Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors

Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, Daniel Gruss

Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors

Moritz Lipp
Graz University of Technology

Vedad Hadžić
Graz University of Technology

Michael Schwarz
Graz University of Technology

Arthur Perais
Unaffiliated

Clémentine Maurice
Univ Rennes, CNRS, IRISA

Daniel Gruss
Graz University of Technology

ABSTRACT

To optimize the energy consumption and performance of their CPUs, AMD introduced a way predictor for the L1-data (L1D) cache to predict in which cache way a certain address is located. Consequently, only this way is accessed, significantly reducing the power consumption of the processor.

In this paper, we are the first to exploit the cache way predictor. We reverse-engineered AMD's L1D cache way predictor in microarchitectures from 2011 to 2019, resulting in two new attack techniques. With Collide+Probe, an attacker can monitor a victim's memory accesses without knowledge of physical addresses or shared memory when time-sharing a logical core. With Load+Reload, we exploit the way predictor to obtain highly-accurate memory-access traces of victims on the same physical core. While Load+Reload relies on shared memory, it does not invalidate the cache line, allowing stealthier attacks that do not induce any last-level-cache evictions.

We evaluate our new side channel in different attack scenarios. We demonstrate a covert channel with up to 588.9 kB/s, which we also use in a Spectre attack to exfiltrate secret data from the kernel. Furthermore, we present a key-recovery attack from a vulnerable cryptographic implementation. We also show an entropy-reducing attack on ASLR of the kernel of a fully patched Linux system, the hypervisor, and our own address space from JavaScript. Finally, we propose countermeasures in software and hardware mitigating the presented attacks.

CCS CONCEPTS

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

ACM Reference Format:
Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), June 1–5, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3320269.3384746

ASIA CCS '20, June 1–5, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-6750-9/20/06…$15.00
https://doi.org/10.1145/3320269.3384746

1 INTRODUCTION

With caches, out-of-order execution, speculative execution, or simultaneous multithreading (SMT), modern processors are equipped with numerous features optimizing the system's throughput and power consumption. Despite their performance benefits, these optimizations are often not designed with a central focus on security properties. Hence, microarchitectural attacks have exploited these optimizations to undermine the system's security.

Cache attacks on cryptographic algorithms were the first microarchitectural attacks [12, 42, 59]. Osvik et al. [58] showed that an attacker can observe the cache state at the granularity of a cache set using Prime+Probe. Yarom et al. [82] proposed Flush+Reload, a technique that can observe victim activity at a cache-line granularity. Both Prime+Probe and Flush+Reload are generic techniques that allow implementing a variety of different attacks, e.g., on cryptographic algorithms [12, 15, 50, 54, 59, 63, 66, 82, 84], web server function calls [85], user input [31, 48, 83], and address layout [25]. Flush+Reload requires shared memory between the attacker and the victim. When attacking the last-level cache, Prime+Probe requires it to be shared and inclusive. While some Intel processors do not have inclusive last-level caches anymore [81], AMD always focused on non-inclusive or exclusive last-level caches [38]. Without inclusivity and shared memory, these attacks do not apply to AMD CPUs.

With the recent transient-execution attacks, adversaries can directly exfiltrate otherwise inaccessible data on the system [41, 49, 68, 74, 75]. However, AMD's microarchitectures seem to be vulnerable to only a few of them [9, 17]. Consequently, AMD CPUs do not require software mitigations with high performance penalties. Additionally, with the performance improvements of the latest microarchitectures, the share of AMD CPU's used is currently increasing in the cloud [10] and consumer desktops [34].

Since the Bulldozer microarchitecture [6], AMD uses an L1D cache way predictor in their processors. The predictor computes a μTag using an undocumented hash function on the virtual address. This μTag is used to look up the L1D cache way in a prediction table. Hence, the CPU has to compare the cache tag in only one way instead of all possible ways, reducing the power consumption.

In this paper, we present the first attacks on cache way predictors. For this purpose, we reverse-engineered the undocumented hash function of AMD's L1D cache way predictor in microarchitectures from 2001 up to 2019. We discovered two different hash functions that have been implemented in AMD's way predictors. Knowledge of these functions is the basis of our attack techniques. In the first attack technique, Collide+Probe, we exploit μTag collisions of

virtual addresses to monitor the memory accesses of a victim time-sharing the same logical core. Collide+Probe does not require shared memory between the victim and the attacker, unlike Flush+Reload, and no knowledge of physical addresses, unlike Prime+Probe. In the second attack technique, Load+Reload, we exploit the property that a physical memory location can only reside once in the L1D cache. Thus, accessing the same location with a different virtual address evicts the location from the L1D cache. This allows an attacker to monitor memory accesses on a victim, even if the victim runs on a sibling logical core. Load+Reload is on par with Flush+Reload in terms of accuracy and can achieve a higher temporal resolution as it does not invalidate a cache line in the entire cache hierarchy. This allows stealthier attacks that do not induce last-level-cache evictions.

We demonstrate the implications of Collide+Probe and Load+Reload in different attack scenarios. First, we implement a covert channel between two processes with a transmission rate of up to 588.9 kB/s outperforming state-of-the-art covert channels. Second, we use μTag collisions to reduce the entropy of different ASLR implementations. We break kernel ASLR on a fully updated Linux system and demonstrate entropy reduction on user-space applications, the hypervisor, and even on our own address space from sandboxed JavaScript. Furthermore, we successfully recover the secret key using Collide+Probe on an AES T-table implementation. Finally, we use Collide+Probe as a covert channel in a Spectre attack to exfiltrate secret data from the kernel. While we still use a cache-based covert channel, in contrast to previous attacks [41, 44, 51, 70], we do not rely on shared memory between the user application and the kernel. We propose different countermeasures in software and hardware, mitigating Collide+Probe and Load+Reload on current systems and in future designs.

Contributions. The main contributions are as follows:

(1) We reverse engineer the L1D cache way predictor of AMD CPUs and provide the addressing functions for virtually all microarchitectures.
(2) We uncover the L1D cache way predictor as a source of side-channel leakage and present two new cache-attack techniques, Collide+Probe and Load+Reload.
(3) We show that Collide+Probe is on par with Flush+Reload and Prime+Probe but works in scenarios where other cache attacks fail.
(4) We demonstrate and evaluate our attacks in sandboxed JavaScript and virtualized cloud environments.

Responsible Disclosure. We responsibly disclosed our findings to AMD on August 23rd, 2019.

Outline. Section 2 provides background information on CPU caches, cache attacks, way prediction, and simultaneous multithreading (SMT). Section 3 describes the reverse engineering of the way predictor that is necessary for our Collide+Probe and Load+Reload attack techniques outlined in Section 4. In Section 5, we evaluate the attack techniques in different scenarios. Section 6 discusses the interactions between the way predictor and other CPU features. We propose countermeasures in Section 7 and conclude our work in Section 8.

