Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis

Daniel Moghimi, *Worcester Polytechnic Institute;* Moritz Lipp, *Graz University of Technology;* Berk Sunar, *Worcester Polytechnic Institute;* Michael Schwarz, *Graz University of Technology*

https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-medusa

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis

Daniel Moghimi[1], Moritz Lipp[2], Berk Sunar[1], and Michael Schwarz[2]

[1]Worcester Polytechnic Institute, Worcester, MA, USA
[2]Graz University of Technology, Graz, Styria, Austria

Abstract

In May 2019, a new class of transient execution attack based on Meltdown called microarchitectural data sampling (MDS), was disclosed. MDS enables adversaries to leak secrets across security domains by collecting data from shared CPU resources such as data cache, fill buffers, and store buffers. These resources may temporarily hold data that belongs to other processes and privileged contexts, which could falsely be forwarded to memory accesses of an adversary.

We perform an in-depth analysis of these Meltdown-style attacks using our novel fuzzing-based approach. We introduce an analysis tool, named Transynther, which mutates the basic block of existing Meltdown variants to generate and evaluate new Meltdown subvariants. We apply Transynther to analyze modern CPUs and better understand the root cause of these attacks. As a result, we find new variants of MDS that only target specific memory operations, e.g., fast string copies.

Based on our findings, we propose a new attack, named Medusa, which can leak data from implicit write-combining memory operations. Since Medusa only applies to specific operations, it can be used to pinpoint vulnerable targets. In a case study, we apply Medusa to recover the key during the RSA signing operation. We show that Medusa can leak various parts of an RSA key during the `base64` decoding stage. Then we build leakage templates and recover full RSA keys by employing lattice-based cryptanalysis techniques.

1 Introduction

Microarchitectural side channels have been known for more than a decade, with attackers mostly focusing on leaking memory access patterns through shared CPU resources [48]. These side-channel leakages can be used to compromise specific secrets such as cryptographic keys [28, 63]. However, in 2018 a new generation of microarchitectural attacks, including Meltdown [39] and Spectre [35] changed the perspective by introducing data leakage from the CPU. These new attacks, under the taxonomy of transient-execution attacks, rely on extracting secrets that are only visible in transient states within the CPU [11]. Compared to previous side-channel attacks, the significant impact of transient-execution attacks is that they can leak actual data bits instead of access patterns.

Spectre attacks [21, 34, 35, 37, 40] miss-train branch predictors into executing control paths that might not be taken by the architecture. Meltdown-style attacks [10, 39, 50, 52, 56, 57, 58] exploit the heavily optimized out-of-order load operations in which faulting memory loads still proceed with stale or illegal data. In both cases, the microarchitecture may access secrets across security boundaries. These secrets, never architecturally visible, can be transmitted via a covert channel, e.g., using Flush+Reload [63]. Canella et al. [11] proposed a taxonomy to classify transient-execution attacks based on the cause of the transient-instruction sequence and the exploited microarchitectural buffer. While this classification captures the cause and targets of known variants in a structured way, it does not provide enough information on how a certain attack can be carried out. For most Meltdown attacks, there are multiple ways to trigger the leakage, e.g., some attacks seem to require TSX to enable the leakage [25, 52], while others can also leverage signal handlers or miss-speculation [10, 39].

Meltdown-type attacks exploit special events in the microarchitecture, which require so-called microcode assists. Microcode assists execute software routines in the CPU to handle operations which cannot be directly handled in hardware, e.g., certain faults, or updating bits in page-table entries. For some Meltdown attacks, microcode assists have enabled new variants [10, 52]. In this paper, we propose a systematic approach for evaluating data leakage due to the combination of microcode assists caused by a load with dependent operations. To achieve this goal, we propose *Transynther*[1], a tool to automatically generate and test the combination of known building blocks for Meltdown attacks with various faults and microcode assists. Furthermore, we use fuzzing-type techniques to mutate, evolve, and combine building blocks. Transynther can automatically evaluate whether the newly synthe-

[1] Transynther tool and Medusa attack code are available as an open-source implementation on GitHub: https://github.com/vernamlab/Medusa

sized code variants are indeed a variant of a Meltdown attack by trying to leak predefined values.

We automatically generated thousands of different combinations using Transynther. Transynther reproduced Meltdown [39], ZombieLoad [52], RIDL [58], Fallout [10] (MSBDS), Store-to-Leak (S2L) [50], Spectre v1.2 [34], and Microarchitectural Load Port Data Sampling (MLPDS) [24]. Furthermore, with Transynther, we synthesized multiple new, previously unknown variants to trigger these attacks. Consequently, by analyzing the generated variants, we gained additional insights into Meltdown-type attacks. We identified that the root cause of all known Meltdown-type attacks is that an aborted load operation simply consumes any data which can be fetched first, and provides them to dependent operations.

In addition to reproducing known attacks, Transynther also discovered new variations of MDS variants, which we refer to as *Medusa*. Medusa provides more in-depth insight into how the memory subsystem is implemented in Intel microarchitectures. Medusa specifically targets data values which are transferred via the common data bus but are not normal data loads. In addition to AVX2 loads, Medusa has the unique property to observe the inner workings of implicit write combining (WC) used by the CPU, e.g., fast string operations such as `rep mov`. For WC, the CPU allocates parts of the line-fill buffer to combine multiple stores to the same cache line to increase the throughput. In contrast to ZombieLoad [52] and RIDL [58], which leak arbitrary data from the line-fill buffer, Medusa specifically targets data transfers caused by WC.

With Medusa, the leakage is extremely targeted and noise-free, as only specific loads are leaked. Thus, while the property to only leak data from WC sounds like a limitation, it is an advantage over previous data-sampling attacks. Where data-sampling attacks such as ZombieLoad [52] or RIDL [58] require extensive post-processing to find the target data within the leaked data, Medusa does not leak such large amounts of unrelated data in the first place. This is especially important as ZombieLoad and RIDL, in practice, leak too many unrelated data when they are applied to applications that perform a long sequence of operations. For instance, in our case study on RSA, the computation steps, including loading the key from the disk and performing the RSA signing operations, consists of thousands of load operations that may not be interested for an attacker to be leaked. In a case study, we use Medusa to steal private RSA keys loaded in OpenSSL. This attack takes at most 7 minutes during the online phase. By leaking various blocks of the RSA private key, we can employ lattice-based cryptoanalysis techniques to recover the entire key.

Finally, we discuss how the current mitigations against MDS attacks apply to Medusa. We show that currently, Medusa cannot be prevented if hyperthreading is enabled. Hence, we stress that hyperthreading has to be disabled to entirely prevent Medusa.

To summarize, we make the following contributions:

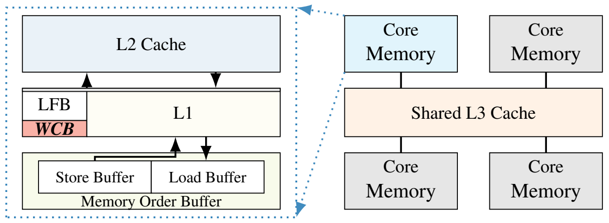

Figure 1: The fill buffer serves memory accesses that miss the L1 cache. The *WC buffer* is a part of the fill buffer optimizing multiple store operations that target the same cache line.

1. We introduce a new open-source tool, Transynther, to analyze the CPU microarchitecture for Meltdown-style vulnerabilities.
2. We provide insight into the root cause of Meltdown attacks and disclose new exploitation methodologies.
3. We introduce the Medusa attack, exploiting implicit write combining of memory store operations, e.g., `rep mov`.
4. In a case study, we use Medusa to recover RSA keys from OpenSSL by exploiting leakages during key decoding.

Responsible Disclosure. We disclosed our initial finding to Intel on June 24, 2019. Intel confirmed that the WC is part of the fill buffer. The paper was under embargo until November 12, 2019, as we exploit TSX Asynchronous Abort (TAA, CVE-2019-11135) [25] in several proof of concepts.

2 Background

2.1 Superscalar Memory Architecture

Modern CPUs have multiple levels of caches and buffers to mitigate the speed gap between execution units and the main memory. Figure 1 illustrates the memory components on the data path of an Intel processor. The last level cache (LLC) which is shared across CPU cores is connected through an interconnect bus to the main memory. Further, each core has an L1 and L2 cache, consisting of multiple cache lines which are usually 64 B. When the processor accesses data that is not present in a cache level (cache miss), the corresponding cache line is fetched from the next level of cache or the main memory. The processor also uses a fill buffer to service memory accesses missing in the L1 cache. The data in the fill buffer can be forwarded to memory loads before filling the entire cache line. ZombieLoad [52] and RIDL [58] showed that Intel processors may falsely forward data that resides in the fill buffer from a benign to a malicious load.

Memory operations are executed out of order and speculatively. The processor may execute a load before preceding stores to avoid pipeline stalls due to the potential dependency of the load on stores. The store buffer, as part of the memory order buffer (MOB), temporarily holds the data and metadata for stores before committing them to the cache. The CPU may forward data from the store buffer to a load (store-forwarding). The CPU may fail to predict correct dependencies between the load and stores [30, 46]. While such failures are finally resolved before committing the results, it facilitates transient execution attacks [10, 21, 50].

Memory Types. CPUs support multiple per-page memory types with different policies for caching and ordering guarantees. The supported memory types on x86 are write-back (WB), write-through (WT), write-protect (WP), write-combining (WC), and uncachable (UC). Most pages are write-back, which allows them to be cached and written back to the memory at a later point. Both UC and WT write data directly to memory. Write-combining memory, as discussed later on, tries to reduce the number of bus requests by combining multiple writes to the same cache line into a single request.

2.2 Write Combining

A memory store has to update core-private caches, the LLC, and possibly the main memory. Thus, for performance, it is beneficial to combine multiple stores into a single request. This reduces the number of bus requests and cross-core snoops that update the core-private copy of the cache. Employing write combining (WC), the CPU temporally holds the data of store operations to the same cache line in an internal buffer, until all the data bytes that modify that cache line are available. The WC buffer can be either implemented as a dedicated component as in AMD CPUs [1] or as part of the fill buffer as in Intel CPUs [27]. WC is often used for memory where memory ordering guarantees are weak, e.g., for frame buffers of graphic cards, which are usually treated as write-only by programmers [22].

