that *follow* the TAA attack also controls the cache line offset from which the TAA attack reads. More specifically, Listing 1 shows our leakage primitive that allows us to control the offset from which we read data. As we can see, the code is basically the same as the TAA leak primitive, but we added two `movq` instructions in Lines 16–17.

Analysing the CacheOut Primitive. We note that the leakage in the CacheOut primitive occurs in Line 11. At a first
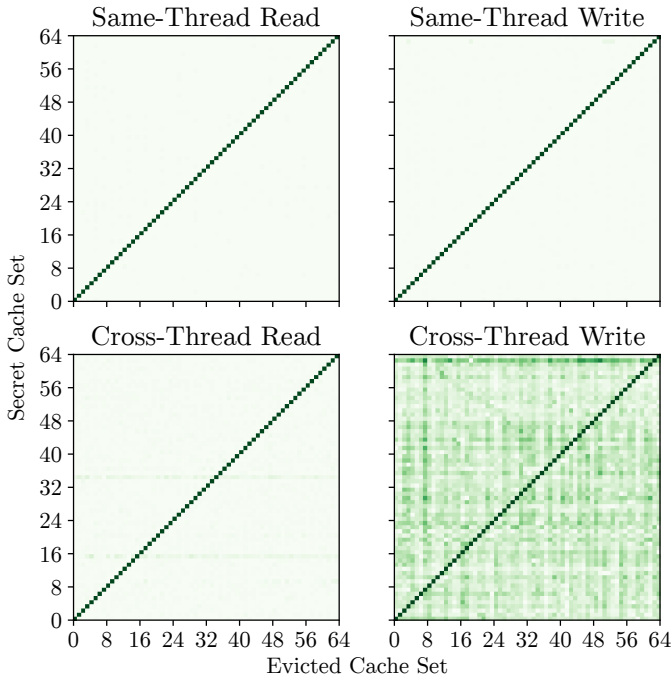
Fig. 4: The victim loads/stores a secret to every possible cache line (y-axis), while the attacker evicts every possible cache line (x-axis) to leak it. We ran 10K iterations per test.

```
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      xend
19  abort:
20      retq
```

Listing 1: CacheOut leak primitive.

glance it seems odd that later movq instructions 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. Thus, due to out-of-order execution they can execute *before* the 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 from which to read data. This read offset gets reused by the leaking movq when the transaction aborts, thereby allowing us to select the desired cache line offset.

Reducing Noise and Data Stitching. Modern Intel CPUs typically employ two load ports, which allows them to execute two load instructions in parallel. Exploiting this, in our attack we duplicated and interleaved the instructions from lines 11–14 in Listing 1, such that they execute two load instructions in parallel on both load ports. Compared to executing a single load instruction, we found that the strength of our signal doubles when both load instructions refer to the same offset.

Next, this technique also allows us to avoid using the Domino bytes method used in prior MDS works [37, 43, 48]. Instead, in our attack, we leak two consecutive data bytes at a time, where two consecutive attack iterations share a common data byte (at offset 2 in the first iteration and offset 1 in the second iteration). Observing the leakage via the cache channel, we stitch together data that matches on the overlapping data byte as shown in Figure 5.
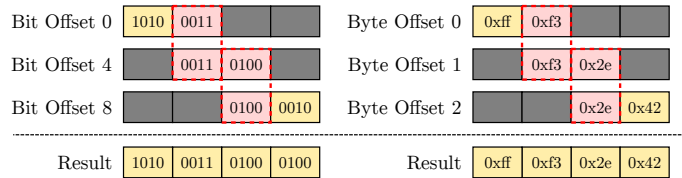

Fig. 5: On the left we show how the Domino attack samples a *byte* at a time and uses four bits of every byte to stitch data together. On the right we show our technique for CacheOut where the attacker samples *two bytes* at a time and uses the leading and trailing byte to stitch data together, effectively doubling the attack's speed.

Evaluating Offset Selection. To evaluate our offset selection method, we use a victim process that chooses a byte offset and writes a 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 6, we can successfully select the offset in the cache line from which we leak. Next, combining this behavior with the L1-D cache set eviction method described in Section IV-A, CacheOut is equally effective against all addresses, and improves on prior MDS attacks by allowing the attacker to access any data located in the L1-D cache while being able to select the precise byte he is interested in leaking.

Evaluating Leakage Amount. Finally, we evaluate the rate of information leakage resulting from exploiting targeted L1-D evictions into the leaky LFB. Our victim writes some byte value to a known cache location, while the attacker running on the same physical core uses our address selection techniques in order to recover the victim's writes 10K times. We distinguish between not leaking anything, leaking the correct value and leaking the incorrect value. We find the leakage rate, i.e., how often we leak the correct value over a certain period of time, to be much larger than ZombieLoad's 0.1B/s, peaking out at 2.85KiB/s for reads and 2.38KiB/s for writes.
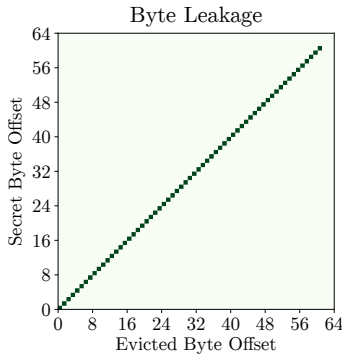
Fig. 6: The victim loads/stores a secret byte to every possible offset within a fixed cache line (y-axis), while the attacker tries to leak from every possible byte offset (x-axis).

C. Determining the Leakage Source

Flushing the MDS Buffers. While the `verw` instruction is now used by Intel as a defense against MDS attacks, the ability to overwrite the contents of MDS-affected buffers is also helpful in determining the source of the leakage observed by CacheOut. More specifically, our attacker issues the `verw` instruction after evicting from the cache, but before executing the leakage primitive. When the victim and attacker execute sequentially on the same hyper-thread, this completely removes the signal. Thus, we conclude that the actual leakage stems from one of the MDS buffers. Next, when we move the `verw` instruction before evicting from the cache in our attacker, 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 LFB when it is written back to the L2 cache.