2 BACKGROUND

In this section, we provide background on CPU caches, cache attacks, high-resolution timing sources, simultaneous multithreading (SMT), and way prediction.

2.1 CPU Caches

CPU caches are a type of memory that is small and fast, that the CPU uses to store copies of data from main memory to hide the latency of memory accesses. Modern CPUs have multiple cache levels, typically three, varying in size and latency: the L1 cache is the smallest and fastest, while the L3 cache, also called the last-level cache, is bigger and slower.

Modern caches are set-associative, *i.e.*, a cache line is stored in a fixed set determined by either its virtual or physical address. The L1 cache typically has 8 ways per set, and the last-level cache has 12 to 20 ways, depending on the size of the cache. Each line can be stored in any of the ways of a cache set, as determined by the replacement policy. While the replacement policy for the L1 and L2 data cache on Intel is most of the time pseudo least-recently-used (LRU) [1], the replacement policy for the last-level cache (LLC) can differ [79]. Intel CPUs until Sandy Bridge use pseudo least-recently-used (LRU), for newer microarchitectures it is undocumented [79].

The last-level cache is physically indexed and shared across cores of the same CPU. In most Intel implementations, it is also inclusive of L1 and L2, which means that all data in L1 and L2 is also stored in the last-level cache. On AMD Zen processors, the L1D cache is virtually indexed and physically tagged (VIPT). The last-level cache is a non-inclusive victim cache. To maintain this property, every line evicted from the last-level cache is also evicted from L1 and L2. The last-level cache, though shared across cores, is also divided into slices. The undocumented hash function that maps physical addresses to slices in Intel CPUs has been reverse-engineered [52].

2.2 Cache Attacks

Cache attacks are based on the timing difference between accessing cached and non-cached memory. They can be leveraged to build side-channel attacks and covert channels. Among cache attacks, access-driven attacks are the most powerful ones, where an attacker monitors its own activity to infer the activity of its victim. More specifically, an attacker detects which cache lines or cache sets the victim has accessed.

Access-driven attacks can further be categorized into two types, depending on whether or not the attacker shares memory with its victim, e.g., using a shared library or memory deduplication. Flush+Reload [82], Evict+Reload [31] and Flush+Flush [30] all rely on shared memory that is also shared in the cache to infer whether the victim accessed a particular cache line. The attacker evicts the shared data either by using the `clflush` instruction (Flush+Reload and Flush+Flush), or by accessing congruent addresses, *i.e.*, cache lines that belong to the same cache set (Evict+Reload). These attacks have a very fine granularity (*i.e.*, a 64-byte memory region), but they are not applicable if shared memory is not available in the corresponding environment. Especially in the cloud, shared memory is usually not available across VMs as memory deduplication is disabled for security concerns [76]. Irazoqui et al. [38] showed that an attack similar to Flush+Reload is also possible in a cross-CPU

attack. It exploits that cache invalidations (e.g., from `clflush`) are propagated to all physical processors installed in the same system. When reloading the data, as in Flush+Reload, they can distinguish the timing difference between a cache hit in a remote processor and a cache miss, which goes to DRAM.

The second type of access-driven attacks, called Prime+Probe [37, 50, 59], does not rely on shared memory and is, thus, applicable to more restrictive environments. As the attacker has no shared cache line with the victim, the `clflush` instruction cannot be used. Thus, the attacker has to access congruent addresses instead (cf. Evict+Reload). The granularity of the attack is coarser, *i.e.*, an attacker only obtains information about the accessed cache set. Hence, this attack is more susceptible to noise. In addition to the noise caused by other processes, the replacement policy makes it hard to guarantee that data is actually evicted from a cache set [29].

With the general development to switch from inclusive caches to non-inclusive caches, Intel introduced cache directories. Yan et al. [81] showed that the cache directory is still inclusive, and an attacker can evict a cache directory entry of the victim to invalidate the corresponding cache line. This allows mounting Prime+Probe and Evict+Reload attacks on the cache directory. They also analyzed whether the same attack works on AMD Piledriver and Zen processors and discovered that it does not, because these processors either do not use a directory or use a directory with high associativity, preventing cross-core eviction either way. Thus, it remains to be answered what types of eviction-based attacks are feasible on AMD processors and on which microarchitectural structures.

2.3 High-resolution Timing

For most cache attacks, the attacker requires a method to measure timing differences in the range of a few CPU cycles. The `rdtsc` instruction provides unprivileged access to a model-specific register returning the current cycle count and is commonly used for cache attacks on Intel CPUs. Using this instruction, an attacker can get timestamps with a resolution between 1 and 3 cycles on modern CPUs. On AMD CPUs, this register has a cycle-accurate resolution until the Zen microarchitecture. Since then, it has a significantly lower resolution as it is only updated every 20 to 35 cycles (cf. Appendix A). Thus, `rdtsc` is only sufficient if the attacker can repeat the measurement and use the average timing differences over all executions. If an attacker tries to monitor one-time events, the `rdtsc` instruction on AMD cannot directly be used to observe timing differences, which are only a few CPU cycles.

The AMD Ryzen microarchitecture provides the *Actual Performance Frequency Clock Counter* (`APERF` counter) [7] which can be used to improve the accuracy of the timestamp counter. However, it can only be accessed in kernel mode. Although other timing primitives provided by the kernel, such as `get_monotonic_time`, provide nanosecond resolution, they can be more noisy and still not sufficiently accurate to observe timing differences, which are only a few CPU cycles.

Hence, on more recent AMD CPUs, it is necessary to resort to a different method for timing measurements. Lipp et al. [48] showed that *counting threads* can be used on ARM-based devices where unprivileged high-resolution timers are unavailable. Schwarz et al. [66] showed that a counting thread can have a higher resolution than the `rdtsc` instruction on Intel CPUs. A counting thread constantly increments a global variable used as a timestamp without relying on microarchitectural specifics and, thus, can also be used on AMD CPUs.

2.4 Simultaneous Multithreading (SMT)

Simultaneous Multithreading (SMT) allows optimizing the efficiency of superscalar CPUs. SMT enables multiple independent threads to run in parallel on the same physical core sharing the same resources, e.g., execution units and buffers. This allows utilizing the available resources better, increasing the efficiency and throughput of the processor. While on an architectural level, the threads are isolated from each other and cannot access data of other threads, on a microarchitectural level, the same physical resources may be used. Intel introduced SMT as *Hyperthreading* in 2002. AMD introduced 2-way SMT with the Zen microarchitecture in 2017.

Recently, microarchitectural attacks also targeted different shared resources: the TLB [24], store buffer [16], execution ports [2, 13], fill-buffers [68, 75], and load ports [68, 75].