2.3 Advanced CPU Features

Simultaneous Multithreading. Simultaneous multithreading (SMT) allows multiple threads to execute on the same core simultaneously while sharing the same resources. These threads are architecturally isolated from each other according to memory protection semantics and only access their intended data. This allows one thread to use the available resources not used by other threads.

Intel Hyperthreading technology implements SMT by sharing the core between two simultaneous threads, logical CPUs. These logical CPUs share some of the resources such as the store buffer in a compartmentalized fashion where the resource is halved into two separate sections upon activation of the second thread. Other resources, such as the fill buffer, are time shared. Intel Hyperthreading has suffered from various microarchitectural side channels due to the time-sharing of resources such as translation look-aside buffer (TLB) [17] and execution ports [2]. MDS attacks demonstrated data leakage due to sharing of various buffers within the core [24, 52, 58].

Transactional Memory. *Intel Transactional Synchronization Extension (TSX)* implements *Hardware Transactional Memory* by extending the instruction set with a new set of barriers in which application developers can define a block of code as atomic by surrounding it with the `xbegin` and `xend` instructions. The CPU only commits the results of a transaction if the entire block executes successfully. TSX transactions are aborted on conflicting cache and memory operations that may affect the atomicity of the transaction, as well as on interrupts. Intel TSX has been exploited for both attack and defense [18, 31, 51]. In Meltdown attacks, TSX can be abused as a silent event suppression mechanism that may enable further leakages (cf. Section 2.4).

2.4 Microarchitectural Attacks

Flush+Reload. Flush+Reload [63] exploits the difference in memory-access times for cached and uncached shared memory pages. In a Flush+Reload attack, the attacker flushes the cache line for a shared memory address using the `clflush` instruction and subsequently measures the access time to the memory. If the execution time is high, the data has not been cached. However, if another execution context accesses the address, the attacker observes a low access time as the data is cached. Flush+Reload has been used to attack cryptographic implementations [5, 20, 29] as well as to spy on user's behavior [19, 38, 64]. As in previous meltdown-type attacks [39, 52, 57], we use Flush+Reload as a covert channel from the microarchitectural to the architectural domain.

Transient-Execution Attacks. Modern CPUs employ out-of-order and speculative execution to increase performance. With out-of-order execution, CPUs can execute instructions further in the instruction stream as long as their dependencies are satisfied. Similarly, speculative execution enables a CPU to guess the outcome of a conditional branch to continue executing the most likely path.

If an instruction which was executed out of order or speculatively was wrongly executed, this instruction is simply not committed to the architectural state. However, the instruction might have had a side effect on the microarchitectural state, such as the cache. In this case, such an instruction is called a transient instruction [11, 35, 39]. Transient-execution attacks exploit such transient instructions to leak data and are divided into Meltdown-type and Spectre-type attacks [11].

While Spectre-type attacks exploit transient instructions caused by wrongly predicted conditional branches, in Meltdown-type attacks, the attacker leverages out-of-order execution following a faulting load. The transient instructions after the faulting load still have access to the data and can encode it into the microarchitectural state [10, 39, 50, 52, 53, 57, 58, 61]. Using a covert channel, such as Flush+Reload, the attacker can then bring the microarchitectural state to the architectural state, ultimately leaking the secret.

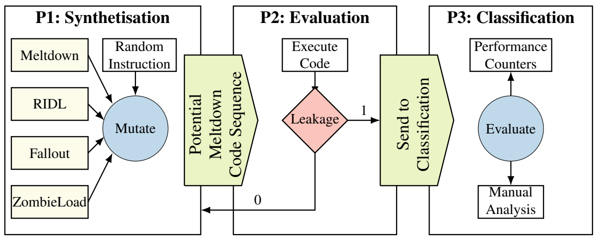

Figure 2: Transynther phases: After mutating a new code sequence for a meltdown-style attack, the code is evaluated. If there is a leakage detected, the sample is analyzed further during the classification phase.

3 Automatically Exploring Meltdown Attacks

We introduce Transynther, an automated approach for exploring Meltdown-type attacks. Transynther uses an innovative techniques based on fuzzing to systematically explore Meltdown-type attacks. The aim is to identify new variants of existing attacks, which are, e.g., faster, less complex, or are not mitigated, as well as entirely new Meltdown-type variants.

Transynther works in three phases, as outlined in Figure 2. In the first phase, the *synthetisation phase*, Transynther uses building blocks of existing attacks to mutate and combine them to potential new attacks. In the second phase, the *evaluation phase*, Transynther executes the code from the synthetisation phase and evaluates whether the code leads to data leakage. Finally, if the evaluation phase was successful, the *classification phase* tries to automatically classify the source of the leakage using performance counters.

3.1 Synthetisation Phase

The first phase is the synthetisation phase. In this phase, Transynther generates a code snippet, which is a potential Meltdown-type attack. For this, Transynther relies on building block from existing Meltdown-type attacks, including Meltdown [39], ZombieLoad [52], RIDL [58], Foreshadow [57], Fallout [10], Meltdown-PK [11], Meltdown-AVX [24], and Meltdown-RW [34].

The common pattern for all these attacks is as follows:

① Preparing the microarchitectural state (e.g., flushing, accessing, or storing data).

② Executing a load operation causing a fault (as Schwarz et al. [52], we consider microcode assists as microarchitectural faults).

③ Consuming the loaded data with dependent instructions and encoding it in a microarchitectural element.

As the encoding in ③ does not affect the root cause of a Meltdown-type attack [39, 56], we always encode the loaded value in the cache, which allows us to recover the encoded values using a Flush+Reload covert channel. This approach is used in the majority of Meltdown-type attacks [10, 11, 34, 39, 52, 53, 57, 58, 61]. Initially, Transynther sets up two pools to be used in ②. One pool contains possible *load operations* and one contains possible *load targets*:

Load operations. Memory Loads are operations that load data from memory addresses into registers. The simplest load operation is a `mov` from a memory address to a general-purpose register. Transynther supports `mov` with all possible sizes, from 8 bits to 64 bits. Additionally, aligned and unaligned AVX loads (`{v}movaps`/`{v}movups`) with a size of 128 and 256 bits are supported.

Load targets. Load targets are virtual addresses with a systematic pattern of different setup of the page-table entry, as discussed by Canella et al. [11]. As a starting point, we rely on load targets with certain page-table bits, which were already used for Meltdown-type attacks. This includes the user-accessible bit [39, 52], accessed bit [10, 52], present bit [10, 57, 58, 61], writable bit [34], and protection key [11]. For a systematic approach, we also add load targets with page-table bits that have not been used in successful Meltdown-type attacks, including the dirty bit, write-through bit, uncachable bit, size bit, and non-executable bit. Finally, we also add addresses that do not have a valid mapping to physical pages, such as non-canonical addresses (addresses where the bits 48 to 63 are different than bit 47, e.g., `0x1234567812345000`) and physically unmapped addresses, e.g., `NULL`.

Furthermore, Transynther creates a victim that injects known data throughout microarchitectural buffers by repeatedly loading and storing that data to different virtual addresses and memory types. The victim can either be a separate application running on the sibling CPU thread or running time-sliced on the same thread, e.g., using multithreading.

During synthesis, Transynther randomly chooses, mutates, and combines building blocks for ① and ②. To prepare the microarchitecture (①), Transynther randomly chooses an operation (load, store, or flush) and an address from the load-target pool. Then, the address is mutated by adding a random offset between 0 B and 4 kB. This ensures that the address still maps to the same page in most cases, however, to the page offset of a different cache line. Note that there is the case that a multi-byte load might lead to a split-page load if parts if the offset is too large. We intentionally allow this behavior, as split-page loads are also corner cases that may trigger leakage. For ②, Transynther randomly chooses a load operation and a load target. Similarly, a randomly chosen offset between 0 B and 4 kB is added to the load target address.

Transynther also randomly inserts independent operations between the preparation of the microarchitecture (①) and the faulting load (②). Such operations are, e.g., `nops` (no operations), ALU operations on unrelated registers, as well as additional architectural faults. These instructions add a certain amount of timing differences and thus increase the chance of triggering a race condition in the pipeline. These operations have been shown to increase the leakage rate for existing attacks, as observed in the published proof-of-concept implementations for other transient-execution attacks [39, 52].

Finally, Transynther adds another load operation consuming the value of the faulting load in ② and encoding it into the cache. This operation simply accesses the n^{th} page in a 256-page array, where n is the byte value provided by the faulting load in ② [11, 39]. Again, Transynther randomly inserts independent operations between this step and the faulting load to vary the timing between ② and ③.

3.2 Evaluation Phase

In the evaluation phase, Transynther evaluates whether the synthesized code snippets from the synthetisation phase lead to data leakage. Transynther uses an evaluation framework consisting of a preparation part that fills microarchitectural buffers, the synthesized code snippet augmented with exception suppression, and a Flush+Reload loop to recover the values encoded in ③. The code in the evaluation framework is executed in an endless loop for a user-specified amount of time, e.g., 2 seconds. The values recovered using Flush+Reload are compared to the known values from the preparation part. For every evaluated snippet, Transynther logs the number of correct and wrong leaked values. Snippets for which correct leakage is detected are *candidate snippets* used in the classification phase. Snippets that do not leak correct values are discarded and not further analyzed. In contrast to traditional application fuzzing, there is no feedback in our approach enabling Transynther to improve a snippet. The only feedback that the CPU provides is whether the snippet leaks data or not. Moreover, as we try to discover vulnerabilities in the microarchitecture, we cannot use a CPU emulator [42].

3.3 Classification Phase

In the final phase, Transynther analyzes the source of the leakage using *microarchitectural buffer grooming*,and *performance counters*.

Microarchitectural Buffer Grooming. The main idea of microarchitectural buffer grooming is to put microarchitectural buffers into a known state. To achieve this, we fill every microarchitectural buffer with known data that is unique for each buffer. Hence, if any leakage is observed, the leakage source can be inferred from the values. In the simplest case, each buffer contains a repeated, single printable character. For example, by storing several 'S'-characters, we "fill" the store buffer with this character. If we then leak multiple 'S'-characters, we can consider the store buffer as a potential leakage source. By having a unique character per buffer, buffer grooming provides an elementary form of data taint tracking [4]. In the case of data leakage, Transynther at least knows the origin of the data.