Exploiting `verw`. Ironically, we discovered that issuing the `verw` instruction before evicting from the cache significantly improves the signal for victim writes, both in cross-thread and same-thread scenarios. As the `verw` instruction does not require root privileges, we are able to abuse Intel's MDS countermeasures to reduce noise encountered during our attack by having the attacker use `verw` to remove unwanted values from the LFB. We conjecture that this attacker-executed `verw` removes all values but the leaked one from the LFB, thereby increasing the probability of the leaked value be successfully recovered by TAA. To confirm this we run an experiment where we try to leak from writes in the same-thread and cross-thread scenarios, as well as from reads in the cross-thread scenario, both with and without `verw`. Without `verw`, we report an actual throughput of 26.57B/s, 2918.33B/s and 343.25B/s for same-thread writes, cross-thread reads and cross-thread writes respectively. With `verw`, we report an actual throughput of 81.45B/s, 1833.93B/s and 2433.97B/s, respectively.

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 flushing the L1-D using `MSR_IA32_-FLUSH_CMD` (MSR 0x10b) between the victim access and the cache eviction. We find that in the same-thread case,

this completely removes the signal. This again supports the hypothesis that evictions of modified data from the L1-D transit through the LFB, where it is leaked by CacheOut.

V. CROSS PROCESS ATTACKS

To demonstrate the implications of CacheOut, we developed multiple proof of concept attacks wherein an unprivileged user process leaks confidential data from another process: recovering AES keys, RSA keys, and the weights of a neural network. Moreover, in our examples we demonstrate how address selection enables more powerful attacks. That is, CacheOut allows the attacker to select the locations to read in the victim's address space, rather than waiting for data to become available in the LFB. In particular, unlike Zombie-Load [43] and RIDL [48], we can effectively leak random-looking data spanning multiple cache lines. This allows us to lift the known-prefix or known-suffix restriction of [43, 48], which requires prior knowledge of some prefix or suffix of the data to leak. Indeed, instead of using a known prefix or suffix, we use CacheOut to simply read as much data as we can from the L1-D cache. As we know the location of the data pieces relative to each other, we are able to partially reconstruct a portion of the victim's address space that is located inside the L1-D cache. Next, we exploit redundancies in the data such as derived AES keys or the relationship between p, q and $n = pq$ for RSA in order to find these inside the reconstructed parts of the victim's memory.

Finally, we also improve on ZombieLoad and RIDL [43, 48] by showing attacks with and without hyper-threading.

Experimental Setup. We run the attacks presented in this section on two machines. The first is equipped with an Intel Core 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 machine uses microcode version `0xca`, while the latter uses `0xb8`.

A. Recovering AES Keys

Same-Thread Leakage. Our cross-process attack aims to leak plaintext message and key material from an AES decryption operation. 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 hyper-thread, and repeatedly calls `sched_yield()` to allow the victim to run and decrypt the ciphertext. After the victim finishes running, the attacker evicts the set of interest from the L1-D cache into the LFB. The attacker then uses TAA to sample the decrypted message from the LFB; see Figure 7 for an illustration. Furthermore, we found that if the buffer holding the plaintext messages shares cache line with other data, we can also sample that data. For instance, we were able to sample the 128-bit AES key from the LFB when the victim writes the plaintext message to the same 64B cache line as the 128-bit AES key. Finally, even though we artificially instrumented the victim process to

9

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.

Cross-Thread Leakage. We also run our experiment with the victim and attacker running on the same physical core, but different threads and without using `sched_yield()` in either attacker or victim. Here, we are not only able to see the decrypted plaintext and AES key, but also the expanded round keys used in each of the AES's rounds. Unlike the same-thread case, we do not require the assumption that the victim performs a write operation into the 64-byte cache line containing the 128-bit AES key. Finally, since both the initial AES key and the round keys are laid out consecutively in memory, we can use CacheOut's address selection capability to recognize the AES keys from within the leakage data.

Locating AES Keys. To locate the AES keys inside the leakage without knowing the location of the key data structure inside the victim's memory, we follow the technique of [15] and consider every consecutive chunk of 128 bits, 192 bits or 256 bits of data as a key candidate. We then expand the candidate into the AES round keys and check if they match the following chunk of data, up to a certain threshold. If we find such a series of round keys, we conclude that the candidate is the correct key.

Experimental Results. Compared to previous techniques [37, 43, 48], our attack benefits from CacheOut's cache line selection capabilities which removes the need to perform online noise reduction techniques (e.g., the "Domino-bytes" method of [37, 43]). In addition, unlike previous works, CacheOut has the ability to work with and without hyper-threading and to recover the AES round keys. By targeting specific cache lines from which to leak, our attack classifies plaintext bytes with 96.8% accuracy and 128-bit AES keys with 90.0% accuracy, taking 15 seconds on average to recover a single 64 bytes cache line over ten runs for the same-thread setup.

For our cross-thread setup, we sample data from all 64 cache lines in our online phase at 500 iterations of TAA per byte offset. This part of the attack takes 76.2 seconds on average
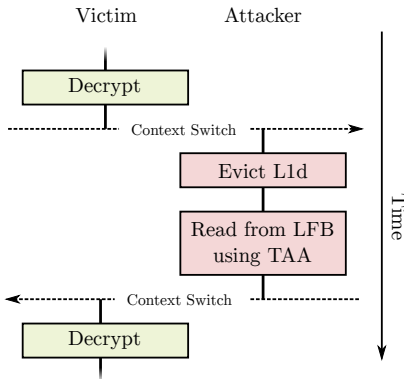

Fig. 7: After decrypting, the victim writes the plaintext, bringing it into the L1-D cache. The attacker can then evict it from the L1-D cache and use TAA to read it from the LFB.

over ten runs, and we leak data with a raw throughput of 8.90KiB/s and an actual throughput of 63.39B/s. Furthermore, we observe 98.34% of the AES key and round keys, where the initial AES key appears at three different locations for 128-bit, and two different locations for 256-bit providing us with additional redundancy. We then proceed to locate the AES key in our offline phase, which takes 183.29s on average.

