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 [29] and Meltdown [31] left a lasting impact on the design of modern processors. Speculative and out-of-order execution, which were previously considered to be harmless and important CPU performance features, were discovered to have severe and dangerous security implications. Since their original discovery, many *transient-execution attacks* have emerged, violating numerous security domains, such as Intel's Software Guard Extension (SGX) [46], virtual machine boundaries [49], AES hardware accelerators [45] and others [3, 6, 8, 15, 25, 28, 29, 30, 34, 35].

More recently, the security community uncovered a deeper source of leakage: internal and mostly undocumented CPU buffers. With the advent of RIDL, Fallout and ZombieLoad [5, 43, 48], it was discovered that an attacker can use an assisting or faulting load to sample data as it appears in the buffer, siphoning off information as the victim accesses it. Dubbed Microarchitectural Data Sampling (MDS) by Intel, these attacks were proven capable of bypassing all previous countermeasures against transient-execution attacks, again breaking nearly all hardware-backed security domains.

*Work partially done while affiliated with Vrije Universiteit Amsterdam.

Luckily, MDS attacks seem limited by two factors. First, the attacker can only leak recently accessed data and only for as long as the data remains in small internal CPU buffers. Secondly, even if the data is still in the buffer, the attacker has limited control over which part of the data leaks, requiring the combination of structure in the leaked data (e.g., a known prefix or suffix) and signal processing techniques to organize the information leaked from the victim's address space. Tackling these limitations, in this paper we ask the following question:

Can an attacker overcome the control and availability limitations of MDS attacks? In particular, can the attacker target specific data from the victim's address, present in larger more persistent structure such as the L1-D cache?

Next, to protect against MDS attacks, Intel deployed countermeasures for blocking leakage from internal CPU buffers. For older hardware, Intel augmented `verw`, a legacy x86 instruction, to overwrite the contents of the leaking buffers upon security domain changes. In parallel, Intel launched the new Whiskey Lake architecture, which is designed to be MDS-resistant [5, 43, 48]. The intuition behind the buffer overwrite countermeasure is that it removes any remaining sensitive data from the buffer. However, previous works [43, Section 7] already report observing residual leakage of 0.1 B/s despite buffer overwriting. Thus, in this paper we ask the following secondary question:

Are buffer overwrites sufficient to block MDS-type leakage?

A. Our Contributions

We present CacheOut, a new transient-execution attack which is capable of alleviating many of the challenges limiting previous MDS works. More specifically, unlike MDS attacks that are limited to leaking in-flight data present in the 12 entries of the CPU's Line Fill Buffer (LFB), CacheOut is able to leak data present in the CPU's entire L1-D cache (32 KiB). Next, CacheOut is not limited to leaking data that is recently accessed by the victim, but instead allows the attacker to leak arbitrary data from the L1-D cache, while controlling what to leak from the victim's address space. Finally, we show that CacheOut is applicable to nearly all hardware-backed secu-

rity domains, including process and kernel isolation, virtual machine boundaries, and SGX enclaves.

A New Leakage Path. The microarchitectural vulnerability behind CacheOut stems from an undocumented interaction between the LFB and the L1-D cache present on Intel machines. More specifically, on modern CPUs, the LFBs 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 cache, 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. Moreover, we show that this path exists even despite recent microarchitectural changes done on Whiskey Lake machines, aimed at mitigating MDS-related issues.

Exploiting Evictions. Next, we note that Intel's MDS countermeasures, both in software and in hardware, do not stop the LFB from leaking. Instead, Intel's approach sanitizes the contents of the MDS-affected buffers, preventing the attacker from recovering information across security boundaries. Thus, by first evicting data from the L1-D cache to the LFB, we are able to subsequently use a faulting or assisting load to retrieve the victim's evicted data from the still leaky LFB. In particular, our leakage path allows CacheOut to bypass Intel's `verw` countermeasures, as the transfer of evicted data from the L1-D cache to the LFB occurs well after the context switch and completion of the associated `verw` instruction.

Controlling What Cache Set To Leak. Our technique of forcing L1 eviction also allows us to lift another restriction of MDS attacks, namely the attacker's lack of control over which data to read from the victim's address space. Specifically, the attacker can force contention on a specific cache set, causing eviction of the corresponding victim data from this cache set into the LFB. This allows the attacker to focus on leaking from a specific cache set, rather than opportunistically siphoning information as it transits through the LFB.

Leaking Specific Bytes. A final limitation of previous MDS works is the ability of the attacker to leak specific bytes of interest from the victim's address space. More specifically, previous works like TSX Asynchronous Abort (TAA) [16] are limited to leaking at most 8 bytes of every 64-byte cache line, leaving the other 56 bytes out of reach. Tackling this issue, we discovered a new feature in Intel's LFB implementation, which seems to have a *read offset* mechanism that controls the position within the LFB from which a load instruction reads. Surprisingly, we observe that some faulting or assisting loads can use stale offsets from *subsequent* load instructions which are executed *antecedently* and out-of-order, before the faulting loads themselves. Using this mechanism, we construct TAA-NG, a new variant of TAA [16] which is capable of leaking arbitrary attacker-controlled byte-offsets from within the LFB.

Reading Entire Victim Pages. Combined with cache evictions, TAA-NG allows us to control the 12 least-significant bits of the address of the data we leak. 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, recovering 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. In particular, CacheOut can recover random-looking data (e.g., cryptographic keys) from the victim's address space, thereby lifting the known-prefix/suffix requirement of previous works [43, 48].

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.

Attacking Process and Virtual Machine Isolation. We show that CacheOut has severe implications for process-to-process and VM-to-VM isolation, as it allows an unprivileged attacker to leak information across these boundaries, thereby breaching their confidentiality. We demonstrate this risk by implementing attacks on private data across processes in different security domains. Targeting OpenSSL's AES and RSA operations, we successfully recover secret keys and plaintext data. Finally, we also demonstrate CacheOut's capability of leaking secret weights from an artificial neural network.

Attacking the Kernel Hypervisor. We also demonstrate highly practical attacks against the Linux kernel and virtual machine hypervisor, without requiring elevated permissions. By taking advantage of CacheOut's cache line selection capabilities, we are able to completely derandomize Kernel Address Space Layout Randomization (KASLR) and Hypervisor ASLR in under two seconds. Furthermore, we demonstrate extraction of stack canaries from the kernel.

Attacking Software Guard Extensions (SGX). We show that CacheOut can attack data at rest, dumping the contents of enclaves without requiring the victim enclave to perform any operation, or even execute at all, after placing the secret into memory. We emperically demonstrate this through the recovery of images from enclaves, and the extraction of the production EPID attestation keys. These attacks are performed on fully updated Whiskey and Coffee Lake CPUs, which are resistant to previous MDS attacks, including ZombieLoad, RIDL [43, 48], and TAA [16]. With the attestation keys, we are able to pass fake enclaves as genuine, issue fake attestation quotes, or even allow AMD / ARM machines to masquerade as genuine Intel hardware.

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

- We present CacheOut, a new transient-execution attack that can leak across arbitrary address spaces while still retaining fine-grained control over what data to leak.
- We demonstrate the effectiveness of CacheOut in violating process and VM isolation by recovering AES and RSA keys and plaintexts from an OpenSSL-based victim.

- We demonstrate practical exploits for derandomizing kernel and hypervisor ASLR, and for recovering secret stack canaries from the Linux kernel.
- 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.

B. Current Status and Disclosure

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. Finally, dedicated microcode updates mitigating the root cause behind CacheOut were published on June 9, 2020. We recommend installing these on affected Intel platforms.

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 that the program uses. A multi-core processor typically has one L1-D cache in each physical core.