For buffer grooming, we only consider on-core data buffers, *i.e.*, the L1 data cache, store buffer, line-fill buffers, load buffer, load ports, and WC buffers. While buffer grooming is straightforward for certain buffers, e.g., the L1 cache, it is more difficult for other buffers, e.g., the line-fill buffer. Fortunately, Intel provides software sequences for mitigating some of the MDS attacks if microcode update cannot be used. These software sequences are designed to zero-out the data in all microarchitectural data buffers [24], *i.e.*, it sets the values in all buffers to a known value of zero.

```
1  mov %[scratch], %rdi
2  lfence
3  orpd (%[zero_ptr]), %xmm0
4  orpd (%[zero_ptr]), %xmm0
5  xorl %eax, %eax
6  1: clflushopt 5376(%[scratch],%rax,8)
7  addl $8, %eax
8  cmpl $8*12, %eax
9  jb 1b
10 sfence
11 movl $6144, %ecx
12 xorl %eax, %eax
13 rep stosb
14 mfence
```

Listing 1: Software sequence to overwrite all microarchitectural buffers for Skylake and newer microarchitectures [24].

Listing 1 shows the software sequence used to zero-out the buffers on Skylake and newer microarchitectures. In Lines 3 to 4, the load ports are zeroed out. Then, 12 cache lines are flushed (Line 6) to ensure that 12 of the subsequent writes in Line 13 have to go through the 12 line-fill-buffer entries [52]. Using `rep stosb` additionally ensures that the WC-buffer entries of the line-fill buffer are also used, and thus zeroed-out. For buffer grooming, we can rely on an adapted software sequence. Instead of writing zero to all buffers, we write a repeated, unique character to every buffer. This is as simple as, e.g., letting `zero_ptr` point to a memory content not containing 0 but 'L'-characters to ensure that load port is overwritten with repeating 'L's. Moreover, we can replace the `rep stosb` with a normal `mov` in a loop to distinguish WC buffers from general line-fill buffers.

The obvious limitation is that Transynther cannot track the actual flow of the data in hardware. For example, data in the store buffer could have already been written to the L1 cache and subsequently been leaked from the L1 cache. Still, for Transynther it looks as if the data was leaked from the store buffer. To reduce the number of false classifications, we additionally rely on hardware performance counters.

Performance Counters. To gain additional insight on the leakage source, we augment Transynther with the ability to record hardware performance counters while leaking values. Thus, in addition to the source of the leaked values, we also observe the active microarchitectural elements. Table 1 shows the performance counters we used.Some of these performance counters have already been shown to successfully identify leakage sources [30, 52]. Transynther correlates the

Table 1: The performance counters used in Transynther to identify the active microarchitectural elements.

Counter	Description
`MEM_LOAD_RETIRED.FB_HIT`	Data loaded from a line-fill-buffer entry.
`MEM_LOAD_RETIRED.L1_HIT`	Data loaded from the L1 data cache.
`MEM_LOAD_RETIRED.L2_HIT`	Data loaded from the L2 data cache.
`L1D_PEND_MISS.FB_FULL`	Data is neither in L1 nor in fill buffer.
`LD_BLOCKS.STORE_FORWARD`	Store buffer blocks load.
`LD_BLOCKS_PARTIAL.ADDRESS_ALIAS`	Load blocked by partial address match.
`MEM_INST_RETIRED.SPLIT_LOADS`	Data spans across two cache lines.

performance-counter values with the number of leaked bytes using the Pearson correlation coefficient (Figure 8 in Appendix B). A high positive correlation between the number of leaked bytes and the events for a microarchitectural element indicates that this element is involved in the leakage. With microarchitectural buffer grooming and the correlation coefficient from the performance counters, Transynther can provide an educated guess of the leakage source.

3.4 Transynther Results

In our first set of experiments on Intel CPUs, we ran Transynther for about 46 500 test cases distributed on the three Intel Core i7-7700 (Kaby Lake), i7-8650U (Kaby Lake R), and i9-9900K (Coffee Lake) CPUs. We ran each test case for 2 s, totaling about 26 CPU hours. Transynther generated 5100 code snippets, which showed transient leakage. Based on the classification and subsequent manual analysis, we filtered the generated code snippet to 100 interesting cases with a unique code and leakage pattern. We identified multiple classes of leaking code sequences, as described in Section 3.4.1.

We also ran some tests on an AMD Ryzen 5 2500U and show that while there is no data leakage on AMD, AMD is not by-design immune to the root cause of Meltdown-type attacks. In our second experiment, we ran Transynther for about 10 000 test cases on an AMD machine. Similarly, we ran each test case for 2 s, totaling about 5 CPU hours. We report our findings in Section 3.4.2.

3.4.1 Intel

Split Cache Access. Transynther reproduced various variants of split cache access that lead to MLPDS. Split accesses refer to memory accesses that span over two cache lines and are handled differently from normal loads accessing a single cache line. In the generated proof of concepts, we can observe that when split access is suffering a faulty load, it directly leaks the data that is loaded by the sibling CPU thread (①). Split access works for page faults (user-accessible and present), as well as for microcode assists caused by setting the accessed bit. We only saw MLPDS leakage on Kaby Lake and Kaby Lake R but not on the Coffee Lake microarchitecture. Another observation is that MLPDS with split access works much faster when there is a page fault caused by accessing a non-present page before the target faulty load[2]. In contrast, a page fault caused by accessing a non-user-accessible page does not increase the leakage rate. Split accesses can also be triggered via vector move instructions (②), which lead to the same behavior and leakage.

Vector Move. A faulting vector load instruction with correct alignment and without crossing a cache line can leak data (③). [3] Depending on which part of the vector is read, it can leak different parts of the implicitly write-combined data. Prior faults also affect which part of the data is leaked. We hypothesize that this is due to the different time it takes to handle the exception for the fast string copy operation. Faulting vector loads also show fast leakage for a non-canonical address, whereas a simple non-canonical fault requires additional memory grooming to work. In contrast, to split cache accesses, we did not observe leakage for a page fault in our setup of microarchitectural buffer grooming. Note that while Intel refers to all these cases as MLPDS [24], we distinguish the specific case of leaking from implicit WC.

AVX Alignment Fault. Transynther created many variants of alignment-enforcing vector loads, e.g., `vmovaps`, in combination with unaligned addresses, leading to a general-protection exception. The results indicate that the alignment exception is prioritized in the pipeline as it does not depend on the address type (④). In contrast to ③, ④ also works with page faults and even valid addresses that are not causing any faults for regular memory operations, e.g., `vmovups or mov`.

Store-to-leak. Transynther showed that during a TSX transaction, Store-to-leak [11] works on all addresses except for non-present addresses (⑤).

Transynther also generated a case that when an unrelated `rep mov` instruction is executed before the store, Store-to-leak does not forward the data anymore. We further noticed that adding a fence instruction between the store and load prevents Store-to-leak. For Fallout [10], it has no effect (⑥).

4K-Aliasing Forwarding (Fallout). As discovered in Fallout [10], store-to-load forwarding can falsely forward data when the least-significant 12 bits of the store and load address match [46]. Transynther reproduced combinations of addresses that can forward when the store and load are a multiple of 4 kB apart (⑦). We verified that false forwarding on 4 kB aliasings only works with supervisor fault and access-bit assist. Transynther showed that the forwarding is agnostic to the address of the store, *i.e.*, any store regardless of whether the target is a valid or invalid address is forwarded as far as it meets the 4 kB aliasing condition.

Store-to-load Forwarding and AVX. In our experiments, both Fallout and Store-to-leak [10] also work with aligned

[2]In contrast to non-canonical addresses, Intel microarchitectures do not treat the null addresses differently than any other non-present pages.

[3]Vector load instructions can enforce alignment e.g., `movaps` or be alignment-agnostic e.g., `movups`. A correct alignment here means that either the address of the load is aligned, or the alignment-agnostic version is used.

Table 2: Leakage variants discovered by Transynther.

Case	Preparation	Store	Load	Name
①	(access ⊘, random instructions)	-	≺ + 🔒 / ☝ / ⊘	MLPDS
②	(access ⊘, random instructions)	-	AVX ≺ + 🔒 / ☝ / ⊘	MLPDS
③	(access ⊘, random instructions)	-	AVX + 🔒 / ☝ / <✖>	Medusa
④	(access ⊘, random instructions)	-	AVX ⇉ + 🔒 / ☝ / ⊘ / <✖> / ✓	Medusa
⑤	-	store (to load)	🔒 / ☝ / <✖> / ✓	S2L
⑥	(`rep mov` + store, store + fence + load)	store (to load)	🔒 / ☝ / <✖> / ✓	-
⑦	-	store (4K Aliasing) + 🔒 / ☝ / ⊘ / <✖> / ✓	🔒 / ☝	MSBDS
⑧	-	store (4K Aliasing, to load) + 🔒 / ☝ / ⊘ / <✖> / ✓	AVX ⇉ + 🔒 / ☝ / ⊘ / <✖> / ✓	MSBDS, S2L
⑨	(Sibling on/off)	store (random address) + ⊘	🔒 / <✖>	MSBDS
⑩	(Sibling on/off + clflush (store address))	store (Cache Offset of Load) + ⊘	🔒 / <✖>	MSBDS
⑪	(Sibling on/off + repmov (to Load))	store (to Load)	AVX ⇉ + 🔒 / ☝ / ⊘ / <✖> / ✓	Medusa, MLPDS
⑫	-	Store (Unaligned to Load)	🔒 / ☝ / <✖>	Medusa
⑬	(random instructions)	AVX Store (to Load)	<✖>	Medusa, MLPDS, MSBDS
⑭	-	random fill stores	<✖>	MSBDS

<✖> Non-canonical Address Fault ⊘ Non-present Page Fault 🔒 Supervisor Protection Fault ⇉ AVX Alignment Fault
☝ Access-bit Assist ≺ Split-Cache Access Assist ✓ Access without fault or Assist

AVX loads. However, when the load suffers a vector alignment general-protection exception, Store-to-leak and Fallout both ignore the address types for both stores and loads (⑧).

Store-Forwarding and Faulting Stores. Transynther discovered that faulting stores can be forwarded independently of address aliasing and matching. In ⑨, we perform a store to non-present addresses causing a page fault, e.g., a null address. When the sibling thread is turned on and off, the store is forwarded to the faulting load without any aliasing. Interestingly, we can still index over which byte of the store to be leaked. This variant of MSBDS only works with supervisor fault and non-canonical address exceptions.