B. Recovering RSA Private Keys

Leaking from OpenSSL RSA. To attack RSA in OpenSSL, we run a victim that repeatedly decrypts a given ciphertext in a loop. In our setup, both the victim and the attacker run on different threads on the same physical core, where the attacker samples data from the victim using TAA, without the need for `sched_yield()` in both attacker and victim. Within the sampled data, we observe 8 byte chunks of p and q, though not in any particular order. Address selection does not help us in this particular scenario, as we observe these chunks of 8 bytes for all possible 8-byte aligned offsets. Within the raw dump of the sampled data, we are able to observe all of the chunks of p and q, but without address selection, we cannot determine whether the chunks are from p or q, or where in p and q they appear.

Notably, we did not observe any data from the other components of the private key (i.e., d, d_p, and d_q). Inspecting OpenSSL's modular exponentiation code, we find that it requires p and q to be repeatedly loaded into the cache, due to OpenSSL's use of the Chinese Reminder Theorem (CRT). We thus conjecture that the leakage signal observed from the loadings of p, q dominates any other signal. Our algorithm for reconstructing p and q from the unordered chunks is described in Appendix B.

Key Extraction Results. For this experiment, we generated 512-bit, 1024-bit, 2048-bit and 4096-bit RSA keys. We then performed the online phase of our attack, sampling sufficient key data from our victim. We gathered 100% of the key data in all cases by sampling from a single cache line for 2048-bit keys and smaller, and from four cache lines for 4096-bit keys. To sample data at each byte offset, we used 3K iterations for 1024-bit and smaller, and 5K iterations for 2048-bit and larger. Our online phase took 7.4s, 7.4s, 13s, and 51s for the different key sizes, averaged over 5 runs, compared to Medusa's [37] 7min for 1024-bit.

Key Reconstruction. Next, in the offline phase, we recovered the RSA private keys from the collected data. We were able to recover 512-bit, 1024-bit, 2048-bit and 4096-bit RSA private keys from the sampled data in 0.3s, 0.3s, 3.5s and 82.8s on average respectively, with the worst-case performance recorded being 186s. We confirmed correct private key recovery via the corresponding public key. Finally, we note that CacheOut's cleaner leakage signal allows us to improve Medusa's [37] 400 CPU hour result for 1024-bit keys to mere seconds, while also attacking larger 2048 and 4096-bit keys.

C. Attacking Neural Networks

To further demonstrate the utility of address selection, we also use CacheOut to steal the weights from an artificial neural network (ANN). We note that these weights are valuable IP for companies that invest resources on training networks, which creates economic incentive for stealing such weights [53]. In this section, we consider an attacker that aims to leak the weights from a propriety victim neural network classifier.

Recovering Weights from FANN. We demonstrate our weight recovery attack against the popular Fast Artificial Neural Network (FANN) Library. The victim uses the generic FANN model created by `fann_create_standard()` to repeatedly classify a randomly chosen piece of English text as one of three languages. On a parallel logical thread on the same physical core as the classifier, the attacking process uses 5K iterations to sample data from each byte offset, without the need for `sched_yield()` in either victim or attacker. In this manner, we observe 98.4% of the weights among the extracted data. However, the vast amount of raw data that CacheOut leaks complicates the process of identifying the network's weights, requiring us to use a number of techniques to clean the noise and identify weights' values.

Exercising Address Selection. The model has 376 weights, with each weight represented with 32 bits, resulting in a 1504B array. Since the weights are stored sequentially in an array allocated by `calloc()`, finding the start of the array reveals the page offsets of all of the weights. After instrumenting the FANN classifier to reveal the address of the weights' array, we found that the array always starts at a fixed page offset. Thus, the attacker can find this location in an offline phase, thereby enabling her to specifically target the cache lines containing the weights during the online phase. With the naive approach of simply selecting the 8 byte value that has been seen the greatest number of times for each offset containing a weight, we achieve 63.0% accuracy for determining the value of each weight. In Appendix C we describe how to improve the accuracy to 96.1% by exploiting both the weights' storage format and the observation that the weights of a neural network tend to be small. Crucially, we note that without address selection, the attacker would not be able to map the recovered weights to the neural network model. As the 1504B weight array spans 23 different cache lines, even if the attacker could accurately identify each weight with 100% accuracy, she would not be able to determine which weight connects which two neurons.

Experimental Results. We performed an experiment where we try to leak the weights from a trained neural network. Our attacker took 40s to run on average over ten runs with a raw throughput of 17.08KiB/s and an actual throughput of 662B/s. We observed 98.4% of the weights and recover the weights with top-1, top-3 and top-5 accuracies of 95.2%, 96.6% and 96.6% respectively.

VI. ATTACKS ON LINUX KERNEL

CacheOut can also leak sensitive data from the unmodified Linux kernel, even when hyper-threading is disabled. We demonstrate how by developing attacks for breaking KASLR and recovering secret kernel stack canaries.

A. Derandomizing Kernel ASLR

KASLR Overview. Kernel Address Space Layout Randomization (KASLR) is a defense-in-depth countermeasure to binary exploits. By randomizing offsets of entire code sections, the kernel impedes control flow redirection attacks, which require knowledge of the location of targeted code pieces.