2.5 Way Prediction

To look up a cache line in a set-associative cache, bits in the address determine in which set the cache line is located. With an n-way cache, n possible entries need to be checked for a tag match. To avoid wasting power for n comparisons leading to a single match, Inoue et al. [36] presented way prediction for set-associative caches. Instead of checking all ways of the cache, a way is predicted, and only this entry is checked for a tag match. As only one way is activated, the power consumption is reduced. If the prediction is correct, the access has been completed, and access times similar to a direct-mapped cache are achieved. If the prediction is incorrect, a normal associative check has to be performed.

We only describe AMD's way predictor [8, 23] in more detail in the following section. However, other CPU manufacturers hold patents for cache way prediction as well [56, 64]. CPU's like the Alpha 21264 [40] also implement way prediction to combine the advantages of set-associative caches and the fast access time of a direct-mapped cache.

3 REVERSE-ENGINEERING AMDS WAY PREDICTOR

In this section, we explain how to reverse-engineer the L1D way predictor used in AMD CPUs since the Bulldozer microarchitecture. First, we explain how the AMD L1D way predictor predicts the L1D cache way based on hashed virtual addresses. Second, we reverse-engineer the undocumented hash function used for the way prediction in different microarchitectures. With the knowledge of the hash function and how the L1D way predictor works, we can then build powerful side-channel attacks exploiting AMD's way predictor.

3.1 Way Predictor

Since the AMD Bulldozer microarchitecture, AMD uses a way predictor in the L1 data cache [6]. By predicting the cache way, the CPU only has to compare the cache tag in one way instead of all ways.

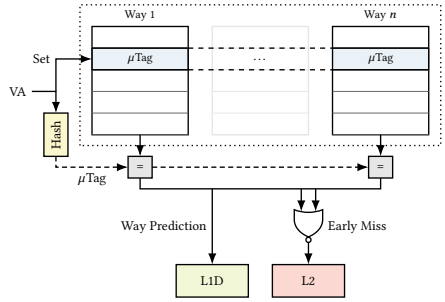


Figure 1: Simplified illustration of AMD's way predictor.

While this reduces the power consumption of an L1D lookup [8], it may increase the latency in the case of a misprediction.

Every cache line in the L1D cache is tagged with a linear-address-based μTag [8, 23]. This μTag is computed using an undocumented hash function, which takes the virtual address as the input. For every memory load, the way predictor predicts the cache way of every memory load based on this μTag. As the virtual address, and thus the μTag, is known before the physical address, the CPU does not have to wait for the TLB lookup. Figure 1 illustrates AMD's way predictor. If there is no match for the calculated μTag, an early miss is detected, and a request to L2 issued.

Aliased cache lines can induce performance penalties, *i.e.*, two different *virtual* addresses map to the same *physical* location. As VIPT caches with a size lower or equal the number of ways multiplied by the page size behave functionally like PIPT caches. Hence, there are no duplicates for aliased addresses and, thus, in such a case where data is loaded from an aliased address, the load sees an L1D cache miss and thus loads the data from the L2 data cache [8]. If there are multiple memory loads from aliased virtual addresses, they all suffer an L1D cache miss. The reason is that every load updates the μTag and thus ensures that any other aliased address sees an L1D cache miss [8]. In addition, if two different virtual addresses yield the same μTag, accessing one after the other yields a conflict in the μTag table. Thus, an L1D cache miss is suffered, and the data is loaded from the L2 data cache.

3.2 Hash Function

The L1D way predictor computes a hash (μTag) from the virtual address, which is used for the lookup to the way-predictor table. We assume that this undocumented hash function is linear based on the knowledge of other such hash functions, e.g., the cache-slice function of Intel CPUs [52], the DRAM-mapping function of Intel, ARM, and AMD CPUs [5, 60, 71], or the hash function for indirect branch prediction on Intel CPUs [41]. Moreover, we expect the size of the μTag to be a power of 2, resulting in a linear function.

We rely on μTag collisions to reverse-engineer the hash function. We pick two random virtual addresses that map to the same cache set. If the two addresses have the same μTag, repeatedly accessing them one after the other results in conflicts. As the data is then loaded from the L2 cache, we can either measure an increased access time or observe an increased number in the performance counter for L1 misses, as illustrated in Figure 2.

Creating Sets. With the ability to detect conflicts, we can build N sets representing the number of entries in the μTag table. First, we create a pool v of virtual addresses, which all map to the same cache set, *i.e.*, where bits 6 to 11 of the virtual address are the same. We start with one set S_0 containing one random virtual address out of the pool v. For each other randomly-picked address v_x, we measure the access time while alternatively accessing v_x and an address from each set $S_{0\ldots n}$. If we encounter a high access time, we measure conflicts and add v_x to that set. If v_x does not conflict with any existing set, we create a new set S_{n+1} containing v_x.

In our experiments, we recovered 256 sets. Due to measurement errors caused by system noise, there are sets with single entries that can be discarded. Furthermore, to retrieve all sets, we need to make sure to test against virtual addresses where a wide range of bits is set covering the yet unknown bits used by the hash function.

Recovering the Hash Function. Every virtual address, which is in the same set, produces the same hash. To recover the hash function, we need to find which bits in the virtual address are used for the 8 output bits that map to the 256 sets. Due to its linearity, each output bit of the hash function can be expressed as a series of XORs of bits in the virtual address. Hence, we can express the virtual addresses as an over-determined linear equation system in finite field 2, *i.e.*, GF(2). The solutions of the equation system are then linear functions that produce the μTag from the virtual address.

To build the equation system, we use each of the virtual addresses in the 256 sets. For every virtual address, the b bits of the virtual address a are the coefficients, and the bits of the hash function x are the unknown. The right-hand side of the equation y is the same for all addresses in the set. Hence, for every address a in set s, we get an equation of the form $a_{b-1}x_{b-1} \oplus a_{b-2}x_{b-2} \oplus \cdots \oplus a_{12}x_{12} = y_s$.

While the least-significant bits 0-5 define the cache line offset, note that bits 6-11 determine the cache set and are not used for the μTag computation [8]. To solve the equation system, we used the Z3 SMT solver. Every solution vector represents a function which XORs the virtual-address bits that correspond to '1'-bits in the solution vector. The hash function is the set of linearly independent functions, *i.e.*, every linearly independent function yields one bit of the hash function. The order of the bits cannot be recovered. However, this is not relevant, as we are only interested whether addresses collide, not in their numeric μTag value.