Store Forwarding and Cache Aliasing. Transynther also created code sequences that leak the store data based on aliasing of only the cache offset. This is in contrast to the current understanding that only full address matching or 4 kB aliasing forwards the data (⑩).

Store Forwarding and Stale Load Forwarding. As we mentioned in various cases, grooming the pipeline may affect which data will be forwarded/leaked first. For instance, Transynther generated a multitude of proof of concepts that different types of buffers and values can be leaked with vector alignment exception. We only mention one example here that, Store-to-Leak can be turned into to a case where both the store, and a value from the sibling thread (MLPDS or Medusa) are leaked. In this case, we prepared the architecture with a `rep mov` instruction with the destination address being the faulty load address. When the sibling thread is switching on/off, we see that both the forwarded store and the values loaded by the sibling thread are leaked (⑪).

In this proof-of-concept, `rep mov` which is handled by a specific microcode assist [26], is causing the value from a sibling thread to be loaded instead of the expected store-forwarding, *i.e.*, the value stored previously. We investigated the effect of `rep mov` and found out that we can use it to create a new variant of leakage from the WC buffer (Section 4.2.3).

Unaligned Store Forwarding. We also found using Transynther that unaligned store forwardings can leak values from a sibling thread. This is a special case of store-forward in which the store and load overlap partially, but the actual data bytes on the store can not be forwarded to the load. We investigate this case further and use it as a new attack variant for Medusa in Section 4.2.2 (⑫).

Non-canonical Addresses. Non-canonical addresses are handled differently from regular memory addresses on Intel CPUs [55]. During an early stage of address decoding, the processor converts a 64-bit address to a compacted form, as the actual supported address space is not 64-bit. During this conversion, if the address does not follow the canonical form [27], a general-purpose exception will be thrown. We also verified that there is no page table walk for non-canonical addresses and an early mechanism throw an exception matching the description in the patent.

Medusa observed various cases where the combination of non-canonical address faults will leak data with a different behavior. For instance, store-to-leak on a no-canonical address may not always leak the value of the store. Instead, depending on specific grooming of the architecture, we see that both the store and loads from the sibling thread are leaked (⑬). Another interesting observation is that in certain cases for the store buffer, a non-canonical fault would always leak the last store disregarding any type of aliasing. In this case, we have filled the store buffer with various valid stores, and depending on what state the store buffer will be (a different set of random stores), there are cases where the last store will always be forwarded to the load (⑭).

Transactional Asynchronous Abort (TAA). The Transactional Asynchronous Abort (TAA) [25] represents another vulnerability allowing to leak data from the same microarchitectural buffers as MDS. TSX transactions can be aborted by data conflicts, resource exhaustion, certain instructions, synchronous exception events, e.g., page faults, or asynchronous events within the pipeline [27].

We recorded the performance counters statistics of different variants of Medusa, ZombieLoad [52] and RIDL [58] in Appendix B. We observed that only Variant 2 of ZombieLoad [52] exploits TAA by actively inducing asynchronous aborts as shown by the high number of the `tx_mem.abort_conflict` counter. Other variants that use TSX for exception suppression only show synchronous aborts and hence do not exploit TAA. However, in the rare case that an unrelated event, such as an interrupt or cache eviction, asynchronously aborts the transaction during the load, these variants could also trigger TAA.

3.4.2 AMD

Exception Bypass. One of the requirements for Meltdown-type attacks is to bypass exceptions in an out-of-order fashion. The results from Transynther suggest that the AMD Zen microarchitecture might potentially be vulnerable to Meltdown-type attacks. We found that various exceptions, such as division by zero, an aligned vector store general-purpose exception, as well as a faulting store to a supervisor address, do not stop the out-of-order execution. In line with the AMD whitepaper [3], some of the exceptions are bypassed speculatively. Hence, an important requirement for Meltdown is also present on AMD CPUs, the forwarding of data from faulting instructions. CPUs immune to Meltdown-type attacks have to ensure that operations depending on a faulting instruction cannot get the transient data, e.g., by stalling. While AMD ensures that for page faults, they do not ensure that property for other faulting instructions, e.g., General Protection Memory Access (cf. AMD whitepaper [3], page 5). While we could not show data leakage that violates a security guarantee, e.g., leakage from the kernel, AMD is not by-design immune to the root cause of Meltdown-type attacks.

Vector Move Alignment Fault. We also observed that the faulty vector alignment exceptions are handled differently than other faulty loads. In particular, these exceptions do not block the data flow, and we observe that the pipeline will still speculatively consume the data despite the exception. We observe that the value of the memory page or the value that is written recently to the memory page will be leaked using a Meltdown-style gadget. Again, this does not violate any architectural data flow, but from a microarchitectural standpoint, it shows that computation over transient data that was not supposed to be available is feasible.

3.5 Meltdown Root Cause Generalisation

From the vast amount of results generated by Transynther, we can generalize the common root cause of known Meltdown attacks. As stated by Canella et al. [11], the leakage for all known Meltdown attacks is caused by a faulting load, where microcode assists are considered as microarchitectural faults [52]. In all attacks, we see the same behavior, that the faulting load does not stall and thus cannot simply return no data. As a consequence, the faulting load transiently returns data that can be accessed immediately and where at least parts of the address match.

The microarchitectural element from which an attack leaks depends on the microarchitectural implementation of data-forwarding checks, and where the fault occurs. For example, ZombieLoad and Fallout exploit the same fault as the original Meltdown attack, and RIDL exploits the same fault as Foreshadow. In the case of RIDL and Foreshadow, it is the cleared present bit in the page-table entry of the load target. In the case that the L1 cache contains data with an address that matches the page-frame number, the load simply takes this value. This case is known as Foreshadow or Meltdown-P-L1 [11]. If this is not the case, e.g., because the page-frame number is 0 in the case of a NULL-pointer, the next possibility for data with partial address matches is the line-fill buffer. This case is known as RIDL or Meltdown-P-LFB [11]. Similarly, for Meltdown, ZombieLoad, and Foreshadow, where the user-accessible-bit in the page-table entry is exploited. First, the store buffer is checked in parallel with the L1 data cache. If a store-buffer entry has a partial address match, the faulting load consumes this data, which is known as Fallout or Meltdown-US-SB [10]. Otherwise, if the cache can provide data with partially matching addresses, this is considered as Meltdown-US [11]. In case the L1 cache cannot satisfy the request due to a cache miss or a cache-line conflict, the line-fill buffer can provide the data, resulting in ZombieLoad [52] or Meltdown-US-LFB [11].

Hence, one of the insights from Transynther is that the type of the fault is less important than where the fault occurs, *i.e.*, which microarchitectural element is the "closest" to the fault from which the faulting load can consume data.

4 Medusa: Pre-filtering Data

In this section, we further evaluate a novel ZombieLoad variant, which we discovered using Transynther. First, we show that Medusa allows prefiltering leaked values. Medusa only leaks values used in implicit WC by exploiting the microarchitectural implementation of the WC buffer. Second, we show 3 different variants of Medusa, which each have unique properties. Finally, we analyze potential attack targets for Medusa based on where implicit WC is used in real-world software.

4.1 Leakage Analysis

To evaluate the practicality of Medusa, we first analyze the leakage of Medusa. This includes the source of the leakage as well as the leakage pattern, *i.e.*, how much control an attacker has over the leakage and how much noise is in the leaked data. We first reduced the generated snippet, *i.e.*, we removed instructions as long as the leakage was still visible.

4.1.1 Leakage Source

For the leakage source, Transynther already provides an educated guess that the leakage source of the snippet is the fill buffer. For Medusa, Transynther reports a Pearson coefficient of $r_p = 0.99$ for the fill buffer, while the correlation for the other performance counters is not statistically significant. However, the only leaked value is the character written with `rep stosb`. Hence, in contrast to ZombieLoad [52], Medusa can only leak from a part of the line-fill buffer.

We additionally verify that using the publicly available proof-of-concept for ZombieLoad. Using this victim, we do not see any leakage when using Medusa, while we see a strong leakage when using the ZombieLoad attack. We also used the public proof-of-concept for RIDL [58]. Interestingly, RIDL only works when reading data after a flush and a memory barrier. If either the flush or the memory barrier (*i.e.*, `cpuid` or `mfence`) is missing, we do not get any leakage.

In Table 3, we compare different victims and whether different variant of MDS attacks (ZombieLoad, RIDL, Fallout) or Medusa can leak data from these victims. While data larger than 128 bits, e.g., `rep mov`, can also be leaked with ZombieLoad (same and cross hyperthread) or Fallout (same hyperthread), Medusa only leaks data larger than 128 bits. Hence, while Medusa does not exploit any new data source, it targets exactly one type of victim, and there is no unrelated data from other processes.

WC and Fill Buffer. According to Intel, their microarchitectures use the line-fill buffer as WC buffers [23]. Thus, officially, 10 line-fill-buffer entries can be used for WC [27]. Schwarz et al. [52] experimentally verified this for pre-Skylake microarchitectures but detected 12 line-fill-buffer entries since Skylake. We devised several experiments to analyze the WC-behavior of the line-fill buffer for all memory types supported on x86_64 (cf. Appendix A).

Implicit WC. While there is an explicit WC memory type, there are certain instructions that always use WC independent of the underlying memory type, e.g., non-temporal stores. Depending on the CPU, Intel also documents that non-temporal loads (`MOVNTDQA`) may reduce the number of cache evictions by leveraging the WC buffer [27]. Recent Intel CPUs support fast-string operations via the `rep mov` and `rep stos` instructions [26, 27]. These instructions do not guarantee any order of the written data [27]. Hence, they can employ WC to reduce the number of write requests sent on the memory bus.

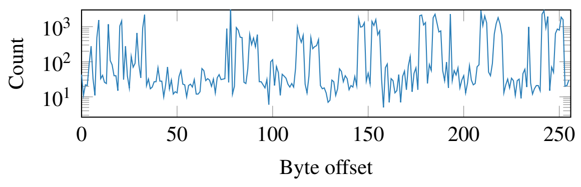

Figure 3: Leaking values with Medusa when copying a 256-byte buffer using `rep mov` shows an interesting pattern. While all bytes can be leaked, certain offsets in the buffer have a much higher probability of being leaked.