Cache Organization. Caches generally consist of multiple cache sets that can host up to a certain number of cache lines or *ways*. Part of the virtual or physical address of a cache line maps that cache line to its respective cache set, where the *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 virtual 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 [12, 51] and Prime+Probe [24, 27, 32, 40, 41].

B. Microarchitectural Buffers

In addition to caches, modern processors contain multiple microarchitectural buffers that are used for storing in-transit data. In this work we are mainly interested in the *Line Fill Buffers*, depicted in Figure 1, which handle data transfers between the L1-D cache, the L2 cache, and the core. 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 serves 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.

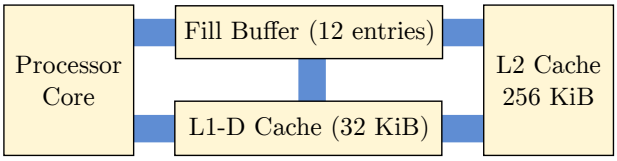

Fig. 1: Connections between the CPU core, caches, and LFB

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 a transient instruction has been executed incorrectly, it discards all of the instruction's results, rather then committing them to the architectural state.

D. Speculative-Execution Attacks

Because transient instructions are not part of the nominal program order, they may sometimes process data that is not accessible in nominal program order. In recent years, multiple *speculative-execution attacks* have demonstrated the possibility of leaking such data [6, 8, 15, 28, 29, 30, 35]. 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 speculative-execution attacks use a Flush+Reload-based covert channel for sending the data. In a typical attack, the attacker maintains a *probing* array consisting of 256 distinct cache lines. The attacker flushes all of these cache lines from the cache before triggering 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. Transactional Memory

Introduced in Intel's 4th generation cores (Haswell), the Transactional Synchronization Extensions (TSX) lets programmers start and end transactions using the `xbegin` and `xend` instructions (respectively). To guarantee that transactions become either globally visible or not at all, the CPU executes the transaction speculatively until the CPU reaches the `xend` instruction, commiting the results, or when a transactional abort occurs, discarding the results.

TSX Asynchronous Abort. TSX Asynchronous Abort (TAA) is a transient-execution attack that allows an attacker to sample data from the LFB. The primitive leaks this data by reading the data from a memory address for which a `clflush` instruction is in flight while executing the transaction. This ultimately causes a cache line conflict which results in a transactional abort. However, as the code within the transaction runs speculatively, it is possible for an attacker to sample data from the LFB and leak it through a FLUSH+RELOAD side channel. See Appendix A for a TAA code example.

TAA Limitations. While TAA is a powerful primitive for leaking LFB contents, it is limited to only leaking the in-flight data bytes used by the victim's load and store operations [16, 43, 48]. As on Intel CPUs the victim can only access at most 8 bytes in a single load or store operation, this upperbounds TAA attacks to leaking at most 8 bytes of every 64-byte cache line, leaving the other 56 bytes, and any cache lines not accessed by the victim out of reach. In Section IV-D, we improve this limitation by presenting a variant of TAA, called TAA-NG, that can leak the entire 64-byte cache line regardless of any accesses performed by the victim.

III. Threat Model and Hardware Setups

Threat Model. We assume that the attacker is an unprivileged user, such as a VM or unprivileged user process on the victim's system. For the victim, we assume an Intel-based system that has been fully patched against Meltdown [31], Foreshadow [46, 49], and MDS [5, 43, 48] 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 is present and enabled and that the attacker can run on the same physical processor core as the victim. At the time of writing, TSX is indeed enabled by default on all Intel machines launched prior to 2018-Q4.

Finally, when attacking SGX, we assume a malicious operating system capable of configuring the CPU and scheduler as chosen by the attacker. This includes re-enabling TSX (if disabled), stopping and starting enclaves, and making arbitrary calls to SGX's paging instructions as needed.

Hardware Setup. We have tested CacheOut on multiple generations of Intel processors, ranging from Skylake (2016) to Whiskey Lake (launched 2019-Q2). However, since the discovery of transient-execution attacks [5, 6, 16, 29, 31, 46, 48], Intel has launched several generations of processors, each containing an increasing amount of hardware-based countermeasures. See Table I for a summary.

We now proceed to describe MDS-specific software and hardware countermeasures present on Intel machines.

Flushing MDS Buffers. Fallout [5], RIDL [48], and ZombieLoad [43] show that attackers can leak data transiting through various internal microarchitectural buffers, such as the LFBs discussed in Section II-B. To address these issues for older hardware, Intel provided microcode updates [21] that repurposed 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.

Disabling TSX. In November 2019, during the course of our work and after our disclosure of CacheOut, Intel attempted to mitigate TAA by publishing microcode updates that enable turning off TSX on CPUs made after 2018-Q4. These have been deployed by OS vendors, preventing non-SGX variants of CacheOut on Intel machines made after 2018-Q4.

We note however, that all variants of CacheOut (e.g., cross-process, -kernel and -VM) are still applicable on Intel machines launched before 2018-Q4, which represent the majority of deployed hardware at the time of writing. Moreover, as TSX can still be re-enabled by a malicious OS, CacheOut's ability to access data at rest allows us to bypass Intel's SGX $\oplus$ TSX policy, thereby leaking content from SGX enclaves even on fully updated machines with all countermeasures in place. In particular, as Whiskey Lake machines are only vulnerable to TAA, which is limited to leaking LFB data directly accessed by the victim, CacheOut is the only attack currently capable of leaking the contents of the L1-D cache on these machines.

IV. The CacheOut Attack

In this section we present CacheOut, a transient-execution attack that allows an attacker to read the victim's data from the L1-D cache through cache evictions. We describe two variants: the first targets data modified by the victim's write operations, while the second aims at data that the victim reads but does not modify. Besides defeating all previously deployed MDS mitigations both in software and hardware, CacheOut addresses three challenges left open by previous MDS works.

[C_1] Controlling What to Leak. How can an attacker control what to leak from the victim's address space?

[C_2] Reading the Entire Cache Line. How can an attacker read all the data present in the victim's cache line, including data not affected by the victim's load or store operations?

[C_3] Temporal Persistence. How can an attacker read the victim's data long after the execution of the victim's load or store operation, as opposed to being limited to in-flight data briefly present in the CPU's LFB?

CacheOut Overview. Listing 1 presents a high-level overview of CacheOut. Assume that the victim performed a write operation to some address `a`, which the attacker would like to read (Line 2). After finding the correct eviction set which evicts `a` from the L1-D cache, the attacker accesses the eviction set, causing the data written by the victim to be evicted from the CPU's L1-D cache (Lines 5-7) into the LFB.

CPU	Year	CPUID	Meltdown	Foreshadow	MDS	TAA	CacheOut
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	🐧	🐧	🐧	🐧	✓

TABLE I: Vulnerability of Intel processors to transient-execution attacks. ✓ indicates that the machine is vulnerable. 🐧 indicates that vulnerability is mitigated by a microcode update and/or the OS, while ⚙ indicates that the vulnerability is mitigated in silicon. Finally, ✓/ 🐧 indicates that the mitigations are incomplete, as the attack is possible against SGX, or if TSX is enabled.

```
1  victim:
2  a = secret
3
4  attacker:
5  for (i = 0; i < 8; ++i) //evict secret
   ↪  from L1D cache
6    *(evict_set + 4096 * i) = 0;
7
8  TAA-NG(FRbuffer);
9
10 for (i = 0; i < 256; ++i)
11   if (flush_reload(FRbuffer + i * 4096))
12       ++results[i];
```

Listing 1: high-level overview of the CacheOut attack.