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. The attacker 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 attacker from the kernel, we use TAA to leak stale L1-D data leftovers 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, running the attack code on a machine running the same kernel version as the victim. 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.

B. Defeating Kernel Stack Canaries

Stack Canaries Overview. Stack Canaries [11] are another widely deployed defense-in-depth countermeasure to binary exploits.More specifically, these aim to protect against stack-based buffer overflows, 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).

Extracting Kernel Canary Values. We used CacheOut to leak the Linux kernel's 64-bit stack canary value, which is shared for all kernel functions running on the same core in the context of the same process. The attacking code is similar to the KASLR break, but instead of repeatedly calling `sched_-yield()`, we execute 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 23s when evaluated on an i9-9900K stepping 12 CPU with microcode `0xca` running Ubuntu 16.04. To our knowledge, CacheOut is the first microarchitectural side-channel that manages to recover stack canaries from the kernel. This is made possible by the address selection
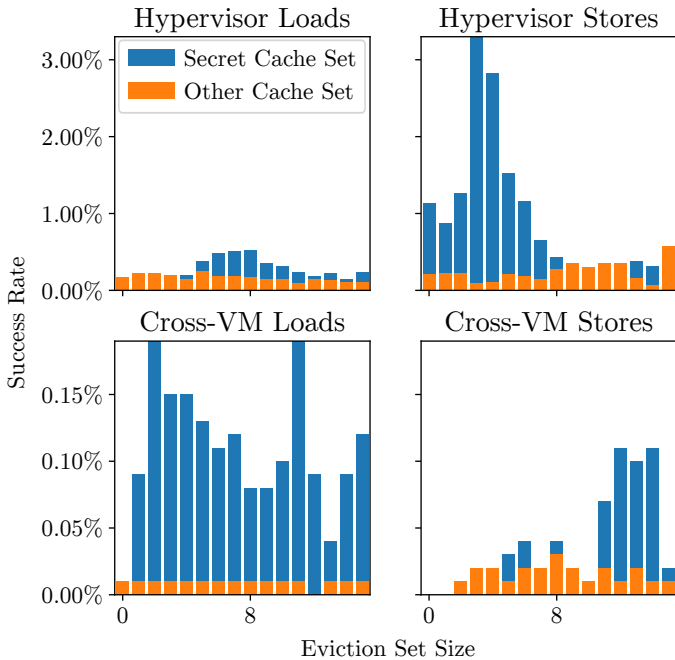
Fig. 8: The number of loads/stores required to evict the L1-D cache sets for loads (left), stores (right), against the hypervisor (top) and across VMs (bottom).

capabilities, as a completely random 64-bit value is extremely difficult to detect without targeting a particular cache line.

VII. BREAKING VIRTUALIZATION

Infrastructure-as-a-Service (IaaS) cloud-computer services provide their end-users virtualized system resources, where each tenant runs in a separate VM. Modern processors support virtualization by means of extensions where the hypervisor can create and manage these VMs that each run their own OS and applications in an isolated environment, analogous to how an OS creates and manages processes. In this section, we demonstrate that CacheOut can break VM 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 ran 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`. We used QEMU 2.11.1 with KVM enabled and 1GB hugepages set up on two different threads on the same physical core.

Evicting L1-D Cache Sets. We perform the same experiment as in Section IV-A to determine the number of loads and stores necessary to evict any L1-D cache set across VMs and to attack the hypervisor across CPU threads. Figure 8 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 IV-A. 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 9, 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.
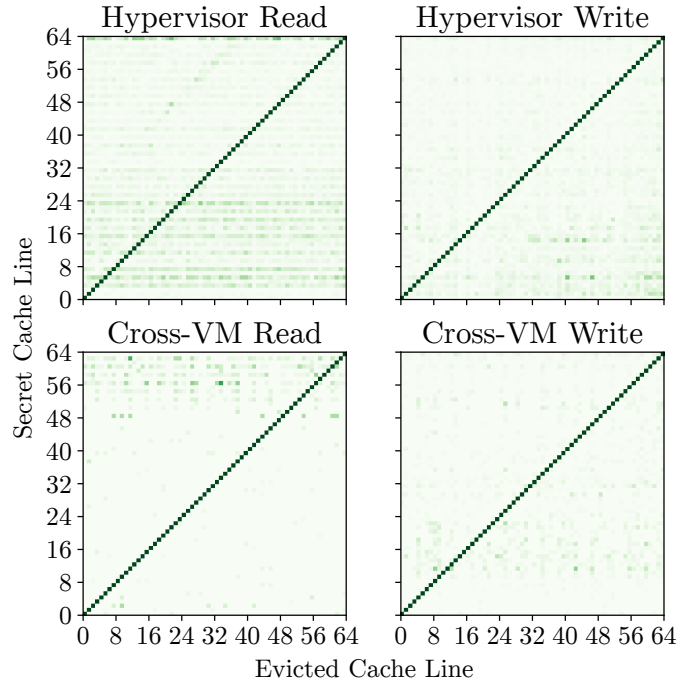

Fig. 9: The hypervisor and victim VM loads/stores a secret from/to every possible cache line (y-axis), while the attacker VM tries to evict every possible cache line (x-axis) to leak it.

A. 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 targeted a specific cache line to leak from, and ran our experiment three times. For each run, we attempted to leak each key byte 10,000 times. During all three runs, the bytes corresponding to the victim's AES key were observed during 20 out of the 10,000 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. Upon subtraction, we were able to recover 75% of the key bits on average across the three runs. Finally, as in the case of Section V, it took about 15 seconds on average to leak a single 64-byte cache line.