We successfully recovered the undocumented μTag hash function on the AMD Zen, Zen+ and Zen 2 microarchitecture. The function illustrated in Figure 3a uses bits 12 to 27 to produce an 8-bit value mapping to one of the 256 sets:

$$h(v) = (v_{12} \oplus v_{27}) \parallel (v_{13} \oplus v_{26}) \parallel (v_{14} \oplus v_{25}) \parallel (v_{15} \oplus v_{20}) \parallel (v_{16} \oplus v_{21}) \parallel (v_{17} \oplus v_{22}) \parallel (v_{18} \oplus v_{23}) \parallel (v_{19} \oplus v_{24})$$

We recovered the same function for various models of the AMD Zen microarchitectures that are listed in Table 1. For the Bulldozer microarchitecture (FX-4100), the Piledriver microarchitecture (FX-8350), and the Steamroller microarchitecture (A10-7870K), the hash function uses the same bits but in a different combination Figure 3b.

3.3 Simultaneous Multithreading

As AMD introduced simultaneous multithreading starting with the Zen microarchitecture, the filed patent [23] does not cover any

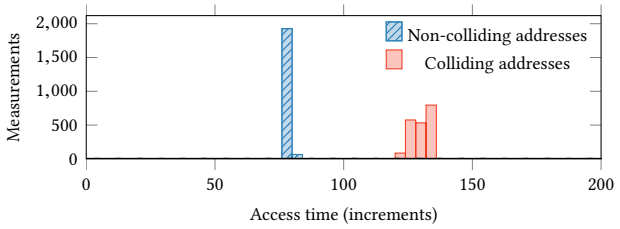
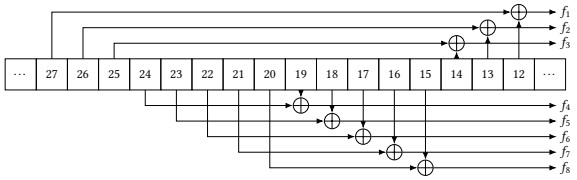
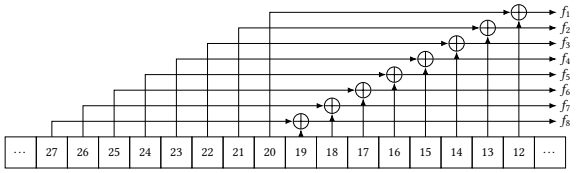


Figure 2: Measured duration of 250 alternating accesses to addresses with and without the same μTag.



(a) Zen, Zen+, Zen 2



(b) Bulldozer, Piledriver, Steamroller

Figure 3: The recovered hash functions use bits 12 to 27 of the virtual address to compute the μTag.

insights on how the way predictor might handle multiple threads. While the way predictor has been used since the Bulldozer microarchitecture [6], parts of the way predictor have only been documented with the release of the Zen microarchitecture [8]. However, the influence of simultaneous multithreading is not mentioned.

Typically, two sibling threads can either share a hardware structure *competitively* with the option to tag entries or by *statically partitioning* them. For instance, on the Zen microarchitecture, execution units, schedulers, or the cache are competitively shared, and the store queue and retire queue are statically partitioned [18]. Although the load queue, as well as the instruction and data TLB, are competitively shared between the threads, the data in these structures can only be accessed by the thread owning it.

Under the assumption that the data structures of the way predictor are competitively shared between threads, one thread could directly influence the sibling thread, enabling cross-thread attacks. We validate this assumption by accessing two addresses with the same μTag on both threads. However, we do not observe collisions, neither by measuring the access time nor in the number of L1 misses. While we reverse-engineered the same mapping function (see Section 3.2) for both threads, the possibility remains that additional per-thread information is used for selecting the data-structure entry, allowing one thread to evict entries of the other.

Hence, we extend the experiment in accessing addresses mapping to all possible μTags on one hardware thread (and all possible cache sets). While we repeatedly accessed one of these addresses on one hardware thread, we measure the number of L1 misses to a single virtual address on the sibling thread. However, we are not able to observe any collisions and, thus, conclude that either individual structures are used per thread or that they are shared but tagged for each thread. The only exceptions are aliased loads as the hardware updates the μTag in the aliased way (see Section 3.1).

In another experiment, we measure access times of two virtual addresses that are mapped to the same physical address. As documented [8], loads to an aliased address see an L1D cache miss and, thus, load the data from the L2 data cache. While we verified this behavior, we additionally observed that this is also the case if the other thread performs the other load. Hence, the structure used is searched by the sibling thread, suggesting a competitively shared structure that is tagged with the hardware threads.

4 USING THE WAY PREDICTOR FOR SIDE CHANNELS

In this section, we present two novel side channels that leverage AMD's L1D cache way predictor. With Collide+Probe, we monitor memory accesses of a victim's process without requiring the knowledge of physical addresses. With Load+Reload, while relying on shared memory similar to Flush+Reload, we can monitor memory accesses of a victim's process running on the sibling hardware thread without invalidating the targeted cache line from the entire cache hierarchy.

4.1 Collide+Probe

Collide+Probe is a new cache side channel exploiting μTag collisions in AMD's L1D cache way predictor. As described in Section 3, the way predictor uses virtual-address-based μTags to predict the L1D cache way. If an address is accessed, the μTag is computed, and the way-predictor entry for this μTag is updated. If a subsequent access to a different address with the same μTag is performed, a μTag collision occurs, and the data has to be loaded from the L2D cache, increasing the access time. With Collide+Probe, we exploit this timing difference to monitor accesses to such colliding addresses.

Threat Model. For this attack, we assume that the attacker has unprivileged native code execution on the target machine and runs on the same logical CPU core as the victim. Furthermore, the attacker can force the execution of the victim's code, e.g., via a function call in a library or a system call.

Setup. The attacker first chooses a virtual address v of the victim that should be monitored for accesses. This can be an arbitrary valid address in the victim's address space. There are no constraints in choosing the address. The attacker can then compute the μTag μ_v of the target address using the hash function from Section 3.2. We assume that ASLR is either not active or has already been broken (cf. Section 5.2). However, although with ASLR, the actual virtual address used in the victim's process are typically unknown to the attacker, it is still possible to mount an attack. Instead of choosing a virtual address, the attacker initially performs a cache template attack [31] to detect which of 256 possible μTags should

be monitored. Similar to Prime+Probe [58], where the attacker monitors the activity of cache sets, the attacker monitors μTag collisions while triggering the victim.

Attack. To mount a Collide+Probe attack, the attacker selects a virtual address v' in its own address space that yields the same μTag $\mu_{v'}$ as the target address v, *i.e.*, $\mu_v = \mu_{v'}$. As there are only 256 different μTags, this can easily be done by randomly choosing addresses until the chosen address has the same μTag. Moreover, both v and v' have to be in the same cache set. However, this is easily satisfiable, as the cache set is determined by bits 6-11 of the virtual address. The attack consists of 3 phases performed repeatedly:

Phase 1: Collide. In the first phase, the attacker accesses the pre-computed address v' and, thus, updates the way predictor. The way predictor associates the cache line of v' with its μTag $\mu_{v'}$ and subsequent memory accesses with the same μTag are predicted to be in the same cache way. Since the victim's address v has the same μTag ($\mu_v = \mu_{v'}$), the μTag of that cache line is marked invalid and the data is effectively inaccessible from the L1D cache.