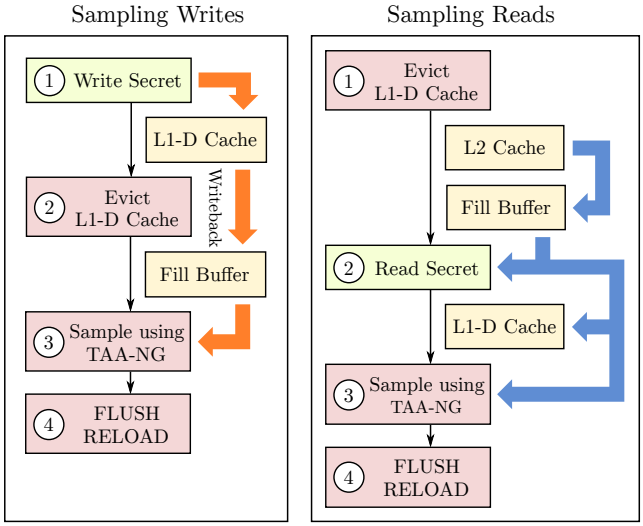

Fig. 2: Overview of our use TAA-NG for leaking LFB data. Victim activity, attacker activity, and microarchitectural effects are shown in green, red, and yellow respectively. Victim and attacker may run on the same or sibling hyper-threads.

The attacker then samples the data from the LFB using our TAA-NG sampling primitive (Line 9), leaking the results using a FLUSH+RELOAD side channel. Finally, the attacker checks the FLUSH+RELOAD channel by measuring the access time to every entry of the `FRbuffer`, thereby recovering the victim's data (Lines 11-14). See Figure 2 (left) for an overview.

Overcoming Challenges. We now describe how each component of CacheOut allows us to overcome limitations of previous MDS and TAA techniques. First, by constructing eviction sets that precisely evict only certain cache lines, we alleviate challenge $[C_1]$, allowing the attacker to control what cache line to leak from the victim's address space. Next, as TAA is limited to only leaking bytes directly accessed by the victim (and at most 8 bytes in total from every 64B cache line [43]), in Section IV-D we present TAA-NG, a new variant of TAA which is capable of leaking arbitrary byte offsets from the LFB. This overcomes challenge $[C_2]$, allowing the attacker to read arbitrary bytes of their choice from the victim's address space. Finally, as we show in Section IV-E, data present in the L1-D cache remains cached for millions of cycles. Thus, by exploiting L1-D evictions rather than opportunistic LFB sampling, we overcome challenge $[C_3]$, giving the attacker the capability of selecting and leaking any L1-D data written by the victim, as opposed to being limited to in-flight data present in the LFB following the victim's memory operations.

Attacking Reads. The attack described so far is capable of extracting data written by the victim into its address space. However, for data that is only read by the victim but never modified, we note that in case of an L1-D miss, the data must also traverse through the LFB in order to satisfy the victim's read operation. Thus, by evicting the victim's data beforehand, we are able to attack the victim's read operations, selecting which data to leak from the victim's address space. The right part of Figure 2 shows a variant of CacheOut for attacking victim read operations. As before, we assume that the attacker has already constructed an eviction set for the cache set that contains the victim's data. Unlike the attack for writes however, which can be executed both with and without hyper-threading, here we assume that the attacker and the victim run on the two hyper-threads of the same physical core.

The attacker starts by reading from all of the addresses in the eviction set (Step 1). This loads the eviction set into the L1-D cache to evict the victim's data. Next, the attacker waits for the victim to access their 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-D cache. Finally, the attacker uses TAA-NG (Step 3) to sample values from the line fill buffer and transmit them via a FLUSH+RELOAD channel (Step 4).

Microbenchmarking. Before discussing the attacks in Sections V, VI, and VII, we proceed to benchmark individual components required to mount CacheOut in the remainder

of this section. Furthermore, we discuss CacheOut's leakage source and potential signal improvements.

A. Measuring Cache Eviction

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

Measuring Eviction. To confirm that we leak data from the L1-D cache through evictions, we first measure the number of accesses required to evict the victim's cache line out of the cache. We run a victim that repeatedly accesses the same cache set, while we test CacheOut with varying eviction set sizes under three different attack scenarios. Figure 3 shows the result of the three scenarios: on the same hyper-thread (left), victim reads across hyper-threads (middle) and victim writes across hyper-threads (right). As expected, an eviction set of eight addresses yields the best result, except for victim writes where a set of six addresses yields the best result. Our attack likely requires fewer evictions due to the increased L1-D cache contention while both hyper-threads are active. Moreover, further increasing the eviction set has a negative impact, as larger eviction sets result in a longer execution time, reducing the frequency at which the attack can sample data.

Conclusion: By using cache evictions, we can reliably leak data present in the CPU's L1D cache.

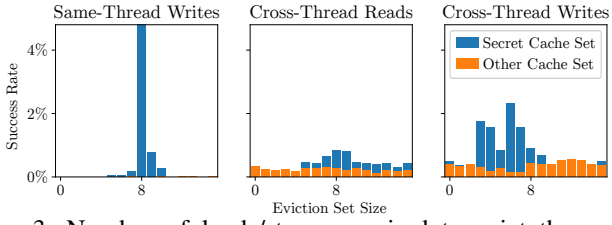

Fig. 3: Number of loads/stores required to evict the secret cache line from the victim. The blue bars indicate how often we observe the secret from the correct cache line, while the orange bars indicate how often we observe it from the strongest wrong cache line. We ran 10,000 iterations per tested value.

B. Investigating CacheOut's Leakage Source

We now investigate CacheOut's microarchitectural leakage source and the leakage's path through internal CPU structures.

Flushing Methodology. To ascertain CacheOut's source of leakage, we use the instructions Intel provided to flush these caches and MDS-affected buffers in order to address previous vulnerabilities (e.g., MDS and Foreshadow), and then try to mount CacheOut to see whether these buffers affect the actual leakage. In all experiments, we run the victim and attacker on the same hyper-thread, where the victim first writes some data to different cache lines, followed by an `mfence` to ensure that these stores have been committed. We then perform some flushing operation, followed by an explicit `lfence` to ensure that the flushing completes before executing any other code, with the exception of the base case where no flushing is performed. Finally, we execute the CacheOut attacker, which attempts to leak the victim's data from a specific cache line. Figure 4 shows a summary of our findings.

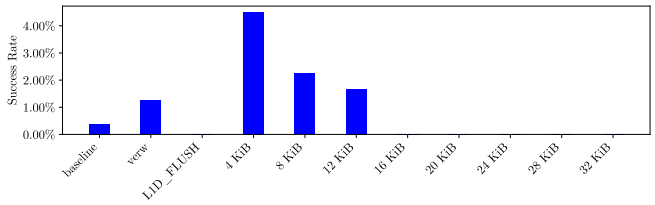

Fig. 4: CacheOut's success rate when using different flushing operations between attacker and victim.

Ineffective `verw`. We observe that rather than mitigating CacheOut, the `verw` instruction actually improves CacheOut's leakage rate. While `verw` did initially not fully clear the affected buffers [42], we have performed our experiment on an Intel Core i7-8665U (Whiskey Lake) CPU running the November 2019 microcode update, which mitigates this issue. As `verw` has been repurposed to mitigate MDS attacks by overwriting the contents of MDS-affected buffers, we deduce that CacheOut does not directly leak from the LFB.

Effective L1-D Flushing. Next, we use the `MSR_IA32_FLUSH_CMD` countermeasure, originally designed to mitigate the Foreshadow attack, in an attempt to flush the contents of the L1-D cache. Figure 4 shows that this does fully mitigate CacheOut by completely eliminating the leakage signal. As the contents of the L2 and L3 cache were not flushed, we deduce that CacheOut leaks data from the L1-D cache, but is unable to access data stored in higher-level caches.