B. Leaking RSA Keys Across VMs

We adapted the experiment from Section V-B for stealing RSA private keys across VMs. We use the same victim as in Section V-B, which runs RSA decryptions in a loop inside a VM. From within a VM on a parallel hyper-thread running on the same physical core, we use CacheOut to sample data from all cache lines. When repeating the attack from Section V-B, we are able to observe 100% of the chunks of p and q from the extracted data. Compared to the cross-process scenario from Section V-B, the VMs introduce a substantial amount of noise, which we overcome by sampling with CacheOut over a larger number of iterations.

Experimental Results. We found that we can extract 100% of the key data using 5K iterations for 512-bit RSA keys and 10K iterations for 1024-bit RSA keys and larger to sample data per byte offset. This resulted in an average run time of 11.71s, 23.16s, 23.24s and 24.32 to sample the data for 512-bit, 1024-bit, 2048-bit and 4096-bit RSA keys respectively over ten runs. In addition, we observed an actual throughput of 1.21KiB/s, 2.15KiB/s, 2.68KiB/s and 5.82KiB/s for the different key sizes. Our offline phase where we recover the RSA key from our collected data took 2.18s, 2.27s, 33.96s and 95.50s on average for the different key sizes. Finally, we note that Medusa [37] did not demonstrate any cross-VM data extraction attacks (besides a covert channel), presumably due to noise.

C. Stealing FANN Weights Across VMs

We also reproduced the results from Section V-C for stealing the weights from FANN. In our experiment, the same victim from Section V-C repeatedly classifies on one VM, while an attacker uses CacheOut to sample from an attacking VM running on a parallel Intel hyper-thread on the same physical core. When using 5,000 iterations of CacheOut to leak data from each of the targeted locations, the average run time of the attack is 376.69s with 99.90% of the weights observed among the recovered data. We achieve top-1, top-3, and top-5 accuracies of 93.95%, 96.08%, and 96.30% respectively.

D. Breaking Hypervisor ASLR

Similarly to kernels, hypervisors also deploy ASLR. To leak any information regarding ASLR from the hypervisor, we first find a controlled way to trap into the hypervisor. One way of trapping into the hypervisor is by issuing `cpuid` from the VM, 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 from which we leak kernel pointers. In order to disambiguate the kernel pointers we find from actual hypervisor pointers, we simply reboot our VM. This 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 allows us to tell apart the pointers we leak from the hypervisor, as the kernel pointers belonging to the attacker's VM are likely to change after a reboot.

Hypervisor ASLR Attack Evaluation. We first perform an offline phase to determine whether there are static locations from which we can leak hypervisor addresses. 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.

VIII. 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. In this section we present attacks for dumping the contents of an SGX enclave, thereby violating SGX's confidentiality guarantees. Moreover, unlike RIDL [48] and ZombieLoad [43], the ability to control which memory address we would like to leak allows us to recover unstructured large secrets, such as images. Following SGX's threat model, we assume a malicious OS that aims to breach enclave confidentiality. We also assume that hyper-threading is enabled and that the attacker runs in parallel on the same physical core as the victim enclave.

Experimental Setup. We ran the attacks in this section on an Intel Core i7-8665 CPU (Whiskey Lake), running Linux Ubuntu 18.04.3 LTS with a 5.0.0-37 generic kernel with microcode version `0xca`. This machine is fully mitigated against MDS, meaning that enabling hyper-threading does not violate SGX's security and hyper-threading on Whiskey Lake is considered to be a safe configuration. Furthermore, the machine has been updated with Intel's latest microcode, which mitigates TAA attacks on SGX by disallowing TSX transactions on logical cores that are co-resident with logical cores running SGX enclaves [21].

A. Reading Enclave Data

The first building block for attacking SGX with CacheOut is to force the victim enclave's data into the L1-D cache.

Loading Secret Data into the Cache. Even though the malicious kernel cannot directly read the contents of the enclave, the kernel is still responsible for paging the victim enclave's pages using the special SGX instructions `ewb` and `eldu`. Foreshadow [46] discovered that by 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 Foreshadow [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 10.

We improved upon this technique by forcing multiple copies of the plaintext corresponding to the victim's page into the cache. 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. Finally, since the `ewb` and `eldu` instructions operate at page granularity, an attacker using these instructions can choose which pages to read from. This gives the attacker 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 10 Steps 1-2), the attacker can mount a CacheOut attack that performs 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. While we chose to demonstrate the attacks in the section against a victim enclave running on a parallel thread, we were also able to observe data leakage in the sequential model. Even with hyper-threading disabled, we can exercise address selection to read enclave data that remains in the L1-D cache after the enclave exits.

Bypassing TAA Countermeasures. We are able to utilize TSX for this attack despite Intel's mitigation for preventing TSX and SGX simultaneously running on the same core [21]. This is likely because the L1 is not being flushed after the enclave finishes running. Afterwards, once TSX is again enabled, the attacker can evict the targeted data from the L1-D into the LFB, and then perform TAA to read the data.

SGX Image Extraction. In order to quantify our leakage from SGX, we set up an SGX enclave that contains a picture of the Mona Lisa, and use CacheOut to leak and reconstruct the picture. As the image we are trying to extract is 128 by 194 pixels, it spans multiple pages. Thus, we use the aforementioned `ewb` and `eldu` technique on each image page individually, and use address selection in order to leak unstructured pixel data from the entire page. We sampled the image data from the SGX enclave five times. For each byte offset, we used 2.5K TAA iterations to sample data, resulting in a run time of 7.75s per cache line and 496s per page on average. In each such run, we observed 36% of the image data on average and an actual throughput of 24.54B/s. We combined the data from our runs observing 71% of the image.