We verified that with Medusa, we can leak the values both for explicit WC, *i.e.*, memory marked as WC, as well as implicit WC, *i.e.*, `MOVNTDQA`, `rep mov`, and `rep stos`. Hence, Medusa has the unique property among all MDS attacks that the leakage is filtered by instruction types, *i.e.*, the amount of unrelated data is significantly less than in other attacks.

4.1.2 Leakage Pattern

Figure 3 shows the leakage pattern for Medusa when copying a 256-byte buffer in the victim application using `rep mov` over the time of 10 s. It can be seen that while not all offsets in a 256-byte window can be leaked with the same frequency, all offsets can be leaked. For the victim, we use a de Bruijn sequence of order 3 on an alphabet of size 26, *i.e.*, $B(26,3)$, to groom the WC buffer (cf. Section 3.3). We constantly write this sequence to a dummy location using `rep mov`. The victim is running on the sibling logical core.

For the attacker, we always leak 3 bytes at a time by encoding every byte into a different array of 256 pages. As it is possible to compute on the full leaked values in the transient domain [39, 52], we can leak a 32-bit value, split it, and encode it to different arrays. The recovered 3-byte value can then be matched to the de Bruijn sequence used in the victim application. As the position of every 3-byte value within the de Bruijn sequence is unique, this method allows us to analyze the pattern of the leaked values. Notably, we can always see strides of values which occur often in the leaked data, followed by strides which only occur rarely. Especially for the beginning of the buffer, the probability for leaking the first 32 bytes ($p = 67\,\%$, $n = 10\,000$) is significantly higher than for leaking the second 32 bytes ($p = 33\,\%$, $n = 10\,000$). We assume that the split of 32 B is due to the 32 B data-bus size on our test machine (i7-8650U). Hence, to transmit a WC-buffer entry over the common data bus, both halves of the entry have to be transferred separately, and Medusa leaks either the first or second half. Data after the first cache line shows a different pattern. We can always see 16 B strides of values that occur often in the leaked data, followed by 16 B strides, which only occur rarely. Interestingly, this pattern does neither correlate with the bus size, nor with the size of the WC buffer. Moreover, the leakage rate increases after

Table 3: A comparison of MDS attacks in various variants and on different targets.

	With memory barrier		Without memory barrier		>128-bit data	
	load	store	load	store	load	store
⊘	RIDL	RIDL	RIDL (ST)	RIDL (ST)	-	-
<✖>	-	Fallout (ST)	-	Fallout (ST)	-	**Medusa** / Fallout (ST)
🔒	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	**Medusa** / ZombieLoad
TAA	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad
PTE inversion	-	Fallout (CL, ST)	-	Fallout (CL, ST)	-	Fallout (CL, ST)
👆	ZombieLoad	ZombieLoad / Fallout (ST)	ZombieLoad	ZombieLoad / Fallout (ST)	ZombieLoad	ZombieLoad / Fallout (ST)
Attack(s)	ZombieLoad / RIDL	ZombieLoad / RIDL / Fallout (ST)	ZombieLoad / RIDL (ST)	ZombieLoad / RIDL (ST) / Fallout (ST)	ZombieLoad	**Medusa** / ZombieLoad / Fallout (ST)

ST Same CPU thread only CL Coffee Lake only <✖> Non-canonical Address Fault ⊘ Non-present Page Fault 🔒 Supervisor Protection Fault 👆 Access-bit Assist

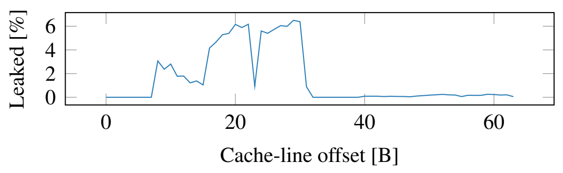

Figure 4: The cache-line offsets and how they contribute to the leakage for Medusa Variant I.

the first 64 B. At the time of writing, we do not know of any way to analyze these effects further, and hence, we leave the investigation of this effect for future work.

4.2 Exploitation Methodology

4.2.1 Variant I: Cache Indexing

We now describe different variants that allow triggering Medusa. In the first variant of Medusa, we rely on faulting loads which are bounded within a cache line. Variant I exploits faulting loads on addresses that point *inside* a cache line (cf. Figure 4) to leak values from the WC buffer. The setup is similar to all Meltdown-type attacks, with a faulting load that transiently encodes the loaded data into a microarchitectural element. In contrast to existing attacks, the type of fault is not important, but the cache-line offset of the faulting address is. We verified Variant I with both non-canonical and supervisor addresses. On our test machine, an i7-8650U, the cache-line offset, *i.e.*, the least-significant 6 bits of the address, has to be at least 8, which is the maximum size of normal memory loads. However, the highest leakage rates are for offsets between 16 and 31. The common data bus has a width of 32 bytes. However, normal loads can only use up to 8, and AVX loads 16 bytes (128 bits). As a consequence, offsets 16 to 31 are rarely used, as only AVX2 (256 bits) uses the full width of the common data bus. However, as the goal of WC is to increase the throughput, (implicit) WC also tries to leverage the entire common data bus. Hence, by using address offsets that index the upper half of the common data bus, Variant I leaks stale values of recent WC operations, e.g., `rep mov`, as well as AVX2 memory loads.

While at first, Variant I appears to be similar to MLPDS [24], ZombieLoad [52], or Fallout [10], it has distinctive properties. First, MLPDS requires either a faulting load spanning a cache line (64 B) or a faulting vector load that is larger than 64 bits [24]. For Variant I, neither of these requirements is necessary. In contrast, Variant I only works if the load is within one cache line. Loads spanning over two cache lines do not show data leakage (cf. Figure 4). Second, Variant I leaks data from the same logical core as well as from the sibling logical core, which is different from Fallout [10]. The leakage is limited to data that is stored using either `rep mov`, `rep stos`, or AVX2. In contrast to ZombieLoad or Fallout, Variant I of Medusa is agnostic to other data passing the store buffer or fill buffer, as they never use the upper half of the common data bus.

4.2.2 Variant II: Unaligned Store-to-Load Forwarding

A faulting or assisting load that meets the "Unaligned Store-to-Load Forwarding" condition (similar to MSBDS) consistently leaks stale data. This was observed even across hyperthreads. Note that this is different from MSBDS, as MSBDS does not work across hyperthreads. Here, we can leak the data from the WC buffer by creating an unaligned store-to-load forwarding condition on a faulting or assisting load. Further, an attacker can control which bytes of the WC buffer to leak by combining various load sizes and the offset of the small store. In our experiments, we can control the last 16 bytes of a WC buffer line by combining a 32-byte read 'ymmX' and iterating over various values for the offset of the store.

4.2.3 Variant III: Shadow `REP MOV`

Variant III of Medusa exploits a microcode assist caused by a `rep mov` followed by a dependent faulting load. The `rep mov` copies a single dummy byte to a destination address which causes a fault, e.g., a non-canonical address. A subsequent load from the destination address leaks data from a stale or concurrent `rep mov`. The `rep mov` can either be on the same

Table 4: `rep mov` instruction within cryptographic libraries.

Library	Version	O0	O1	O2	O3	Os
Botan	2.11.0	12	14	68	*137	188
Openssl	1.1.1c	12	23	29	*34	347
Wolfssl	4.1.0	1	7	*49	72	199
Bearssl	0.6	10	26	45	56	*213
Sodium	1.0.18	3	12	*12	13	49
Gcrypt	1.8.4	5	5	*7	11	168

logical core before running Medusa which leaks stale data of the previous `rep mov`. This also works across privilege boundaries, *i.e.*, the stale `rep mov` data can also be from the kernel. Moreover, this attack also works for a concurrent `rep mov` on the sibling logical core across privilege boundaries.

As with Variant I, this variant has the property to only leak data of `rep mov`, `rep stos`, and AVX2 memory loads, which allows a targeted leakage of data used in such constructs. In contrast to Variant I, this variant is entirely address-agnostic, which simplifies the recording of the leakage. However, this increases the complexity of the post-processing, as an attacker does not have any control over the index of the leaked data. Hence, as every byte of the victim buffer can be leaked with a certain probability, the postprocessing has to stitch together the leaked data, e.g., using the Domino technique [52].

4.3 WC in Real-World Software

We analyzed real-world software to find occurrences of WC. We looked both for explicit WC, *i.e.*, WC memory defined through the PAT, as well as for implicit WC in the form of `rep mov` and `rep stos`.

userspace. We first searched for implicit WC, as userspace applications cannot directly change the memory type of a page. We analyzed when and how often GCC emits a `rep mov` sequence during the compilation of popular cryptographic libraries, as potential targets that process sensitive information. As shown in Table 4, if GCC optimizes the application for code size (`-Os`), it emits the most `rep mov` instructions as `rep mov` is the smallest possible code sequence that can be used to copy memory regions. Similarly, `rep stos` is the smallest code sequence to initialize memory with a defined value.

We also found the explicit use of WC memory types in the userspace. Although implementation-specific, both OpenGL and Vulkan support memory buffers, which are marked as WC. Memory buffers allocated as write-only buffers are likely to be allocated as WC memory by the driver.

Linux Kernel. The Linux kernel also relies on `rep mov` to copy data. In contrast to user-space applications, the usage of `rep mov` is not to optimize the kernel binary for size. It is used independently of the used compilation flags, as the kernel generally does not use floating-point or SIMD operations. Hence, `rep mov` is the most efficient way to copy data. As there is a small startup penalty when using `rep mov`, only strings with a minimum length of 64 B are copied using `rep mov`. For shorter strings, or if fast-string operations are not supported, the kernel falls back to a simple copy loop. We reverse-engineered the kernel binary for kernel 5.0.0 shipped with Ubuntu to analyze it for the usage of `rep mov`. We found 517 usages of `rep mov` in 374 functions in the binary. While many of the functions are only used once in the setup phase of the kernel (e.g., to copy and decompress parts of the kernel, setup EFI and several devices, initialize the architecture, or apply microcode updates), some of them are used regularly. These functions include, amongst others, `memcpy`, `memmove`, `copy_from_user`, and `copy_to_user`.

4.4 Performance Evaluation

We evaluated the performance of Medusa based on our proof-of-concept implementations.

Environments. We evaluated all variants of Medusa on our Intel CPUs mentioned before. All environments run Ubuntu with a recent 5.0 kernel version. For CPUs vulnerable to Meltdown, the KPTI software mitigation is enabled. We successfully used all variants in all tested environments.