Partial L1-D Flushing. Finally, we experiment with partial L1-D flushing by performing `verw` to flush MDS buffers and subsequently accessing a contiguous buffer of 4 KiB - 32 KiB not cached in the L1-D cache. Accessing a buffer of at least 16 KiB eliminates the leakage signal, whereas accessing 4-12 KiB buffers before executing the CacheOut attacker appears to actually improve the leakage signal considerably compared to the baseline measurement. We conjecture that this is due to the partitioning of the L1-D cache for cache line allocations while both hyper-threads are active, as the L1-D cache is indeed competitively shared between hyper-threads [18]. We further investigate this effect in Appendix C.

A New Data Path. While on Intel CPUs the LFB is responsible for handling data coming into the L1-D cache, we conclude that there exists an undocumented path between L1-D evictions and the LFB (marked in red in Figure 5). CacheOut exploits this path by causing L1-D evictions and subsequently leaking the evicted data from the LFB. Finally, we demonstrate that this undocumented path exists on MDS-resistant Whiskey Lake machines, making these vulnerable to CacheOut.

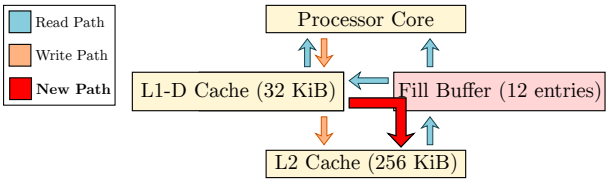

Fig. 5: The data paths within the CPU core, with the paths for loads marked in blue, the path for stores in orange, and the new undocumented path that we uncovered marked in red.

Conclusion: CacheOut leaks data present in the CPU's L1-D cache, exploiting L1-D cache evictions to move data into the LFB through a previously undocumented data path.

C. Overcoming [C_1]: Selecting Cache Lines

Having established CacheOut's source of leakage, we now demonstrate CacheOut's ability to select the L1-D cache set from which the attacker leaks, giving us control over the page offset at a granularity of cache lines (64 bytes). This allows us to overcome previous limitations of MDS attacks where the attacker could only target in-flight data, rather than target data from a particular L1-D cache set. To that aim, we repeat the experiments of Figure 3 but this time we vary the "secret" cache set the victim chooses, while the attacker tries to evict and leak from every possible cache set. Figure 6 contains a summary of our findings for victim reads and writes in both single-threaded and cross-threaded scenarios. As can be seen, in all scenarios the attacker can target a specific cache set, correctly leaking the victim's data, albeit with some noise for the case of cross-thread victim writes.

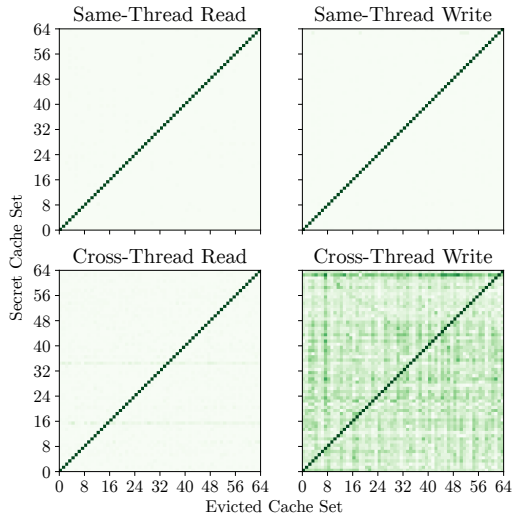

Fig. 6: The victim loads/stores a secret to every possible cache line (y-axis), while the attacker evicts every possible cache line (x-axis) to leak it. We ran 10K iterations per test.

Conclusion: By constructing an eviction set for a particular L1-D cache set, CacheOut can leak from that particular cache set, giving the attacker control over what cache set to leak from the victim's address space.

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

Listing 2: the TAA-NG leak primitive.

D. Overcoming [C_2]: Using TAA-NG for Reading Entire Cache Lines

We proceed to now overcome another limitation of TAA, RIDL and Zombieload [16, 43, 48], where the attacker is limited to leaking only bytes directly accessed by the victim's load or store operations. As on Intel machine a load or store operation can access at most 8 bytes simultaneously, the attacker can recover at most 8 bytes from the 64 byte cache line, leaving the remaining 56 bytes inaccessible.

TAA-NG Overview. We found that by extending the TAA primitive, we can target entire cache lines. More specifically, Listing 2 shows our leakage primitive, which we call TAA-NG. The code is virtually identical to the code in Appendix A, with the exception of two critical `movq` instructions added at Lines 16–17. Like the original TAA, the attacker starts by flushing a cache line in their own address space (Line 6 in Listing 2). The attacker then initiates a TSX transaction (Line 10) which attempts to read the cache line flushed at Line 6.

Next, during the speculative execution of the load instruction (`movq`) at Line 11, much like the original TAA attack, the processor allocates an LFB entry. Similarly to the original TAA, as the previous `clflush` makes the data of the load instruction unavailable, the transaction proceeds speculatively with stale data present in the LFB entry from a previous memory access being served to the `rax` register at Line 11.

However, while in the original TAA the bytes accessed by the victim appear in the `rax` register (leaving the other bytes out of reach), here we discovered that reading from the exact attacker-chosen byte offset into the `rax` register at Lines 16-17 gives us precise control over the byte offset of the victim's data appearing in `rax`. Finally, similarly to the original TAA primitive, TAA-NG proceeds to leak the recovered information through a FLUSH+RELOAD side channel (Lines 12-14).

TAA-NG Microarchitectural Root Cause. We note that the instructions controlling the attacker's read offset (Lines 16-17 in Listing 2) appear after the instruction that actually reads the victim's data (Line 11 in Listing 2). While it is odd that

later `movq` instructions can affect the outcomes of earlier instructions, we observe that the there is no data dependency between the offset-control instructions at Lines 16-17 with the data-reading instruction at Line 11. Consequently, due to out-of-order execution, the instructions at Lines 16-17 can execute *before* the instructions that precede them in program order (i.e., before Line 11). We hypothesize that the line fill buffer output has a *read offset*, i.e., some internal state that determines the offset at which buffer entries have been read. This read offset gets reused by the leaking `movq` at Line 11 when the transaction aborts, allowing us to select the desired cache line offset. However, as we were unable to find any documentation of these mechanisms in Intel manuals and patents, we leave the task of precisely understanding this internal mechanism to future work. Finally, to the best of our knowledge, we are the first paper to demonstrate how latter instructions in program order can affect the speculative output of former instructions.

Evaluating Offset Selection. To evaluate our cache-line offset selection method, we used a victim that chooses a byte offset and writes a secret value to this byte, setting the rest of the bytes in the cache line to zero. The attacker then tries to leak the secret from every byte offset from the victim's cache line. As we can see in Figure 7, we can successfully select the offset in the cache line from which we leak.

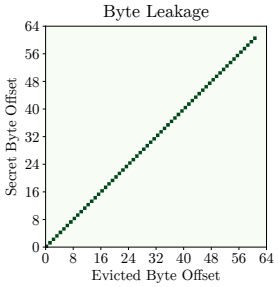

Fig. 7: The victim loads/stores a secret byte to every possible offset within a fixed cache line (y-axis), while the attacker tries to leak from every possible byte offset (x-axis) to leak it.

Conclusion: Our TAA-NG primitive is capable of leaking all 64 bytes in a cache line. Combining this with the L1-D cache set eviction described in Section IV-C, CacheOut has the capability of leaking the precise byte located in the L1-D cache that the attacker would like to leak.