Phase 2: Scheduling the victim. In the second phase, the victim is scheduled to perform its operations. If the victim does not access the monitored address v, the way predictor remains in the same state as set up by the attacker. Thus, the attacker's data is still accessible from the L1D cache. However, if the victim performs an access to the monitored address v, the way predictor is updated again causing the attacker's data to be inaccessible from L1D.

Phase 3: Probe. In the third and last phase of the attack, the attacker measures the access time to the pre-computed address v'. If the victim has not accessed the monitored address v, the data of the pre-computed address v' is still accessible from the L1D cache and the way prediction is correct. Thus, the measured access time is fast. If the victim has accessed the monitored address v and thus changed the state of the way predictor, the attacker suffers an L1D cache miss when accessing v', as the prediction is now incorrect. The data of the pre-computed address v' is loaded from the L2 cache and, thus, the measured access time is slow. By distinguishing between these cases, the attacker can deduce whether the victim has accessed the targeted data.

Listing 1 shows an implementation of the Collide+Probe attack where the colliding address `colliding_address` is computed beforehand. The code closely follows the three attack phases. First, the colliding address is accessed. Then, the victim is scheduled, illustrated by the `run_victim` function. Afterwards, the access time to the same address is measured where the `get_time` function is implemented using a timing source discussed in Section 2.3. The measured access time allows the attacker to distinguish between an L1D cache hit and an L2-cache hit and, thus, deduce if the victim has accessed the targeted address. As other accesses with the same cache set influence the measurements, the attacker can repeat the experiment to average out the measured noise.

Comparison to Other Cache Attacks. Finally, we want to discuss the advantages and disadvantages of the Collide+Probe attack in comparison to other cache side-channel attacks. In contrast to Prime+Probe, no knowledge of physical addresses is required as the way predictor uses the virtual address to compute μTags. Thus, with native code execution, an attacker can find addresses corresponding to a specific μTag without any effort. Another advantage

```
1 access(colliding_address);
2 run_victim();
3 size_t begin = get_time();
4 access(colliding_address);
5 size_t end = get_time() - begin;
6 if ((end - begin) > THRESHOLD) report_event();
```

Listing 1: Implementation of the Collide+Probe attack

of Collide+Probe over Prime+Probe is that a single memory load is enough to guarantee that a subsequent load with the same μTag is served from the L2 cache. With Prime+Probe, multiple loads are required to ensure that the target address is evicted from the cache. In modern Prime+Probe attacks, the last-level cache is targeted [37, 48, 50, 63, 67], and knowledge of physical addresses is required to compute both the cache set and cache slice [52]. While Collide+Probe requires knowledge of virtual addresses, they are typically easier to get than physical addresses. In contrast to Flush+Reload, Collide+Probe does neither require any specific instructions like `clflush` nor shared memory between the victim and the attacker. A disadvantage is that distinguishing L1D from L2 hits in Collide+Probe requires a timing primitive with higher precision than required to distinguish cache hits from misses in Flush+Reload.

4.2 Load+Reload

Load+Reload exploits the way predictor's behavior for aliased address, *i.e.*, virtual addresses mapping to the same physical address. When accessing data through a virtual-address alias, the data is always requested from the L2 cache instead of the L1D cache [8]. By monitoring the performance counter for L1 misses, we also observe this behavior across hardware threads. Consequently, this allows one thread to evict shared data used by the sibling thread with a single load. Although the requested data is stored in the L1D cache, it remains inaccessible for the other thread and, thus, introduces a timing difference when it is accessed.

Threat Model. For this attack, we assume that the attacker has unprivileged native code execution on the target machine. The attacker and victim run simultaneously on the same physical but different logical CPU thread. The attack target is a memory location with virtual address v shared between the attacker and victim, e.g., a shared library.

Attack. Load+Reload exploits the timing difference when accessing a virtual-address alias v' to build a cross-thread attack on shared memory. The attack consists of 3 phases:

Phase 1: Load. In contrast to Flush+Reload, where the targeted address v is flushed from the cache hierarchy, Load+Reload loads an address v' with the same physical tag as v in the first phase. Thereby, it renders the cache line containing v inaccessible from the L1D cache for the sibling thread.

Phase 2: Scheduling the victim. In the second phase, the victim process is scheduled. If the victim process accesses the targeted cache line with address v, it sees an L1D cache miss. As a result, it loads the data from the L2 cache, invalidating the attacker's cache line with address v' in the process.

Phase 3: Reload. In the third phase, the attacker measures the access time to the address v'. If the victim process has accessed the

cache line with address v, the attacker observes an L1D cache miss and loads the data from the L2 cache, resulting in a higher access time. Otherwise, if the victim has not accessed the cache line with address v, it is still accessible in the L1D cache for the attacker and, thus, a lower access time is measured. By distinguishing between both cases, the attacker can deduce whether the victim has accessed the address v.

Comparison with Flush+Reload. While Flush+Reload invalidates a cache line from the entire cache hierarchy, Load+Reload only *evicts* the data for the sibling thread from the L1D. Thus, Load+Reload is limited to cross-thread scenarios, while Flush+Reload is applicable to cross-core scenarios too.

5 CASE STUDIES

To demonstrate the impact of the side channel introduced by the μTag, we implement different attack scenarios. In Section 5.1, we implement a covert channel between two processes with a transmission rate of up to 588.9 kB/s outperforming state-of-the-art covert channels. In Section 5.2, we break kernel ASLR, demonstrate how user-space ASLR can be weakened, and reduce the ASLR entropy of the hypervisor in a virtual-machine setting. In Section 5.3, we use Collide+Probe as a covert channel to extract secret data from the kernel in a Spectre attack. In Section 5.4, we recover secret keys in AES T-table implementations.

Timing Measurement. As explained in Section 2.3, we cannot rely on the `rdtsc` instruction for high-resolution timestamps on AMD CPUs since the Zen microarchitecture. As we use recent AMD CPUs for our evaluation, we use a counting thread (cf. Section 2.3) running on the sibling logical CPU core for most of our experiments if applicable. In other cases, e.g., a covert channel scenario, the counting thread runs on a different physical CPU core.

Environment. We evaluate our attacks on different environments listed in Table 1, with CPUs from K8 (released 2013) to Zen 2 (released in 2019). We have reverse-engineered 2 unique hash functions, as described in Section 3. One is the same for all Zen microarchitectures, and the other is the same for all previous microarchitectures with a way predictor.