Performance. To evaluate the performance, we evaluate the leakage rate as well as the false-positive rate when using Medusa on a colluding victim. This provides an upper bound for the leakage rates we can expect when using Medusa in a side-channel attack where the victim is not colluding. We started a victim application on one logical core, which leaks a known value. On the sibling hyperthread, we ran Medusa repeatedly for 2 s and recorded the correctly and incorrectly leaked values. With variant I, we achieve an average leakage rate of 0.19 kB/s ($n = 100$, $\sigma_{\bar{x}} = 0.0023$), with a false-positive rate of 47.7 % ($n = 100$, $\sigma_{\bar{x}} = 0.002$). For variant II, the leakage rate is on average 36.23 kB/s ($n = 100$, $\sigma_{\bar{x}} = 0.15$) with a false-positive rate of 0.559 % ($n = 100$, $\sigma_{\bar{x}} = 0.0005$). Finally, with variant III, we achieve an average leakage rate of 0.13 kB/s ($n = 100$, $\sigma_{\bar{x}} = 0.0016$) and a false-positive rate of 3.91 % ($n = 100$, $\sigma_{\bar{x}} = 0.0017$).

These numbers are based on our unoptimized proof-of-concept implementation. Hence, these numbers cannot be taken as upper bounds for the leakage rate (and false-positive rate). As we discussed in Section 3.4, these leakages can be improved by syntehsizing the implementation.

4.5 Cross-VM Covert Channel

To evaluate the leakage rate of Medusa in the cross-VM scenario, we evaluate the performance of a cross-VM covert channel. While the covert channel can also be mounted between user applications, we focus on the cross-VM case as it is the most restricted scenario. For our setup, we use two co-located VMs running on an Intel Core i7-8650U running Ubuntu 18.04.3. Both VMs are running Ubuntu 18.04.3.

Sender. For the sender, we use a `rep mov` instruction, which continuously copies a 256-byte buffer containing the encoded data. We redundantly encode every 32-bit data packet by repeating it 32 times inside the buffer. Every 32-bit data packet consists of 8-bit data, 8-bit checksum, a constant prefix, and a sequence number. The data-packet format resembles the setup from Schwarz et al. [52] to make the results comparable.

Receiver. The receiving application leverages Medusa variant III to leak victim data. Although the redundancy in the leaked data reduces the speed, it increases the robustness, as any part of the leaked buffer contains the data. Moreover, due to the checksum, which we can already verify during the transient execution [52], we do not receive any unrelated data, making the receiver robust against any system noise.

Results. We observed an average transmission rate of 14.3 B/s ($n = 1000$, $\sigma_{\bar{x}} = 0.56$) in the cross-VM scenario. In all cases, the transmission was error-free. Due to the overhead of the encoding scheme, the performance is significantly slower than the raw performance of Medusa variant III (cf. Section 4.4). We expect that more sophisticated encoding schemes, including error correction [43], can significantly improve the performance of the covert channel.

5 Attack Case Studies

In this section, we demonstrate the practicality of Medusa by extracting an RSA key from OpenSSL and by leaking kernel data transfers.

5.1 Leaking RSA Keys from OpenSSL

We use Medusa to demonstrate an attack on the latest OpenSSL that successfully recovers an RSA key. We focus on OpenSSL 1.1.1c, as it is both widely used, and it supports countermeasures against traditional side-channel attacks, making it a robust target. Note that while we quantified the occurrence of `rep mov` in popular cryptographic libraries (cf. Section 4.3), we did not analyze further for potential security-critical use cases. However, we expect that they are vulnerable to similar attacks as well. The victim is a simple application that leverages OpenSSL to load an RSA key from a file and signs some data using this key. This application reflects real-world command line or server applications that are spawned upon user request to perform a cryptographic task, e.g., SSH client/server or VPN client/server. In our attack, we can start the application arbitrarily, but we do not control any inputs to the victim application. This scenario, *i.e.*, triggering the victim application, is in line with previous research [10, 52, 58, 63].

Every time the application is started, it has to load the RSA private key from the key file. The key file is in the PEM format, which is a `base64` encoded representation of the key parameters. Hence, to use the actual key parameters, OpenSSL first decodes the key data using its internal `base64` decoder.

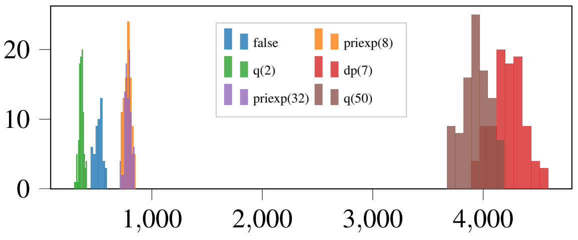

Figure 5: Histogram and score of most likely 6-byte leakages through AVX256-P3 with 10K observations collected in 100 runs (labeled by starting bytes). Six byte block leakages at `q(2)` (q starting at byte 2), `priexp(8)` and `priexp(32)` (RSA exponent d starting at bytes and 8 and 32) and `dp(7)` (leak from $d_p = d \bmod p-1$ starting at byte 7) can be easily identified based on the observation frequencies.

When compiling the library to optimize it for size, the `base64` decoder uses `rep mov` for loading the `base64`-encoded data. We attack exactly this `rep mov` sequence using Medusa to leak the RSA parameters, which are then used to recover the private key.

OpenSSL RSA keys in PEM format include both the default prime and exponents of the RSA alongside the precomputed parameters for the Chinese Remainder Theorem (CRT). This includes modulus N, public exponent e, private exponent d, prime numbers p and q, $d \bmod (p-1)$, $d \bmod (q-1)$ and the coefficient $q^{-1} \bmod p$. The size of the copy operation during the execution of the `rep mov` instruction depends on the key size. For example, for a 1024-bit RSA key, there are $5*64+2*128 = 576$ bytes of key material to be copied. As the key material also includes several bytes for the ASN.1 PEM metadata, the total amount of copied raw data is approximately 600 bytes. As the data is `base64` encoded, which always encodes 3 raw bytes as 4 bytes, the actual amount of copied data is approximately 800 bytes. Hence, depending on the size of the copy operation and the used attack, different parts of this key may be leaked more often (cf. Figure 3).

We create a template based on the frequency of the leakage of different parts of the RSA key parameters. In this attack, we use variant II of Medusa to leak the data with the unaligned store forwarding, which allows us to leak the entire content of the common data bus. We also use the domino technique [52] combined with the frequency of each observed value to build a frequency template of recovered key parts. As discussed in Section 4.1, the probability of leaking specific data depends on the offset of the leaked data transmitted over the common data bus. Hence, depending on which part of the data we want to leak, we have to repeat Medusa between 10 000 and 20 000 times per key byte. In total, we run this experiment 100 times. Our online phase of the attack takes at most 7 minutes on the core i9-9900K CPU.

After stitching the bytes of every 8-byte block of `base64`-encoded data using the Domino technique [52], we can create

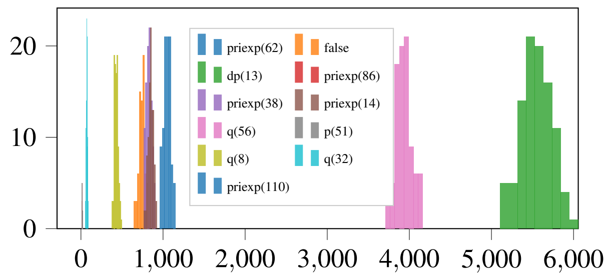

Figure 6: Histogram and score of most likely 6-byte leakages through AVX256-P4 (similar experiment as Figure 5). Block leakages at `q(8)`, `q(32)`,`q(56)` (q starting at bytes 8, 32, 56), `priexp(14)`, `priexp(39)` and `priexp(86)`, `dp(13)` (leak from d_p starting at byte 13), `p(51)` (p starting at byte 51) can be identified based on the block frequencies.

a template based on the frequency of an observed block that tells us which parts of the key material are leaked. Note that each 8-bytes block of `base64` encoded key data holds 6-bytes of valuable raw key material. Figure 5 and Figure 6 show the frequency of each section leaked through different part of an AVX-256 register. Note that in the top histogram we see consistent strong leakage of 6-byte blocks in `priexp` (the RSA key d), starting at byte locations $14, 38, 86$, and 110 as well as strong leakage in q starting at locations $8, 32, 56$.

5.1.1 Recovering full RSA keys using Lattice Attacks

These leakages give us only **partial** information on the RSA secrets p, q, d (`privexp` in the OpenSSL implementation), and $d \bmod (p-1)$, $d \bmod (q-1)$ and the coefficient $q^{-1} \bmod p$ are far from yielding the full secrets. However, there has been significant progress in recovering keys from RSA instantiated with small or partially exposed messages, or decryption keys. Coppersmith introduced a technique for finding small roots of polynomial equations is to reduce the problem of finding roots of a polynomial $f(x)$ over $\mathbb{Z}_p$ [13], which may be used to recover RSA factors, if the least or most-significant half of the bits of p or q are known. Boneh, Durfee, and Frankel proposed a technique to recover the RSA secret and moduli p and q if a quarter of the least or most significant bits of d are leaked, and when e is small enough to be reachable via exhaustive testing [8]. Later Boneh and Durfee [7] presented a technique that recovers RSA factors with $d < N^{0.292}$ without any conditions on e. For an overview, see May [44], and the more recent Takayasu and Kunihiro et al. [54]. Here we focus on two attacks which fit our leakage profile:

Coppersmith. We use the Coppersmith attack to recover the RSA factor q. We combine partial leakages of q at bytes 8, 56 (from P4), and 2, 50 (from P3) and 0, 61, 12, 44 (from P2) to obtain a leakage in q: 18-bytes LSB (bytes 0-17) and 20-bytes MSB (bytes 44-63). This gives us a combined leakage of more than a quarter (38/128 bytes) of N for the 1024-bit RSA. Coppersmith's attack is slightly adjusted to handle the LSB/MSB split in the leaked data. We apply Coppersmith's lattice attack to recover small solutions to

$$f(x) = x + (q_{MSB} 2^{44\times 8} + q_{LSB})(1/2^{18\times 8} \bmod N) .$$

We used `SageMath v8.4` with `NTL` for LLL to implement the attack which takes a few second to successfully recover a root x_0 and the RSA factor: $q = q_{MSB} 2^{44\times 8} + x_0 2^{18\times 8} + q_{LSB}$. We attached *scores* by counting how many times the partial leakages could be stitched together into an 8-byte block over 20 000 samples. The scores serve as a template which we use to classify observations before trial by Coppersmith.

	ymmX-P2				ymmX-P3		ymmX-P4	
Block `q(i)`	44	12	61	0	2	50	8	56
Avg. Score	82	288	304	355	377	4157	401	3651
# Spurious	5	18	16	14	0	1	0	0

To obtain the statistics for the templates, we needed 20 000. With more spurious blocks (selected as to have a score within ±20 % of the target block), we need to try more combinations. On average, we need 58 000 trials and each triage of this lattice attack takes 25 seconds. As a result, in the offline phase of the attack, we use 400 CPU hours to perform these trials which is achieved in a day on our 16-core desktop CPU.