Image Reconstruction. We now reconstruct the picture of the Mona Lisa from our collected data. First, since we have address selection capabilities, we are able to collect all the

Fig. 10: A schematical overview of how the SGX paging mechanism, in combination with TSX Asynchronous Abort, leaks arbitrary SGX data.

candidates for every pixel from our sampled data. Then we calculate a score for each candidate based on the candidates of the neighbouring pixels using a naïve distance function: $(r_1 \cdot r_2)^2 + (g_1 \cdot g_2)^2 + (b_1 \cdot b_2)^2$. Finally, we sort the candidates based on the smallest score first and select the first candidate as the actual pixel to output in the resulting image. The offline phase took 8.39s to reconstruct the image, which can be seen on the right in Figure 11. To reconstruct the the Mona Lisa from our collected data, we first use our address selection capabilities to obtain all the candidates for every pixel from our sampled data. Then we calculate a score for each candidate based on the candidates for neighboring pixels using a distance function: $(r_1 \cdot r_2)^2 + (g_1 \cdot g_2)^2 + (b_1 \cdot b_2)^2$. Finally, we sort the candidates select the candidate with the smallest score as the actual pixel value. The offline phase took 9s to reconstruct the image, which can be seen in Figure 11(right).
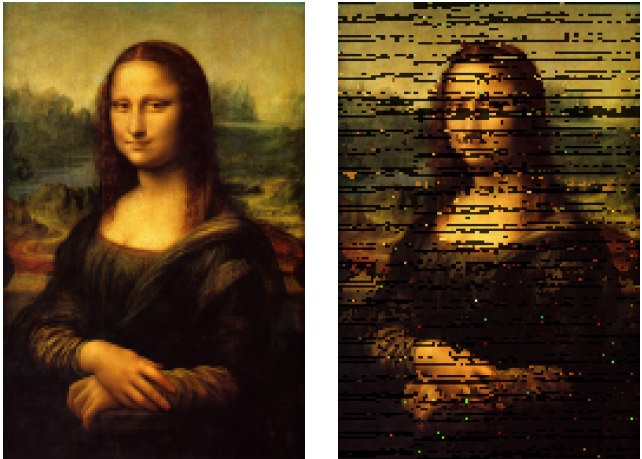

Fig. 11: On the left the original Mona Lisa picture (128x194) and on the right the Mona Lisa picture recovered from an SGX enclave on the Intel Core i7-8665U.
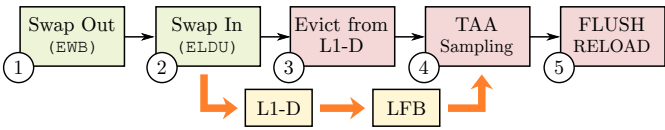
B. Extracting the SGX EPID Key

Trust in the SGX ecosystem is rooted in the *Enhanced Privacy ID* (EPID) key, where compromising a single EPID key breaches the entire SGX ecosystem's security. Thus, this key is available only to enclaves written and signed by Intel. It is stored as a normal file, but encrypted using *seal keys* that are only available to Intel's quoting and provisional enclaves.

EPID Key Extraction in Debug Mode. We begin the process of recovering EPID keys by compiling and self-signing Intel's quoting enclave, running it with debugging EPID keys. We then recovered the sealing key used to seal the file holding the debugging EPID keys and subsequently used it to decrypt the debugging EPID key. To extract the sealing key, we used a controlled-channel attack [51], pausing the quoting enclave after it has loaded the seal key into memory. After this point, the enclave never resumes execution and is permanently stopped. After stopping the enclave with the sealing key in memory, we use the technique from Section VIII to repeatedly swap in and out the page containing the sealing key, extracting

it using CacheOut. This stage takes about 1.5 minutes. Due to noise, we see on average 4.5 candidates per key byte, where the key is 16 bytes. This leaves 747K candidates to brute force. Because the key is sealed using AES-GCM, we can identify the correct key during the offline brute force phase by comparing against the GCM authentication tags. We brute-forced the sealing key in 5 seconds, successfully decrypting the file holding the debugging EPID keys.

Bypassing Software Defenses. We note that the above attack, as well as the attacks in Section VIII, do not require the victim enclave to execute any specific access pattern to the key, or even run at all after loading the key into memory. Thus, CacheOut must be mitigated in hardware, as there is nothing an enclave can do to protect its secrets from being extracted. In particular, our attack bypasses all existing software mitigations for side channels, such as constant-time coding, detecting page-faults [6, 39, 44], and others [12, 42].

Comparison to State-of-the-Art. Our breach of an SGX enclave on this particular machine exemplifies how Cache-Out's advancement over the state of the art in transient-execution attacks enables it to compromise a system that is resistant to previously known attacks. The i7-8665U (Whiskey Lake) contains in-silicon Foreshadow [46] mitigations, which prevent an attacker from directly leaking from the L1-D cache. Fallout [4] cannot target SGX, as the store-buffer is flushed upon swapping to the enclave's page tables. Finally, RIDL [48] and ZombieLoad [43] are mitigated by disallowing TSX and SGX in parallel [21], leaving CacheOut as the only technique for EPID key extraction.

Attacking Production Enclaves. While the above demonstrates the theoretical feasibility of extracting the CPU's EPID key, we did use a version of the quoting enclave which was self-compiled and self-signed. As such, this version is unable to access the machine's actual attestation keys, thus preventing us from extracting them. The reason we made this choice is that at the time of writing, Whiskey Lake machines have an issue with their internal GPU, which allows attackers to leak information from within SGX enclaves [25]. As Whiskey Lake is a laptop architecture, it is impossible to disable the internal GPU, which results in these machines being unable to receive a trusted SGX status and production attestation keys. Being unable to configure the machine in a state trusted by Intel, we have resorted to extracting the EPID sealing key from the quoting enclave that we compiled and signed ourselves, using the official Intel-provided source code.