5.1 Covert Channel

A covert channel is a communication channel between two parties that are not allowed to communicate with each other. Such a covert channel can be established by leveraging a side channel. The μTag used by AMD's L1D way prediction enables a covert channel for two processes accessing addresses with the same μTag.

For the most simplistic form of the covert channel, two processes agree on a μTag and a cache set (*i.e.*, the least-significant 12 bits of the virtual addresses are the same). This μTag is used for sending and receiving data by inducing and measuring cache misses.

In the initialization phase, both parties allocate their own page. The sender chooses a virtual address v_S, and the receiver chooses a virtual address v_R that fulfills the aforementioned requirements, *i.e.*, v_S and v_R are in the same cache set and yield the same μTag. The μTag can simply be computed using the reverse-engineered hash function of Section 3.

Table 1: Tested CPUs with their microarchitecture (μ-arch.) and whether they have a way predictor (WP).

Setup	CPU	μ-arch.	WP
Lab	AMD Athlon 64 X2 3800+	K8	✗
Lab	AMD Turion II Neo N40L	K10	✗
Lab	AMD Phenom II X6 1055T	K10	✗
Lab	AMD E-450	Bobcat	✗
Lab	AMD Athlon 5350	Jaguar	✗
Lab	AMD FX-4100	Bulldozer	✓
Lab	AMD FX-8350	Piledriver	✓
Lab	AMD A10-7870K	Steamroller	✓
Lab	AMD Ryzen Threadripper 1920X	Zen	✓
Lab	AMD Ryzen Threadripper 1950X	Zen	✓
Lab	AMD Ryzen Threadripper 1700X	Zen	✓
Lab	AMD Ryzen Threadripper 2970WX	Zen+	✓
Lab	AMD Ryzen 7 3700X	Zen 2	✓
Cloud	AMD EPYC 7401p	Zen	✓
Cloud	AMD EPYC 7571	Zen	✓

To encode a 1-bit to transmit, the sender accesses address v_S. To transmit a 0-bit, the sender does not access address v_S. The receiving end decodes the transmitted information by measuring the access time when loading address v_R. If the sender has accessed address v_S to transmit a 1, the collision caused by the same μTag of v_S and v_R results in a slow access time for the receiver. If the sender has not accessed address v_S, no collision caused the address v_R to be evicted from L1D and, thus, the access time is fast. This timing difference allows the receiver to decode the transmitted bit.

Different cache-based covert channels use the same side channel to transmit multiple bits at once. For instance, different cache lines [30, 48] or different cache sets [48, 53] are used to encode one bit of information on its own. We extended the described μTag covert channel to transmit multiple bits in parallel by utilizing multiple cache sets. Instead of decoding the transmitted bit based on the timing difference of one address, we use two addresses in two cache sets for every bit we transmit: One to represent a 1-bit and the other to represent the 0-bit. As the L1D has 64 cache sets, we can transmit up to 32 bit in parallel without reusing cache sets.

Performance Evaluation. We evaluated the transmission and error rate of our covert channel in a local setting and a cloud setting by sending and receiving a randomly generated data blob. We achieved a maximum transmission rate of 588.9 kB/s ($\sigma_{\bar{x}} = 0.544$, $n = 1000$) using 80 channels in parallel on the AMD Ryzen Threadripper 1920X. On the AMD EPYC 7571 in the Amazon EC2 cloud, we achieved a maximum transmission rate of 544.0 kB/s ($\sigma_{\bar{x}} = 0.548$, $n = 1000$) also using 80 channels. In contrast, L1 Prime+Probe achieved a transmission rate of 400 kB/s [59] and Flush+Flush a transmission rate of 496 kB/s [30]. As illustrated in Figure 4, the mean transmission rate increases with the number of bits sent in parallel. However, the error rate increases drastically when transmitting more than 64 bits in parallel, as illustrated in Figure 6. As the number of available different cache sets for our channel is exhausted for our covert channel, sending more bits in parallel

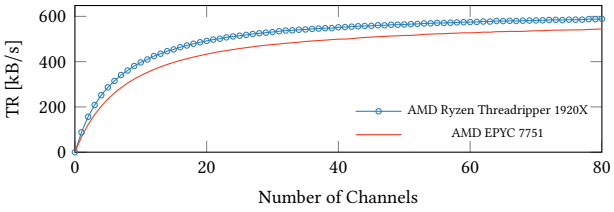


Figure 4: Mean transmission rate of the covert channels using multiple parallel channels on different CPUs.

Table 2: Evaluation of the ASLR experiments

Target	Entropy	Bits Reduced	Success Rate	Timing Source	Time
Linux Kernel	9	7	98.5%	thread	0.51 ms (σ = 12.12 µs)
User Process	13	13	88.9%	thread	1.94 s (σ = 1.76 s)
Virt. Manager	28	16	90.0%	`rdtsc`	2.88 s (σ = 3.16 s)
Virt. Module	18	8	98.9%	`rdtsc`	0.14 s (σ = 1.74 ms)
Mozilla Firefox	28	15	98.0%	web worker	2.33 s (σ = 0.03 s)
Google Chrome	28	15	86.1%	web worker	2.90 s (σ = 0.25 s)
Chrome V8	28	15	100.0%	`rdtsc`	1.14 s (σ = 0.03 s)

would reuse already used sets. This increases the chance of wrong measurements and, thus, the error rate.

Error Correction. As accesses to unrelated addresses with the same μTag as our covert channel introduce noise in our measurements, an attacker can use error correction to achieve better transmission. Using hamming codes [33], we introduce n additional parity bits allowing us to detect and correct wrongly measured bits of a packet with a size of $2^n - 1$ bits. For our covert channel, we implemented different Hamming codes $H(m, n)$ that encode n bits by adding $m - n$ parity bits. The receiving end of the covert channel computes the parity bits from the received data and compares it with the received parity bits. Naturally, they only differ if a transmission error occurred. The erroneous bit position can be computed, and the bit error corrected by flipping the bit. This allows to detect up to 2-bit errors and correct one-bit errors for a single transmission.

We evaluated different hamming codes on an AMD Ryzen Threadripper 1920X, as illustrated in Figure 7 in Appendix B. When sending data through 60 parallel channels, the $H(7, 4)$ code reduces the error rate to 0.14 % ($\sigma_{\bar{x}} = 0.08$, $n = 1000$), whereas the $H(15, 11)$ code achieves an error rate of 0.16 % ($\sigma_{\bar{x}} = 0.08$, $n = 1000$). While the $H(7, 4)$ code is slightly more robust [33], the $H(15, 11)$ code achieves a better transmission rate of 452.2 kB/s ($\sigma_{\bar{x}} = 7.79$, $n = 1000$).