Boneh, Durfee, Frankel (BDF). While the Coppersmith and partial information of the q was sufficient to recover the RSA key, we discuss an alternative attack for potential other templates. The BDF attack [8] recovers RSA factors given the LSB quarter of the secret exponent d bits when e is small enough to be exhaustively tested. The attack iterates the following steps for each $k \in [1, e]$ until a solution is found:

1. Form a polynomial equation:
   $$f(x) = kx^2 + (ed_0 - k(N+1) - 1)x - kN = 0 \pmod{2^{n/4}} .$$
   Here $n = \log_2(N)$ and $d_0 = d \pmod{2^{n/4}}$.
2. Find solutions to $f(x)$. Due to the special structure of the modulus, the equation is efficiently solved to recover at most 2^{t+1} solutions, where t is the largest power of 2 that divides k. For correctly chosen k the solution of $f(x)$ yields p (or q) modulo $2^{n/4}$.
3. Check each recovered solution by taking it as the (candidate) LSB of p or q and running Coppersmith to see if we obtain the RSA factors.

The algorithm runtime is $O(e \log(e))$ Coppersmith iterations.

A Small but Effective Optimization. Our target $e = 2^{16} + 1$ is exhaustible. However, we can do much better since we have some LSB bytes of p and q. We can use these bytes to check the recovered candidate LSBs of p or q and take a shortcut omitting costly Step 3 if there is no match. With a few bytes of leakage, we can reduce the complexity from $O(e \log(e))$ to only $O(\log(e))$ Coppersmith evaluations.

For the 1024-bit case, we exploit the leakage observed on d (`priexp`) with 6-byte leakages starting at bytes: $2, 8, 14, 16, 26$ which gives us 27 LSB of the required 32 bytes of d. We are missing 5 bytes which are now exhaustible. The attack requires about 180 trials to cope with the spurious blocks.

	ymmX-P2			ymmX-P3	ymmX-P4
Block $d(i)$	2	16	26	8	14
Avg. Score	116	104	138	739	724
# Spurious	9	8	0	1	0

Scaling the Attack to 2048-bit RSA. The 1024-bit RSA attack described above recovered the secret key using a simple univariate formulation via Coppersmith's technique since a quarter of contiguous secret bits were available. For a 2048-bit key, this is more challenging, since we can not obtain 64 contiguous bytes of q, p or d through the leakage channel. However, we have observed more leakage from the higher blocks of d and non-contiguous blocks of p and q. The main idea is to form *multivariate* expressions of the form $f_i(x,y)$ using the known parts of d, p, and q where x and y represent the unknown parts of p and q. Then we apply lattice reduction to reduce the size of the coefficients. A *resultant* computation applied on the reduced multivariate polynomials yields a univariate polynomial, whose solution yields the unknown parts of p or q. The success probability for the attack depends on the amount of leakage and the precise lattice formulation. While plausible, this approach is beyond the scope of this paper. For further information on multivariate analysis see [6, 15].

5.2 Leaking Kernel Data Transfers

As discussed in Section 4.3, the Linux kernel uses `rep mov` for the internal data-transfer functions, including `memcpy`, `memmove`, `copy_from_user`, and `copy_to_user`.

Root Password Hash. As described by Van Schaik et al. [58], the unprivileged `passwd -S` command reads the contents of the user-inaccessible `/etc/shadow` file containing the password hashes of local users. They managed to leak 21 B in 24 h using the RIDL attack. Schwarz [49] showed that the same attack is more efficient with ZombieLoad by leaking 16 B in 1.25 min. With TAA [52], the entire hash can even be leaked within seconds [14].

We used Medusa to reproduced this attack. While we can also leak the root password hash with Medusa, the leakage rate depends on the part of the password hash that is leaked. Due to the leakage pattern of Medusa, we always have blocks of the hash that can be leaked within 1 s while for other blocks, it takes up to 1 h, which is comparable to the proofs-of-concept shown for ZombieLoad and RIDL.

File I/O. Generally, Medusa can leak any data transfer between the kernel and the userspace, such as the contents of files when reading or writing them. We verified that we can leak the content by using a file with known contents. We continuously read the file from one application running on one hyperthread, while running Medusa in a different user-space application on the sibling hyperthread. As every file read is handled by the kernel via the `read` syscall, the entire file content is copied from the kernel to the user-space victim application. On average, we leaked 12.3 B/s of correct values from the file.

Another case of data transfer is swapping. If application pages are copied to or from the swap device, the data can potentially also be leaked using Medusa.

6 Countermeasures

As Medusa is a variant of ZombieLoad, the same countermeasures are applicable for both Medusa and ZombieLoad.

Hyperthreading. While Intel claims that hyperthreading can be enabled if group scheduling is implemented [24], we are not aware of any commodity operating system implementing group scheduling. Hence, only disabling hyperthreading would entirely prevent cross-hyperthread attacks.

Flushing Buffers. To prevent the exploitation of MDS attacks, Intel released a microcode update that retrofits the `VERW` instruction with the side effect that it clears the store buffer, fill buffer, and load ports. While this prevents RIDL [58], Schwarz et al. [52] have shown that ZombieLoad can circumvent this mitigation. The only effective solution is to additionally flush the L1 data cache as well. However, flushing the store buffer, fill buffer, load ports, and L1 data cache on every privilege-level switch, e.g., context switch, incurs a non-negligible performance overhead.

New CPUs. Although new CPUs are MDS resistant, there are still variants of ZombieLoad which work on these CPUs by leveraging microcode assists caused by Intel TSX. Hence, even on MDS resistant CPUs, Intel TSX has to be disabled to ensure that no ZombieLoad variant, including Medusa, can leak any data. While Intel TSX cannot be disabled directly, a workaround is to ensure that all TSX transactions abort immediately by setting the `MSR_TSX_FORCE_ABORT` model-specific register. As a consequence, Intel TSX cannot be used for fault suppression any more.

7 Discussion

Other CPU Vendors. In this paper, we mainly focussed on Intel CPUs. While Medusa is a vulnerability we only discovered on Intel CPUs, the general approach of Transynther applies to different CPUs as well. We also used Transynther on AMD (cf. Section 3.4.2), showing that AMD also forwards data after certain exceptions, which is a requirement for Meltdown-type attacks. However, we could not find any variant on AMD that leaks data across a security boundary. Future work has to manually investigate whether the exception bypasses on AMD can lead to security vulnerabilities.

Transynther can also be applied to other microarchitectures, such as ARM or RISC-V. Although the approach is the same, porting Transynther to a different instruction set requires a new backend that generates assembly code for the targeted architecture. As our tool is open source, we encourage researchers to port Transynther to different architectures and analyze whether they suffer from similar vulnerabilities.

Non-Meltdown-type Vulnerabilities. The approach of Transynther is designed to automatically find Meltdown-type vulnerabilities. Other transient-execution attacks, such as Spectre-type attacks, are not in scope for Transynther. The reason is that Spectre attacks exploit the intentional, well-understood behavior of branch predictors. Every branch predictor can likely be abused for Spectre attacks [11], and the types of branch predictors are usually documented for every microarchitecture. Hence, we do not expect that Transynther would detect any new Spectre variants even when it is adapted for finding such attacks.

Meltdown-type attacks, however, exploit CPU vulnerabilities that can be triggered in multiple different ways. Hence, as this paper has also shown with Medusa, Transynther can discover new variants, and can potentially also help to find Meltdown-type attacks on different platforms.

In related work, Xiao et al. [62] analyzes both Meltdown- and Spectre-type vulnerabilities in terms of speculation window, triggers, and different covert channels. They also rely on templates to build code that is analyzed for vulnerabilities.

Starting Set Dependency. As most fuzzers, Transynther relies on a starting set for creating more test cases. The difference to software fuzzers is that Transynther does not have fine-grained feedback, such as e.g., code coverage. While traditional fuzzers can create test cases based on mutation and feedback, Transynther is mostly limited to random mutations. Hence, the better the starting set, *i.e.*, the more different variants are covered, the better the efficiency of Transynther. As with any fuzzer, there is no guarantee that Transynther finds all possible vulnerabilities.

Fuzzing-based Approaches. Fuzzing is a well-established technique for finding vulnerabilities across trust boundaries [9, 12, 16, 32, 33, 36, 41, 45, 51, 59, 60]. These approaches can usually rely on a well-defined interface, e.g., system calls.

SpecFuzz investigated the use of fuzzing for finding Spectre gadgets [47]. They apply fuzzing techniques to find Spectre-PHT (also known as Spectre Variant 1) gadgets in existing code. However, they do not try to find new attack variants. To the best of our knowledge, with Transynther, we are the first to show that fuzzing can be applied to detect microarchitectural vulnerabilities.

8 Conclusion

In this work, we performed an in-depth analysis of MDS attacks. We introduced a fuzzing-based analysis tool, named Transynther, which mutates the basic block of existing variants of Meltdown attacks to generate new subvariants. We analyzed a number of CPUs using Transynther to better understand variants of these attacks and found new variants of MDS that only target fast string copies. Based on our findings, we proposed a new attack named Medusa, which leaks data from WC memory operations. Since Medusa only attacks specific operations, it is more targeted. To demonstrate the effectiveness of Medusa, we ran several case studies: We recovered full RSA keys from OpenSSL by pooling leakages observed during key decoding, amplified using lattice techniques. Further, using Medusa we demonstrated how one can recover information from kernel data transfers, or leak the content of files.