E. Overcoming [C_3]*: Obtaining Temporal Persistence*

Having demonstrated that CacheOut can target data by matching the 12 least-significant bits of the victim's address, we tackle the final limitation of only being able to target in-flight data and instead show that CacheOut is able to leak data that is at rest in the L1-D cache. To benchmark CacheOut's ability to recover data from the L1-D cache far after the victim's access, we perform an experiment where after the victim's access, we delay the attacker's execution for a fixed amount of CPU cycles before mounting CacheOut sequentially after the victim on the same hyper-thread.

Figure 8 shows that we can recover the victim's data even when waiting for 1,000,000 cycles after the victim's access.

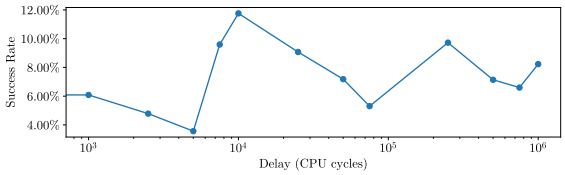

Fig. 8: The amount of leakage when waiting for a given number of CPU cycles before running the attacker.

Rather than targeting the data of the victim's memory access directly, CacheOut evicts the data from the L1-D cache into the LFB to then leak it, which allows CacheOut to leak data long after the victim's execution as long as that data remains present in the L1-D cache. Finally, the varying success rate of Figure 8 may be due to poor synchronization between attacker and victim, as we may end up sampling the LFB before the L1-D eviction or after the data has left the LFB.

Conclusion: CacheOut is not limited to leaking data during the victim's memory operations, but can leak any byte present in the L1-D, long after the victim's memory access.

F. Negative Result: The Need for TSX

We have also explored the possibility of mounting CacheOut without the need for TSX, replacing aborting TSX transactions with other fault-generating mechanisms such as dereferencing `null` pointers or accessing non-canonical addresses. While these were used in previous MDS works [5, 36, 43, 48] for sampling internal CPU buffers, we were unable to use these techniques for mounting CacheOut on MDS-mitigated machines (e.g., Whiskey Lake). Attempting these experiments on older MDS-vulnerable hardware, we were able to correctly identify LFB leakage, but without the ability to leak from the L1-D cache, or control what data is being leaked. As the hardware is MDS-vulnerable, we attribute this leakage to an MDS-like sampling similar to RIDL [48] as opposed to CacheOut's successful exploit of an L1-D evictions.

Conclusion: CacheOut requires TSX to leak data from the L1-D cache. We leave the task of finding a variant of CacheOut which does not rely on TSX as an open problem.

V. Cross-Process and Cross-VM Attacks

Demonstrating the implications of CacheOut, we developed multiple proof-of-concept attacks wherein an unprivileged user process leaks data from another process. First, we estimate CacheOut's leakage rate in the best possible scenario, where a cooperating transmitter attempts to exfiltrate data to a cooperating receiver. We then show more realistic attacks, where an unprivileged attacker recovers AES keys, RSA keys, and the weights of a neural network from another victim process. Finally, we show that CacheOut also extends to the cross-VM setting, allowing data recovery across VM boundaries.

Leaking Unstructured Data. In all the examples, we demonstrate how CacheOut's capabilities allow an attacker to target specific victim data in the L1-D cache. As the attacker knows which data is being targeted, this removes the need for

online data filtration using a known prefix or suffix, allowing us to leak random-looking data spanning multiple consecutive cache lines without the need of some known structure.

Indeed, instead of using a known prefix or suffix, we use CacheOut to simply read as much data as possible from the L1-D cache. As we know the location of the data pieces relative to each other and relative to public data in the victim's address space, we are able to reconstruct a portion of the victim's address space that is located inside the L1-D cache, albeit with some amount of errors due to noise. Finally, we exploit redundancies in the data such as derived AES round keys or the relationship between p, q and $n = pq$ for RSA to denoise the data, recovering the victim's keys inside the reconstructed parts of the victim's memory.

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

A. Evaluating Leakage Amount

To determine CacheOut's leakage rate, we implement a covert channel where the victim writes some byte value to a known cache location, while the attacker attempts to leak the victim's data using 10K iterations of CacheOut on the same physical core. We distinguish between not leaking anything, leaking correct values, and leaking incorrect values. We obtain a leakage rate of 26 B/s, 2918 B/s, and 343 B/s for same-thread writes, cross-thread reads and cross-thread writes respectively.

Next, rather than mitigating CacheOut, we found that Intel's `verw` instruction might ironically assist the attacker. More specifically, using `verw` before evicting the victim's data from the cache can significantly improve the signal when extracting data written by the victim, both in cross-thread and same-thread scenarios. We conjecture that this attacker-issued `verw` removes all values but the leaked one from the LFB, thereby increasing the probability of successfully recovering the leaked value through TAA. To confirm this we extend our previous experiment to issue a `verw` before running CacheOut. With `verw`, we report a leakage rate of 81 B/s, 1833 B/s and 2433 B/s, respectively.

B. Attacking Cryptographic Keys

We now show CacheOut's ability to extract random-looking encryption keys from an OpenSSL-based victim, demonstrating attacks both in the cross-process and cross-VM scenarios.

Victim and Attacker Setup. Our victim process uses the AES and RSA decryption routines from OpenSSL 1.1.1 to repeatedly decrypt an encrypted message. The OpenSSL library remains unmodified and we do not use instrumentation, as our victim only uses the public API exposed by OpenSSL. On the sibling hyper-thread, we run the CacheOut attacker which leaks OpenSSL's AES and RSA key data structures.

AES Key Extraction. We use CacheOut to sample the data from all 64 cache sets in the L1-D cache and use TAA-NG to sample the 64 bytes in every cache line to fully reconstruct 4 KiB of data from the victim, including the address corresponding to each byte of the recovered data. Examining the recovered page, we observed 98.34% of the AES key and associated round keys, where the initial AES key appears at at least two locations, providing us with additional redundancy. Next, as OpenSSL's AES key data structure lays the initial AES key and the associated round keys consecutively in memory, we use a technique similar to Cold Boot attacks [13] in order to recognize the key in the victim's memory.

For cross-process attacks, our measurement phase takes 76.2 seconds on average over ten runs, and we leak data with a raw throughput of 8.90KiB/s and an actual throughput of 63.39B/s. Finally, the key-recognition technique of [13] recovers the correct AES key from the victim's address space within 183.29s on average.

Cross-VM AES Key Extraction. We performed the same experiment on two VMs running on two different hyper-threads using QEMU 2.11.1 with KVM and 1 GB hugepages enabled. We ran our attack three times, where for each run we attempt 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. After some signal processing, we were able to recover 75% of the key bits on average across the three runs, leaking the key at an average rate of 15 seconds per single 64-byte cache line.

Recovering RSA Private Keys. We use a setup similar to the proof of concept for AES, except the victim now repeatedly decrypts a given ciphertext using RSA. Here, the attacker observes 8-byte chunks of p and q, though not in any particular order. However, as we are able to observe all chunks of p and q, we could fully reconstruct p and q using reconstruction technique described in Appendix B.

We generated 512, 1024, 2048 and 4096-bit RSA keys, and sampled key data from our victim. We gathered 100% of the key data in all cases, where the sampling phase taking 7.4-51 seconds, based on the key size. Finally, we recovered the RSA keys from the data sampled for all key sizes, using the technique described in Appendix B in about 2.5 min/key.

We also performed the same experiment across VMs set up on two sibling hyper-threads. We found that we can extract 100% of the key data, taking between 11-24 seconds to sample data from the victim VM (with larger keys requiring more time). Our reconstruction phase described in Appendix B took between 2 and 95 seconds for key reconstruction (with larger keys needing more CPU time). Finally, we note that previous MDS atacks (e.g., Zombieload [43] or Medusa [36]) did not demonstrate any cross-VM data extraction attacks (besides a covert channel), presumably due to noise and the inability to exercise sufficient control over the data leaked by the attacker.