SGAxe: How SGX Fails in Practice. However, in a follow-up work [49], we demonstrate the breach of the SGX ecosystem by extracting production EPID attestation keys from an older Coffee Lake Refresh based desktop, which we were able to configure to a trusted state using an external GPU. In particular, SGAxe [49] demonstrates the extraction of the machine's production attestation keys on a fully updated and trusted machine, defeating recent side-channel countermeasures such as LVI [47] and PlunderVolt [38]

IX. MITIGATIONS

We now discuss various ways to mitigate CacheOut: disabling hyper-threading, flushing the L1-D cache, disabling TSX and microcode updates by Intel.

Disabling Hyper-Threading. Similar to MDS, CacheOut works best when the attacker and victim run in parallel on two threads on the same physical core. However, as CacheOut is also effective in the scenario without hyper-threading where attacker and victim run on the same CPU thread, disabling hyper-threading makes the attack difficult but not impossible (see Section V, VI, and VII). Finally, as disabling hyper-threading carries a significant performance overhead, we do not recommend this countermeasure for mitigating CacheOut.

Flushing the L1-D cache. As discussed in Section IV-C, CacheOut leaks information from the L1-D cache. Thus, one might attempt to flush the L1-D and LFB on security domain changes, in an attempt to eliminate the source of the signal. Unfortunately, L1-D cache flushing adds significant overhead and only covers the case without hyper-threading, as leaving hyper-threading enabled means that CacheOut will be able to leak data from the L1-D as the victim accesses it. Thus, given the cost of implementing both of these countermeasures, we do not recommend deploying them for mitigating CacheOut.

Disabling TSX on New Hardware. To address TAA [21] on the newest platforms released after Q4 of 2018 (i.e., after Coffee Lake Refresh), 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 the first bit in the MSR disables TSX, and the second bit disables CPUID enumeration for TSX capability. Concurrent to our work and after our disclosure, OS vendors started disabling TSX by default on all Intel machines released after Q4 of 2018. We note that, however, this mitigation is only partial as a malicious operating system can always re-enable TSX and use CacheOut to leak data from SGX enclaves while bypassing Intel's SGX countermeasures for TAA (as we demonstrated in Section VIII). Thus, at present SGX remains vulnerable.

Disabling TSX on Older Hardware. We note however that the vast majority of Intel machines currently deployed were released before Q4 2018. For those machines, Intel started rolling out microcode updates to address CPU errata regarding TSX [26], allowing operating systems to disable TSX by making transactions always abort. However, at the time of writing this behavior is not enabled by default, leaving the majority of deployed Intel CPUs exposed to CacheOut. Given that TSX is not widely used, we recommend to disable TSX by default on these CPUs as well. Finally, we note that TSX must be disabled on the microarchitectural level, including during transient and speculative execution, as opposed to aborting the TSX transaction after speculation has occurred.

SGX Security. As we show in Section VIII, a malicious OS can always re-enable TSX and subsequently use CacheOut in order to dump the enclave's contents. While SGX is insecure at present, we recommend that future microcode updates

15

declare TSX to be unsafe in combination with SGX on current machines, and to flush the L1-D every time TSX is enabled.

Microcode Updates. Intel's security advisory [23] indicates that CacheOut (called L1DES in Intel's terminology) will be mitigated via additional microcode updates. These are expected to be available on June 9th, 2020, with preview versions supplied by Intel indeed showing a successful mitigation of CacheOut. In private communication, Intel further indicated that mitigating the new data path between L1-D evictions and the LFB discovered by this work is done by adjusting internal CPU timing, preventing the leakage exploited by CacheOut. We recommend that affected users install these, especially on older machines that do not disable TSX by default.

X. Conclusion

In this paper, we investigated Intel's use of buffer overwriting to mitigate MDS attacks, and found that we could force the victim's data to re-enter microarchitectural buffers even after their contents were overwritten during a transition between security domains. Using this technique we developed CacheOut, a new transient-execution attack that is capable of breaching Intel's buffer overwrite countermeasures, while allowing the attacker to surgically choose exactly what data to leak from the CPU's L1-D cache. We demonstrated the implications of CacheOut by developing attacks breaching confidentiality across a number of security domains, spanning user space, kernel space, and hypervisors. Furthermore, we also demonstrated that SGX is still insecure, despite the deployment of MDS countermeasures. Finally, CacheOut is able to leak data on Intel's Whiskey Lake CPUs, which are resilient to prior MDS attacks.

Limitations. While we clearly demonstrated the feasibility of CacheOut using TSX, we were unable to perform CacheOut using other transient-execution attack primitives (e.g., mispredicted branches). While we acknowledge this limitation, we note that TSX is still enabled on all Intel machines released prior to Q4 2018 and can be re-enabled by a malicious OS in the case of SGX. Next, the signal for the cross-process, VM and kernel variants of CacheOut is noisy, requiring multiple attack iterations for data extraction. Thus, we leave it to future work to demonstrate CacheOut-type leakage without TSX, as well as improving that attack's signal-to-noise ratio. Finally, CacheOut is only able to leak data located inside the CPU's L1-D cache, leaving other levels of the memory hierarchy out of reach. As L3 caches are often shared between physical cores, exploring techniques for reading L3-data is an important research problem with many immediate security implications.

XI. Acknowledgments

References

[1] H. Akkary, J. M. Abramson, A. F. Glew, G. J. Hinton, K. G. Konigsfeld, P. D. Madland, M. S. Joshi, and B. 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.

[2] ——, "Cache memory system having data and tag arrays and multi-purpose buffer assembly with multiple line buffers," US Patent 5,680,572, Jul 1996.

[3] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," in *CCS*, 2019.

[4] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-resistant CPUs," in *CCS*, 2019.

[5] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security*, 2019, pp. 249–266.

[6] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races," in *IEEE SP*, 2018, pp. 178–194.

[7] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T.-H. Lai, "SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution," in *Euro S&P*, 2019, pp. 142–157.