More robust protocols have been used in cache-based covert channels in the past [48, 53] to achieve error-free communication. While these techniques can be applied to our covert channel as well, we leave it up to future work.

Limitations. As we are not able to observe μTag collisions between two processes running on sibling threads on one physical core, our covert channel is limited to processes running on the same logical core.

5.2 Breaking ASLR and KASLR

To exploit a memory corruption vulnerability, an attacker often requires knowledge of the location of specific data in memory. With *address space layout randomization* (ASLR), a basic memory protection mechanism has been developed that randomizes the locations of memory sections to impede the exploitation of these bugs. ASLR is not only applied to user-space applications but also implemented in the kernel (KASLR), randomizing the offsets of code, data, and modules on every boot.

In this section, we exploit the relation between virtual addresses and μTags to reduce the entropy of ASLR in different scenarios. With Collide+Probe, we can determine the μTags accessed by the victim, e.g., the kernel or the browser, and use the reverse-engineered mapping functions (Section 3.2) to infer bits of the addresses. We show an additional attack on heap ASLR in Appendix C.

5.2.1 Kernel. On modern Linux systems, the position of the kernel text segment is randomized inside the 1 GB area from `0xffff ffff 8000 0000` - `0xffff ffff c000 0000` [39, 46]. As the kernel image is mapped using 2 MB pages, it can only be mapped in 512 different locations, resulting in 9 bit of entropy [65].

Global variables are stored in the `.bss` and `.data` sections of the kernel image. Since 2 MB physical pages are used, the 21 lower address bits of a global variable are identical to the lower 21 bits of the offset within the the kernel image section. Typically, the kernel image is public and does not differ among users with the same operating system. With the knowledge of the μTag from the address of a global variable, one can compute the address bits 21 to 27 using the hash function of AMD's L1D cache way predictor.

To defeat KASLR using Collide+Probe, the attacker needs to know the offset of a global variable within the kernel image that is accessed by the kernel on a user-triggerable event, e.g., a system call or an interrupt. While not many system calls access global variables, we found that the `SYS_time` system call returns the value of the global second counter `obj.xtime_sec`. Using Collide+Probe, the attacker accesses an address v' with a specific μTag $\mu_{v'}$ and schedules the system call, which accesses the global variable with address v and μTag μ_v. Upon returning from the kernel, the attacker probes the μTag $\mu_{v'}$ using address v'. On a conflict, the attacker infers that the address v' has the same μTag, *i.e.*, $t = \mu_{v'} = \mu_v$. Otherwise, the attacker chooses another address v' with a different μTag $\mu_{v'}$ and repeats the process. As the μTag bits t_0 to t_7 are known, the address bits v_{20} to v_{27} can be computed from address bits v_{12} to v_{19} based on the way predictor's hash functions (Section 3.2). Following this approach, we can compute address bits 21 to 27 of the global variable. As we know the offset of the global variable inside the kernel image, we can also recover the start address of the kernel image mapping, leaving only bits 28 and 29 unknown. As the kernel is only randomized once per boot, the reduction to only 4 address possibilities gives an attacker a significant advantage.

For the evaluation, we tested 10 different randomization offsets on a Linux 4.15.0-58 kernel with an AMD Ryzen Threadripper 1920X processor. We ran our experiment 1000 times for each randomization offset. With a success rate of 98.5 %, we were able to reduce the entropy of KASLR on average in 0.51 ms (σ = 12.12 µs, n = 10 000).

While there are several microarchitectural KASLR breaks, this is to the best of our knowledge the first which reportedly works on AMD and not only on Intel CPUs. Hund et al. [35] measured

differences in the runtime of page faults when repeatedly accessing either valid or invalid kernel addresses on Intel CPUs. Barresi et al. [11] exploited page deduplication to break ASLR: a copy-on-write pagefault only occurs for the page with the correctly guessed address. Gruss et al. [28] exploited runtime differences in the `prefetch` instruction on Intel CPUs to detect mapped kernel pages. Jang et al. [39] showed that the difference in access time to valid and invalid kernel addresses can be measured when suppressing exceptions with Intel TSX. Evtyushkin et al. [22] exploited the branch-target buffer on Intel CPUs to gain information on mapped pages. Schwarz et al. [65] showed that the store-to-load forwarding logic on Intel CPUs is missing a permission check which allows to detect whether any virtual address is valid. Canella et al. [16] exploited that recent stores can be leaked from the store buffer on vulnerable Intel CPUs, allowing to detect valid kernel addresses.

5.2.2 Hypervisor. The *Kernel-based Virtual Machine* (KVM) is a virtualization module that allows the Linux kernel to act as a hypervisor to run multiple, isolated environments in parallel called virtual machines or guests. Virtual machines can communicate with the hypervisor using hypercalls with the privileged `vmcall` instruction. In the past, collisions in the branch target buffer (BTB) have been used to break hypervisor ASLR [22, 78].

In this scenario, we leak the base address of the KVM kernel module from a guest virtual machine. We issue hypercalls with invalid call numbers and monitor, which μTags have been accessed using Collide+Probe. In our evaluation, we identified two cache sets enabling us to weaken ASLR of the `kvm` and the `kvm_amd` module with a success rate of 98.8 % and an average runtime of 0.14 s (σ = 1.74 ms, n = 1000). We verified our results by comparing the leaked address bits with the symbol table (`/proc/kallsyms`).

Another target is the user-space virtualization manager, e.g., QEMU. Guest operating systems can interact with virtualization managers through various methods, e.g., the `out` instruction. Likewise to the previously described hypercall method, a guest virtual machine can use this method to trigger the managing user process to interact with the guest memory from its own address space. By using Collide+Probe in this scenario, we were able to reduce the ASLR entropy by 16 bits with a success rate of 90.0 % with an average run time of 2.88 s (σ = 3.16 s, n = 1000).

5.2.3 JavaScript. In this section, we show that Collide+Probe is not only restricted to native environments. We use Collide+Probe to break ASLR from JavaScript within Chrome and Firefox. As the JavaScript standard does not define a way to retrieve any address information, side channels in browsers have been used in the past [57], also to break ASLR, simplifying browser exploitation [25, 65].

The idea of our ASLR break is similar to the approach of reverse-engineering the way predictor's mapping function, as described in Section 3.2. First, we allocate a large chunk of memory as a JavaScript typed array. If the requested array length is big enough, the execution engine allocates it using `mmap`, placing the array at the beginning of a memory page [29, 69]. This allows using the indices within the array as virtual addresses with an additional constant offset. By accessing pairs of addresses, we can find μTag collisions allowing us to build an equation system where the only unknown bits are the bits of the address where the start of the array is located. As the equation system is very small, an attacker can trivially solve it in JavaScript.