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. We run a victim which uses the generic FANN model [38] created by `fann_create_standard()` to repeatedly classify a random chosen piece of English text as one of three languages. We again run CacheOut on the

sibling hyper-thread using 5K iterations to sample data from each byte offset. 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 a number of techniques for identifying the weights' values.

More Specifically, the model has 376 weights, each represented in 32 bits, resulting in a 1504 B array. Since the weights are stored sequentially, finding the array's start 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 it always starts at a fixed page offset. Thus, the attacker can find this location in an offline phase, enabling them to specifically target the cache lines containing the weights during the online phase. With the naive approach of simply selecting the most common 8 byte value for each offset containing a weight, we achieve 63.0% accuracy for determining the value of each weight. In Appendix D we describe how to improve the accuracy to 96.1% by exploiting 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 1504 B weight array spans 23 different cache lines, even if the attacker could accurately identify each weight with 100% accuracy, they would not be able to determine which weight connects which two neurons.

Trying to leak the weights from a trained neural network, our attacker takes 40 seconds to run on average over ten runs, recovering the model's weights with a 95.2% accuracy. Finally, we also reproduced similar results for stealing the weights from FANN across VM boundaries, obtaining an average run time of 376.69 seconds and accuracy of 93.95%.

VI. Attacking Kernel and Hypervisor

The results of Section V clearly demonstrate CacheOut's ability to recover information from cross-process and cross-VM boundaries on MDS-resistant machines, assuming hyper-threading is enabled. While hyper-threading has been re-enabled on these architectures, in this section we focus on the case where hyper-threading is disabled, demonstrating that CacheOut can also leak data from the unmodified kernels and hypervisors. More specifically, we demonstrate CacheOut's ability to control what to leak by again focusing on random-looking data, recovering kernel stack canaries and break KASLR by recovering kernel pointers. As both of these countermeasures are aimed at preventing privilege escalation attacks, the ability to recover both kernel stack canaries and pointers makes it significantly easier to exploit software vulnerabilities on unpatched kernels and hypervisors.

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 CacheOut's line selection capabilities enable an attacker to reliably leak a kernel function pointer, thus breaching KASLR in under a second. The attacker binds itself to a single core and executes a loop composed of a `sched_yield()` followed by our TAA-NG primitive. When `sched_yield()` returns from the kernel, we use TAA-NG to leak stale L1-D data leftovers from the kernel during the context switch. We used TAA-NG 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. Here, the attacker only has to leak a single cache line and eight byte offsets that contain the kernel pointer, as opposed to an entire 4 KiB 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 [11] are another widely deployed defense-in-depth countermeasure aiming 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).

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 attack 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-NG to leak from the L1-D cache. We found three different locations where the kernel's stack canary can be leaked. On average, the attack succeeds in 23 seconds. To our knowledge, CacheOut is the first transient-execution attack that recovers kernel stack canaries. This is made possible by the address selection capabilities, as a completely random 64-bit value is extremely difficult to detect without targeting a particular cache line.

C. Breaking Hypervisor ASLR

Similar 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 hypervisors often hide or modify the CPU information presented to the guest VM. We assume an attacker VM controlling both threads of at least a single CPU core. The

attacker runs a loop issuing `cpuid` on one of the threads, while running the attacker program on the other thread.

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 from actual hypervisor pointers, we simply reboot our VM. This forces the guest kernel to choose new random values for KASLR, while the hypervisor keeps using the same random value, allowing 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 that we can leak hypervisor addresses from. We found that there are indeed various locations that leak a hypervisor pointer to `x86_vm_ops`. After establishing the fixed locations for a known kernel, we can mount an online attack on the hypervisor. This reduces the time from roughly 17 minutes in the offline phase to 1.8 seconds.

VII. 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. 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], CacheOut is capable of attacking SGX data at rest, dumping the enclave's entire address space, without even requiring the victim enclave to execute or perform any memory operation. Moreover, CacheOut's ability to control which memory address is being leaked allows us to recover unstructured large secrets, such as images. Finally, we show how to extract EPID signing keys from SGX's quoting enclaves in *release* mode. With production attestation keys at hand, we are able to pass fake enclaves as genuine, issue fake attestation quotes, or even allow AMD and ARM machines to pass as genuine Intel hardware.

Following SGX's threat model, we assume a malicious OS that aims to breach enclave confidentiality.

Experimental Setup. We run the SGX attacks using two machines, an i7-8665U (Whiskey Lake) and an i9-9900K (Coffee Lake Refresh, Stepping 13). Both machines use microcode version `0xca` and are mitigated against MDS and Foreshadow, meaning that enabling hyper-threading does not violate SGX's security and Intel considers hyper-threading to be safe on these machines. Furthermore, the machines execute the latest microcode at the time of writing, which mitigates TAA attacks on SGX by disallowing TSX transactions on logical cores that are co-resident with logical cores running SGX enclaves [16].

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, without having the victim enclave running. 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.

Loading Secret Data into the Cache. Similarly to [46], we used the `ewb` and `eldu` instructions to load the victim's decrypted page into the L1-D. See Steps 1 and 2 in Figure 9. However, we improved upon this technique by forcing multiple copies of the plaintext corresponding to the victim's page into the cache, thereby obtaining a stronger leakage signal. To achieve this, each time the attacker executes `ewb` and `eldu`, they allocate 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 L1-D cache with an attacker-chosen page from the victim's enclave, thereby improving the probability of evicting the correct data.

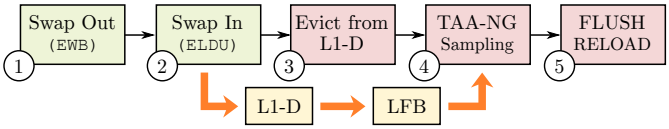

Fig. 9: Overview of how the SGX paging mechanism, in combination with TAA, leaks arbitrary SGX data.

Reading Secret Enclave Data. After loading the secret data into the L1-D (Figure 9, Steps 1-2), we execute a CacheOut attacker that performs Steps 3-5. As CacheOut is able to reconstruct entire 4 KiB pages from the L1-D cache, this allows us to retrieve entire pages from within the enclave. Finally, as `ewb` and `eldu` operate at page granularity, we are able to focus on specific pages or even dump the entire content of the enclave (by iterating over all of its pages), without the need for any memory access from the victim enclave.

Bypassing TAA Countermeasures. With the introduction of the microcode patches to mitigate TAA, on MDS-resistant machines (such as the one used in this section), Intel provides operating systems with the ability to disable TSX. However, as a malicious OS can simply enable TSX, Intel's TAA mitigations abort TSX transactions on the sibling hyper-thread when entering an SGX enclave, resulting in a core-wide 'SGX⊕TSX' policy where both cannot execute concurrently.

However, we observe that by continuously swapping in and out enclave pages of interest from the L1-D cache using the `ewb` and `eldu` instructions, we are still able to use CacheOut's TAA-NG primitive for leaking the page's content, thereby breaking SGX on TAA-mitigated machines. As TAA-NG uses TSX, we conjecture that Intel's countermeasure does not consider the `ewb` and `eldu` instructions to be SGX-related activity, allowing us to circumvent Intel's 'SGX⊕TSX' policy.

SGX Image Extraction. To quantify our leakage from SGX, we set up an SGX enclave that contains a picture of the Mona Lisa (Figure 10 (left)), and leak the picture using CacheOut. As the image is 128 by 194 pixels, it spans multiple pages. Thus, we use the above-described `ewb` and `eldu` technique on each image page individually, and use address selection

for leaking unstructured pixel data from the entire page. We note that CacheOut is effective despite the victim enclave not performing any processing on the image, and is in fact not even executing after placing the image in its memory space. We sample the image data from the SGX enclave five times, observing 36% of the image data on average in each run.