[8] I. Cooperation, "Copying accelerated video decode frame buffers," 2009. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/copying-accelerated-video-decode-frame-buffers.html

[9] D. Coppersmith, "Small solutions to polynomial equations, and low exponent rsa vulnerabilities," *Journal of cryptology*, vol. 10, no. 4, pp. 233–260, 1997.

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

[11] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security*, 1998.

[12] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin, "SGX-LAPD: Thwarting controlled side channel attacks via enclave verifiable page faults," in *RAID*. Springer, 2017, pp. 357–380.

[13] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead: Long live KASLR," in *International Symposium on Engineering Secure Software and Systems*, 2017, pp. 161–176.

[14] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on AES to practice," in *IEEE SP*, 2011, pp. 490–505.

[15] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[16] N. Heninger and H. Shacham, "Reconstructing RSA private keys from random key bits," in *CRYPTO*, Aug. 2009, pp. 1–17.

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

[18] 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.

[19] ——, "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] ——, "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.

[21] ——, "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.

[22] ——, "Intel 64 and IA-32 architectures software developer's manual," 2016.

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

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

[25] ——, "2019.2 IPU – Intel SGX with Intel processor graphics update advisory," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00219.html, Nov 2019.

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

[27] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES," in *IEEE SP*, 2015.

[28] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "SPOILER: Speculative load hazards boost Rowhammer and cache attacks," in *USENIX Security*, 2019, pp. 621–637.

[29] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *DAC*, 2016.

[30] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[31] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE SP*, 2019.

[32] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *WOOT*, 2018.

[33] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018.

[34] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE SP*, 2015.

[35] A. Lutas and D. 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.

[36] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *CCS*, 2018, pp. 2109–2122.

[37] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-medusa

[38] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against Intel SGX," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[39] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *USENIX ATC*, 2018, pp. 227–240.

[40] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *CT-RSA*, 2006.

[41] C. Percival, "Cache missing for fun and profit," 2005.

[42] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx." *IACR Cryptology ePrint Archive*, vol. 2017, p. 549, 2017.

[43] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[44] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs." in *NDSS*, 2017.

[45] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels," *arXiv preprint arXiv:1806.07480*, 2018.

[46] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin,

B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018.

[47] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[48] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Rogue in-flight data load," in *IEEE SP*, 2019.

[49] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How sgx fails in practice," https://cacheoutattack.com/, 2020.

[50] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," https://foreshadowattack.eu/foreshadow-NG.pdf, 2018.

[51] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE SP*, 2015, pp. 640–656.

[52] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.

[53] J. Zhang, Z. Gu, J. Jang, H. Wu, M. P. Stoecklin, H. Huang, and I. Molloy, "Protecting intellectual property of deep neural networks with watermarking," in *AsiaCCS*, 2018, pp. 159–172.

APPENDIX A
TSX ASYNCHRONOUS ABORT

Listing 2 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 5–6. 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.

APPENDIX B
RECOVERING P AND Q

With all the chunks making up p and q successfully recovered, the next challenge is to reconstruct p and q such that $N = p \cdot q$. We assume that the attacker knows the modulus N, which is part of the public key. Then, as observed by Heninger and Shacham [16], $N = p \cdot q$ implies that the low k bits of N are equal to the low k bits of $p \cdot q$. In order to reconstruct p and q, we iteratively recover the the primes 8 bytes at a time as follows, starting from the LSB.

We first iterate over all possible pairs of the 8 byte chunks, and for each pair (p_0, q_0) compute $n_0 \leftarrow p_0 \cdot q_0$. If the low 8 bytes of n_0 match the least significant 8 byte chunk of N, then p_0 and q_0 are the least significant bytes of p and q. To find the

```asm
; %rdi = leak source
; %rsi = FLUSH + RELOAD channel
taa_sample:
    ; Cause TSX to abort asynchronously.
    clflush (%rdi)
    clflush (%rsi)

    ; Leak a single byte.
    xbegin abort
    movq (%rdi), %rax
    shl $12, %rax
    andq $0xff000, %rax
    movq (%rax, %rsi), %rax
    xend
abort:
    retq
```

Listing 2: the leak primitive using TSX Asynchronous Abort

second least significant 8 byte chunks, we again iterate over all pairs and for each pair (p_1, q_1) compute $n_1 \leftarrow (p_1 \| p_0) \cdot (q_1 \| q_0)$, where $\|$ denotes appending 8 byte chunks. If the least significant two bytes of n_1 are equal to the two low bytes of N, then p_1 and q_1 are the 2nd least significant bytes of p and q. By repeating in this manner for each 8 byte chunk, we can fully recover both p and q.

APPENDIX C
ANN WEIGHT RECOVERY

Weight Filtering. We improve the accuracy of our weight stealing attack by exploiting both the weights' storage format and the observation that the weights of a neural network tend to be small (typically within the range [-1,1]). The weights are small due to machine learning algorithms using regularization during the training phase, which pushes the weights towards zero in order to prevent both overfitting of the model and the gradient explosion problem, which results in untrainable neural networks.

The weights are stored as 32-bit single-precision floating points, which are specified by the IEEE 754 single-precision floating-point standard to use bit 31 for the sign bit, bits 23-30 for the exponent with a bias of -127, and the remaining 23 bits for the mantissa. A small value implies that the exponent field will be very near to 127, and despite the 24 bits of precision, this format means that the smallness of the weights result in a very limited set of values for the most significant byte of each weight. In practice, we find that the MSB does not deviate from 0x40 or 0xc0 by more than 3 for positive and negative weights, respectively. By rejecting all candidates for weights that do not fit, we improve the accuracy to 93%.

We further improve the accuracy by observing that the distribution of the frequency of different bytes of noise produced by CacheOut is not uniform. In particular, the values 0x00 and 0xff appear with a far higher frequency than all others. As such, by penalizing the scores for recovered values that contain 0x00 or 0xff, we improve the accuracy to 96.1%.