Acknowledgments

We would like to thank our reviewers and especially our shepherd, Vasileios Kemerlis, for their suggestions that helped improving the paper. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). It was also supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. Moghimi and Sunar were supported by the National Science Foundation under grants no. CNS-1814406. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Advanced Micro Devices. Software Optimization Guide for AMD Family 17h Processors, 2017.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[3] AMD. Speculation Behavior in AMD Micro-Architectures, May 2019.

[4] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A Taint Based Approach for Smart Fuzzing. In *IEEE International Conference on Software Testing, Verification and Validation*, 2012.

[5] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way. In *International Conference on Cryptographic Hardware and Embedded Systems*, 2014.

[6] Johannes Blömer and Alexander May. New Partial Key Exposure Attacks on RSA. In *International Cryptology Conference (CRYPTO)*, 2003.

[7] Dan Boneh and Glenn Durfee. Cryptanalysis of RSA with Private Key d Less than N/sup 0.292. *IEEE transactions on Information Theory*, 2000.

[8] Dan Boneh, Glenn Durfee, and Yair Frankel. An Attack on RSA Given a Small Fraction of the Private Key Bits. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 1998.

[9] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security*, 2008.

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

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

[12] George J Carrette. CRASHME: Random Input Testing, 1996.

[13] Don Coppersmith. Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *Journal of Cryptology*, 1997.

[14] Finn de Ridder. https://www.youtube.com/watch?v=4DQAcCfg3b8, 2020.

[15] Matthias Ernst, Ellen Jochemsz, Alexander May, and Benne De Weger. Partial Key Exposure Attacks on RSA Up to Full Size Exponents. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2005.

[16] Amaury Gauthier, Clément Mazin, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Enhancing fuzzing technique for OKL4 syscalls testing. In *Sixth International Conference on Availability, Reliability and Security (ARES)*, 2011.

[17] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[18] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security Symposium*, 2017.

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

[20] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

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

[22] Intel. Write Combining Memory Implementation Guidelines, 1998.

[23] Intel. Copying Accelerated Video Decode Frame Buffers, 2015.

[24] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling, May 2019.

[25] Intel. Deep Dive: Intel Transactional Synchronization Extensions (Intel TSX) Asynchronous Abort, November 2019.

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

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

[28] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM Asia Conference on Computer and Communications Security*, 2016.

[29] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *International Workshop on Recent Advances in Intrusion Detection*, 2014.

[30] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium*, 2019.

[31] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[32] Moritz Jodeit and Martin Johns. USB Device Drivers: A Stepping Stone into Your Kernel. In *IEEE European Conference on Computer Network Defense*, 2010.

[33] Dave Jones. Trinity: A system call fuzzer. In *13th Ottawa Linux Symposium*, 2011.

[34] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757*, 2018.

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[36] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *IEEE Symposium on Reliable Distributed Systems*, 1997.

[37] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Workshop on Offensive Technologies*, 2018.

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

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

[40] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[41] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. esting System Virtual Machines. In *International Symposium on Software Testing and Analysis*, 2010.

[42] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing cpu emulators. In *International Symposium on Software Testing and Analysis*, 2009.

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

[44] Alexander May. *New RSA Vulnerabilities Using Lattice Reduction Methods*. PhD thesis, University of Paderborn Paderborn, 2003.

[45] Manuel Mendonça and Nuno Neves. Fuzzing Wi-Fi Drivers to Locate Security Vulnerabilities. In *IEEE European Dependable Computing Conference*.

[46] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. MemJam: A False Dependency Attack against Constant-time Crypto Implementations. In *International Journal of Parallel Programming*, 2019.

[47] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type Vulnerabilities to the Surface. *arXiv:1905.10311*, 2019.

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers' track at the RSA conference*, 2006.

[49] Michael Schwarz. https://twitter.com/misc0110/status/1129305720770498561, May 2019.

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

[51] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In *ACM Asia Conference on Computer and Communications Security*, 2018.

[52] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

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

[54] Atsushi Takayasu and Noboru Kunihiro. Partial Key Exposure Attacks on RSA: Achieving the Boneh-Durfee Bound. In *International Conference on Selected Areas in Cryptography*, 2014.

[55] Bret L Toll, John Alan Miller, and Michael A Fetterman. Method and Apparatus for Representation of an Address in Canonical Form, September 5 2006. US Patent 7,103,751.

[56] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdown-Prime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *arXiv:1802.03802*, 2018.

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

[58] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[59] Dmitry Vyukov. syzkaller - linux syscall fuzzer, 2016.

[60] Vincent M Weaver and Dave Jones. perf fuzzer: Targeted Fuzzing of the perf event open() System Call. Technical report, Technical Report, University of Maine, 2015.

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

[62] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *Network & Distributed System Security Symposium*, 2020.

[63] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX) Security 14)*, 2014.

[64] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to ExtractPrivate Keys. In *ACM conference on Computer and communications security*, 2012.

A WC Buffer Size

In this experiment, we determine the size of an entry in the WC buffer. The idea is to detect that there are no available WC-buffer entries anymore by relying on the `L1D_PEND_MISS.FB_FULL` performance counter. We execute an increasing number of non-temporal linear store instructions with a defined stride size. Non-temporal stores ensure that the CPU uses WC for the stores. At the point where the stride size exceeds the size of a WC-buffer entry, a new WC-buffer entry has to be allocated for every store. Hence, if we see that the WC buffer becomes a bottleneck, and the number of executed stores matches the number of fill-buffer entries, we know that the stride size equals the WC-buffer-entry size.

Figure 7 shows the results of this experiment. Only at a stride size of 64 bytes and for more than 12 stores, the performance counter reports unavailability of the WC buffers. For smaller stride sizes, the stores can be combined in the buffers such that not every store requires its buffer entry.

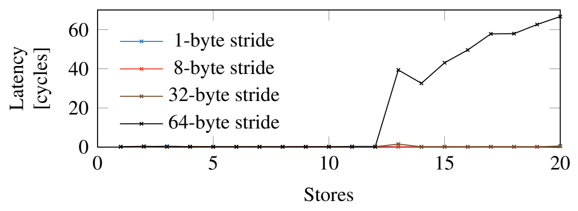

Figure 7: Cycles no fill-buffer entry is available. As Skylake has 12 fill-buffer entries [52] usable as WC-buffer entries [23], one has to be 64 bytes.

B Performance Counters

Figure 8 shows the heatmap for the correlation between the number of leaked bytes and different performance counter events, related to various variants of Meltdown attacks.

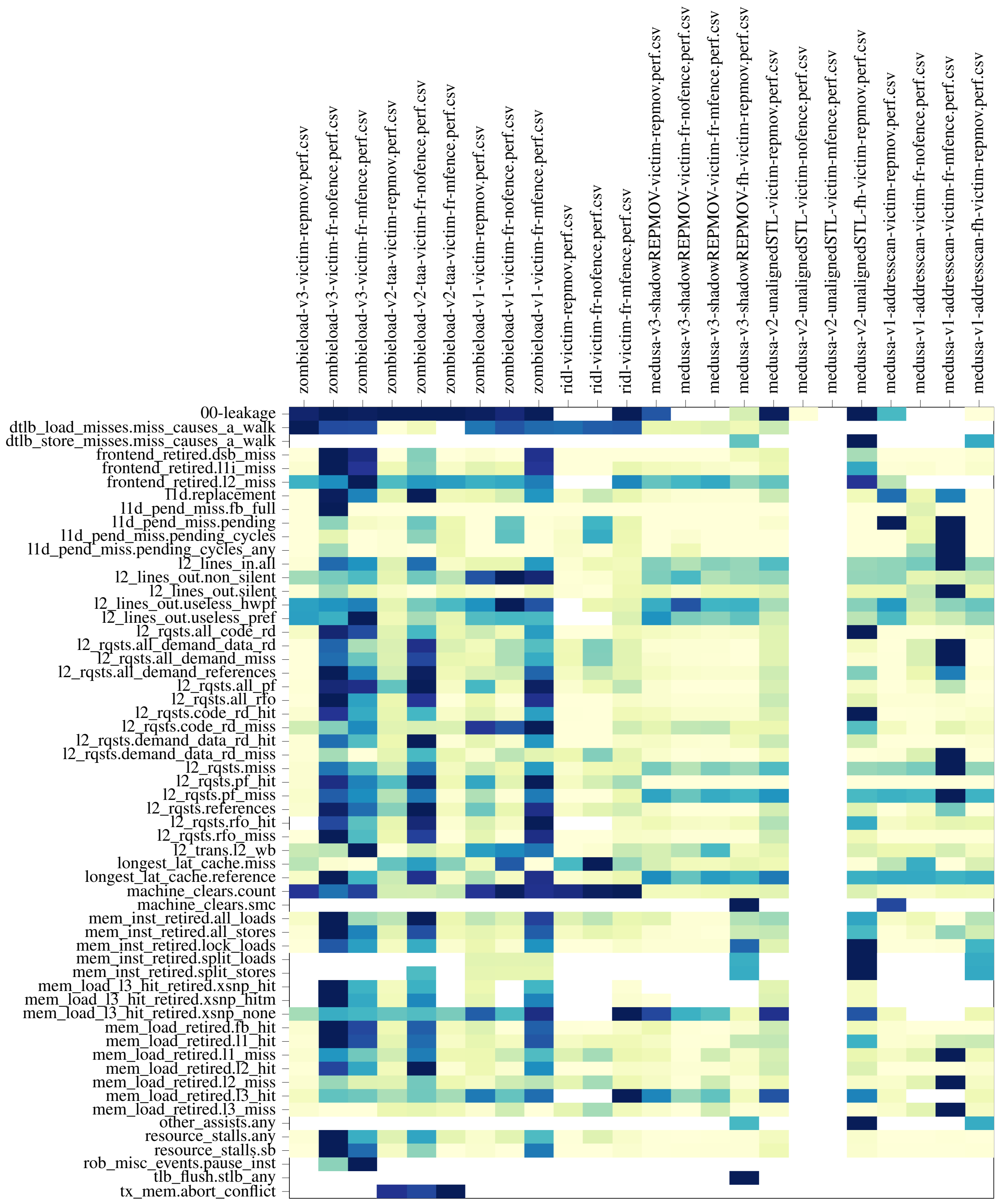

Figure 8: Heatmap of performance counters