Image Reconstruction. To reconstruct the Mona Lisa from the collected data sampled over multiple runs, we use our address selection capabilities to obtain all the candidates for every pixel address from our sampled data. Then we score each candidate based on the candidates for neighboring pixels using a distance function, selecting 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 10 (right).

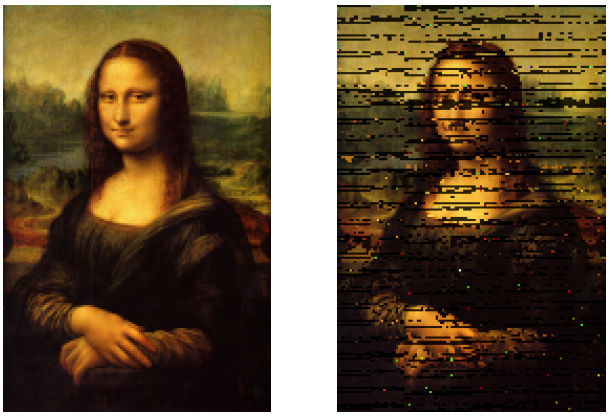

Fig. 10: On the left the original picture (128x194) and on the right the picture recovered from an SGX.

B. Extracting Production SGX EPID Attestation Keys

One of most compelling properties that SGX provides is the ability of an enclave to attest to a remote verifying party that it is running on genuine Intel hardware and not on a malicious SGX simulator that does not offer any security properties. From a cryptographic perspective, this is done using an Enhanced Privacy ID (EPID) attestation key, which is available only to enclaves written and signed by Intel. In particular, a compromise of even a single EPID key breaches the security of the entire SGX ecosystem, as it allows the attacker to sign fake unlinkable attestation quotes, pass fake enclaves as genuine, or even enable AMD / ARM CPUs running SGX simulators to masquerade as genuine Intel hardware.

Experimental Setup. We begin by configuring an SGX-capable machine to be in a fully trusted state according to Intel's Attestation Sever (IAS). As our Whiskey Lake platform cannot obtain a fully trusted status due to an unrelated security vulnerability with its internal GPU [22], we used a Coffee Lake Refresh i9-9900K (stepping 13) desktop with the latest microcode at the time of writing (`0xcc`), mitigating all SGX attacks known at the time [16, 37, 43, 47, 48]. After contacting the IAS, the machine was deemed to be fully trusted and was provisioned by Intel with an EPID attestation key. This key is stored as a normal file, but encrypted using *sealing keys* that are only available to Intel's quoting and provisioning enclaves.

Attacking Production Quoting Enclaves. To extract the machine's EPID key, we reverse engineered the quoting enclave binary shipped and signed by Intel, and used a controlled-channel attack [50] to pause Intel's quoting enclave after it has loaded the EPID sealing key into memory. After this point, the quoting enclave never resumes execution and is permanently stopped. In particular, it is unable to apply any defensive measures, such as side-channel detection [7, 39, 44] or wiping the key from memory. After stopping the enclave, we use the technique from Section VII to repeatedly swap in and out the page containing the sealing key, extracting it using CacheOut. The entire extraction process lasted about 1.5 minutes and the key recovery takes less than a second, allowing us to use the sealing key to successfully decrypt the EPID key file.

Signing Fake Attestation Quotes. Demonstrating our ability to sign arbitrary attestation quotes, we created an attestation report setting the MRENCLAVE field, representing the SHA-256 of the enclave's initial state, to be the string "*This enclave should not be trusted*", the MRSIGNER, representing the SHA-256 of the public key of the enclave writer, to be "*CacheOut*", and setting the report's debug flag to 0, thereby indicating that the enclave is a production enclave. We have also populated the report's body (commonly used for establishing a Diffie-Hellman key exchange with the enclave corresponding to the report) to be "*Mary had a little lamb, Little lamb, little lamb, Mary had a....*". Finally, we signed the report using our malicious quoting enclave that uses our extracted EPID keys, thereby producing an attestation quote.

Quote Verification. To verify the validity of our quote, we sent it to Intel's IAS for verification. According to Intel's SGX manuals [4, 26], the IAS will only approve the quote if it can verify that the quote's EPID signature is correct. Since we have correctly extracted a non-revoked EPID private signing key, using version 3 of the attestation API [23], the IAS accepted our quote and replied with "`isvEnclaveQuoteStatus: OK`". The IAS also signed its response with Intel's private key and accompanied it with the appropriate certificate chain leading to Intel's CA certificate.

Comparison to State-of-the-Art. Our breach of SGX exmplifies CacheOut's advancement over the state of the art in transient-execution attacks, in terms of both noise, and control over the data that the attacker would like to leak. While several previous transient-execution attacks demonstrated leakage from SGX enclaves [43, 47, 48], the only other attack which demonstrated a sufficient amount of control over the leakage signal to extract production EPID keys from architectural enclaves, compiled and signed by Intel, is Foreshadow [46], which has been long since mitigated on recent Intel hardware.

VIII. Mitigations

With Intel's approach of applying spot mitigations to address transient-execution vulnerabilities, some issues are often overlooked resulting in incomplete mitigations. For instance, `verw`'s initial implementation turned out to be

incomplete [42], and we discovered that Intel's mitigations against TAA do not fully protect Intel SGX. Intel's mitigations focus on disabling certain features on vulnerable CPUs, or to provide mechanisms to flush buffers as well as the L1-D cache, rather than addressing the root causes. We discuss both various solutions to mitigate CacheOut on current hardware, as well as various solutions to fundamentally address these issues.

Disabling Hyper-Threading. Similar to MDS, CacheOut works best when the attacker and victim run on two parallel hyper-threads. However, as CacheOut is also effective in the scenario without hyper-threading where the attacker and the victim run on the same CPU thread, disabling hyper-threading makes the attack difficult but not impossible (see Section V and VI). 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, CacheOut leaks information from the L1-D cache. Thus, one might attempt to flush the L1-D cache 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 can leak data from the L1-D cache as the victim accesses it. Thus, given the cost of implementing both of these countermeasures, we do not recommend deploying them to mitigate CacheOut.

Disabling TSX. As discussed in Section III, the microcode update to address TAA [16] only provides the ability to disable TSX on platforms released after 2018-Q4, leaving the vast majority of Intel platforms vulnerable. Given that TSX is not widely in use, we recommend to disable TSX by default on all CPUs. However, as a malicious operating system can always re-enable TSX to use CacheOut to leak data from SGX enclaves while bypassing Intel's SGX countermeasures for TAA (as we demonstrated in Section VII) leaving SGX vulnerable, we recommend to disable TSX at the microarchitectural level.

Microcode Updates. On June 9, 2020 Intel had published a security advisory [20] and a microcode update aiming to mitigate CacheOut (called L1DES in Intel's terminology), which indeed seems to mitigate CacheOut's root cause. 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 to install these on affected systems, especially on pre-2018-Q4 machines where TSX is still enabled by default.

Partitioning. As hyper-threads share resources, those are either partitioned statically or dynamically, or they are competitively shared [18], leaving some of them vulnerable to contention attacks. Even though statically dividing resources such as the LFB and the L1-D cache mitigates this, the CPU could alternatively provide something similar to Intel CAT [17, 33] to provide control over the partitioning. To avoid sharing resources between transactional and non-transactional code, TSX should maintain dedicated read and write sets.

Serializing TSX. Through experimentation we found that issuing `mfence` before the transaction is an effective mitigation against CacheOut. Intel's manual [19, vol. 3 pg. 3-145] states that `mfence` and `clflush` instructions are ordered. Thus, modifying the `xbegin` to issue an `mfence` before executing the transaction forces the CPU to complete any pending `clflush` instruction, preventing the attacker from exploiting `clflush` to perform TAA and TAA-NG.

IX. Related Work

Several prior works explore leakage from internal CPU buffers. These include RIDL [48], ZombieLoad [43], Fallout [5], and Medusa [36], collectively known as *MDS attacks*.

RIDL. RIDL [48] shows that faulty loads are served from LFBs and load ports, bypassing any address and permission checks. While RIDL shows statistical evidence that data evicted from the L1-D cache ends up in the LFB, it does not study the security implications of the issue or Intel's MDS countermeasures. Moreover, RIDL lacks control over what data the attacker leaks, instead relying on averaging techniques to filter the data from the acquired noise.

ZombieLoad. ZombieLoad [43] extends RIDL's findings to loads that require microcode assists showing that leakage exists even without faulting loads. It also demonstrates LFB leakage from the MDS-resistant Cascade Lake architecture. Moreover, ZombieLoad [43, Section 7] mentions the possibility of leakage via L1-D cache evictions to the LFB. However, the authors argue that the leakage is negligible at 0.1 B/s. ZombieLoad resorts to Domino-bytes to process the data as it has no control over the leakage. Finally, while ZombieLoad does mention the possibility of hypervisor and cross-VM leakage, Schwarz et al. [43] only demonstrate a cross-VM covert channel (presumably due to noise).

Medusa. In concurrent independent work, Medusa [36] recovers information from write-combining (WC) operations [9] by sampling data from the LFB where the write combining takes place on Intel CPUs [19, vol. 3 pg. 6-38]. By focusing on recovering data from write combining, Medusa can obtain a cleaner LFB signal. However, Medusa has no control over the exact offsets and can only partially sample the in-flight data. This results in slow leakage rates of 12 B/s for kernel data, and the need for Domino-bytes signal averaging. Unstructured data (e.g., RSA keys) requires a 400 CPU hour lattice attack [10] to recover the 1024-bit RSA key from the raw leakage, which is obtained during a 7 minutes measurement phase.

X. Acknowledgments

This work was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; the Australian Research Council projects numbers DE200101577 and DP210102670; the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL) under contracts FA8750-19-C-0531 and HR001120C0087; the National Science Foundation (NSF) under grant CNS-1954712; the Cyber Security Research Center at Tel-Aviv University; and by gifts from Intel and AMD.

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] E. Brickell and J. Li, "Enhanced privacy ID from bilinear pairing for hardware authentication and attestation," *International Journal of Information Privacy, Security and Integrity 2*, vol. 1, no. 1, pp. 3–33, 2011.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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

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

[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] 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.

[13] 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.

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

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

[16] Intel, "Deep dive: Intel 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.

[17] ——, "Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 family," https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html, Nov 2016.

[18] ——, "Intel 64 and IA-32 architectures optimization reference manual," Jun 2016.

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

[20] ——, "L1D eviction sampling," https://software.intel.com/security-software-guidance/software-guidance/l1d-eviction-sampling, Jan 2020.

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

[22] ——, "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.

[23] *Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation*, Intel, https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf.

[24] 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.

[25] 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.

[26] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel software guard extensions: EPID provisioning and attestation services," White Paper, 2016.

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

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

[29] 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.

[30] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the

return stack buffer," in *WOOT*, 2018.
[31] 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.
[32] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE SP*, 2015.
[33] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, 2016, pp. 406–418.
[34] 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.
[35] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *CCS*, 2018, pp. 2109–2122.
[36] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *USENIX Security*, Aug. 2020.
[37] 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 *IEEE SP*, 2020.
[38] S. Nissen, "Implementation of a fast artificial neural network library (fann)," Department of Computer Science University of Copenhagen (DIKU), Tech. Rep., 2003.
[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] RedHat, "Intel November 2019 microcode update," https://access.redhat.com/solutions/2019-microcode-nov, Nov 2019.
[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 *IEEE SP*, 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] 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.
[50] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE SP*, 2015, pp. 640–656.
[51] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.

APPENDIX A
TSX ASYNCHRONOUS ABORT

The TAA primitive relies on TSX instructions as well as on the the `clflush` instruction, which flushes specific cache lines from the cache hierarchy. In particular, Intel's manual [19, vol. 2A, pg. 3-139–3-142] states that executing the `clflush` instruction for a cache line read or modified by a transaction may cause the transaction to abort, whereupon the CPU reverts the architectural state modified by the transaction.

TAA exploits this behavior by executing a transaction that attempts to load from a cache line that was just flushed via the `clflush` instruction. While this will eventually cause a transactional abort, it does allow the attacker to recover information from the CPU's LFB. First the attacker flushes some cache line in their own address space. The attacker then initiates a TSX transaction which attempts to read that cache line. More specifically, Listing 3 shows the Assembly code for the TAA primitive, as described in [16, 43, 48]. First the attacker flushes some cache line in their own address space (Line 5 in Listing 3). The attacker than initiates a TSX transaction (Line 9) which attempts to read the cache line flushed at Line 5. As the `clflush` instruction does not complete immediately, the transaction proceeds to execute speculatively until the `clflush` instruction completes. Next, during the speculative execution of the load instruction, the processor allocates an entry from the LFB. As the data for the load instruction is not available due to the previous `clflush`, the transaction proceeds speculatively with stale data present in the LFB entry from a previous memory access. The attacker then leaks this data using a FLUSH+RELOAD side channel. Finally, once the `clflush` instruction completes, the CPU aborts the transaction, attempting to roll back the speculation. However, at that point the attacker has already extracted the data using the FLUSH+RELOAD side channel.

```
; %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 3: the leak primitive using TSX Asynchronous Abort.

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 [14], $N = p \cdot q$ 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 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 two least-significant 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
Exploring Cross-Thread Evictions.

To confirm our belief that the L1-D cache influences our leakage, we now perform an experiment where the sibling hyper-thread tries to evict the L1-D cache set used by both attacker and victim as well as an unrelated cache set. For this experiment, we extend the experiment from Figure 4 where the victim first writes data into different cache lines and where the attacker subsequently tries to leak data from a specific cache set on the same hyper-thread. Whereas the sibling hyper-thread was previously idle, the sibling hyper-thread now performs a number of evictions on either the same L1-D cache set or an unrelated one to determine the influence of cross-thread evictions. We expect that if the sibling hyper-thread evicts the same L1-D cache set as the one used by both victim and attacker, that the attacker will no longer be able to observe the victim's data. The results of this experiment can be seen in Figure 11. Indeed, we see that as the sibling hyper-thread evicts the same L1-D cache set, the attacker no longer is able to observe the victim's data. Whereas if the sibling hyper-thread accesses an unrelated L1-D cache set, the signal remains largely unaffected. This again confirms that CacheOut is indeed leaking from the L1-D cache.

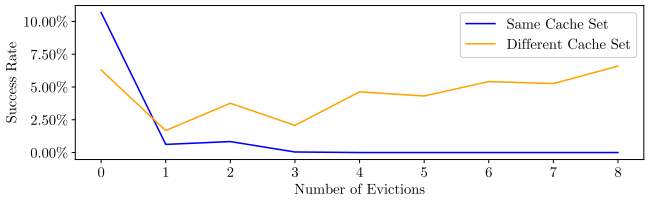

Fig. 11: The victim writes data to different cache lines and the attacker subsequently tries to target a cache set to leak data from on the same hyper-thread. Simultaneously, the sibling hyper-thread performs a number of evictions on the same L1-D cache set as the one used by both victim and attacker, as well as an unrelated one to determine the influence of such evictions.

Appendix D
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%.