However, to distinguish between colliding and non-colliding addresses, we require a high-precision timer in JavaScript. While the `performance.now()` function only returns rounded results for security reasons [3, 14], we leverage an alternative timing source [25, 69]. For our evaluation, we used the technique of a counting thread constantly incrementing a shared variable [25, 48, 69, 80].

We tested our proof-of-concept in both the Chrome 76.0.3809 and Firefox 68.0.2 web browsers as well as the Chrome V8 standalone engine. In Firefox, we are able to reduce the entropy by 15 bits with a success rate of 98 % and an average run time of 2.33 s (σ = 0.03 s, n = 1000). With Chrome, we can correctly reduce the bits with a success rate of 86.1 % and an average run time of 2.90 s (σ = 0.25 s, n = 1000). As the JavaScript standard does not provide any functionality to retrieve the addresses used by variables, we extended the capabilities of the Chrome V8 engine to verify our results. We introduced several custom JavaScript functions, including one that returned the virtual address of an array. This provided us with the ground truth to verify that our proof-of-concept recovered the address bits correctly. Inside the extended Chrome V8 engine, we were able to recover the address bits with a success rate of 100 % and an average run time of 1.14 s (σ = 0.03 s, n = 1000).

5.3 Leaking Kernel Memory

In this section, we combine Spectre with Collide+Probe to leak kernel memory without the requirement of shared memory. While some Spectre-type attacks use AVX [70] or port contention [13], most attacks use the cache as a covert channel to encode secrets [17, 41]. During transient execution, the kernel caches a user-space address based on a secret. By monitoring the presence of said address in the cache, the attacker can deduce the leaked value.

As AMD CPU's are not vulnerable to Meltdown [49], stronger kernel isolation [27] is not enforced on modern operating systems, leaving the kernel mapped in user space. However, with SMAP enabled, the processor never loads an address into the cache if the translation triggers a SMAP violation, *i.e.*, the kernel tries to access a user-space address [9]. Thus, an attacker has to find a vulnerable indirect branch that can access user-space memory. We lift this restriction by using Collide+Probe as a cache-based covert channel to infer secret values accessed by the kernel. With Collide+Probe, we can observe μTag collisions based on the secret value that is leaked and, thus, remove the requirement of shared memory, *i.e.*, user memory that is directly accessible to the kernel.

To evaluate Collide+Probe as a covert channel for a Spectre-type attack, we implement a custom kernel module containing a Spectre-PHT gadget as illustrated as follows:

```
1 if (index < bounds) { a = LUT[data[index] * 4096]; }
```

The execution of the presented code snippet can be triggered with an `ioctl` command that allows the user to control the *index* variable as it is passed as an argument. First, we mistrain the branch predictor by repeatedly providing an index that is in bounds, letting the processor follow the branch to access a fixed kernel-memory location. Then, we access an address that collides with the kernel address accessed based on a possible byte-value located at `data[index]`. By providing an out-of-bounds index, the processor

now speculatively accesses a memory location based on the secret data located at the out-of-bounds index. Using Collide+Probe, we can now detect if the kernel has accessed the address based on the assumed secret byte value. By repeating this step for each of the 256 possible byte values, we can deduce the actual byte as we observe μTag conflicts. As we cannot ensure that the processor always misspeculates when providing the out-of-bounds index, we run this attack multiple times for each byte we want to leak.

We successfully leaked a secret string using Collide+Probe as a covert channel on an AMD Ryzen Threadripper 1920X. With our unoptimized version, we are able to leak the secret bytes with a success rate of 99.5 % ($\sigma_{\bar{x}} = 0.19, n = 100$) and a leakage rate of 0.66 B/s ($\sigma_{\bar{x}} = 0.00043, n = 100$). While we leak full byte values in our proof-of-concept, other gadgets could allow to leak bit-wise, reducing the overhead of measuring every possible byte value significantly. In addition, the parameters for the number of mistrainings or the necessary repetitions of the attack to leak a byte can be further tweaked to match the processor under attack. To utilize this side channel, the attacker requires the knowledge of the address of the kernel-memory that is accessed by the gadget. Thus, on systems with active kernel ASLR, the attacker first needs to defeat it. However, as described in Section 5.2, the attacker can use Collide+Probe to derandomize the kernel as well.

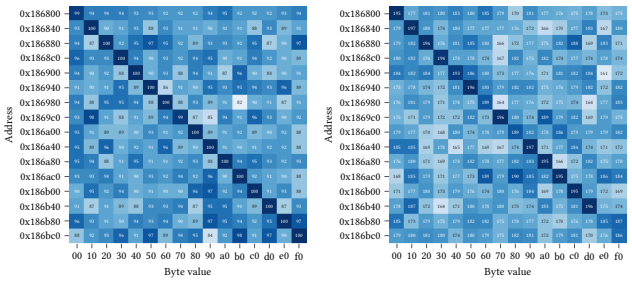
5.4 Attacking AES T-Tables

In this section, we show an attack on an AES [20] T-table implementation. While cache attacks have already been demonstrated against T-table implementations [30, 31, 48, 58, 72] and appropriate countermeasures, e.g., bit-sliced implementations [43, 62], have been presented, they serve as a good example to demonstrate the applicability of the side channel and allow to compare it against other existing cache side-channels. Furthermore, AES T-tables are still sometimes used in practice. While some implementations fall back to T-table implementations [21] if the AES-NI instruction extension [32] is not available, others only offer T-table-based implementations [45, 55]. For evaluation purposes, we used the T-table implementation of OpenSSL version 1.1.1c.

In this implementation, the *SubBytes*, *ShiftRows*, and *MixColumns* steps of the AES round transformation are replaced by look-ups to 4 pre-computed T-tables $T_0, \ldots, T_3$. As the *MixColumns* operation is omitted in the last round, an additional T-table T_4 is necessary. Each table contains 256 4-byte words, requiring 1 kB of memory.

In our proof-of-concept, we mount the first-round attack by Osvik et al. [58]. Let k_i denote the initial key bytes, p_i the plaintext bytes and $x_i = p_i \oplus k_i$ for $i = 0, \ldots, 15$ the initial state of AES. The initial state bytes are used to select elements of the pre-computed T-tables for the following round. An attacker who controls the plaintext byte p_i and monitors which entries of the T-table are accessed can deduce the key byte $k_i = s_i \oplus p_i$. However, with a cache-line size of 64 B, it is only possible to derive the upper 4 bit of k_i if the T-tables are properly aligned in memory. With second-round and last-round attacks [58, 73] or disaligned T-tables [72], the key space can be reduced further.

Figure 5 shows the results of a Collide+Probe and a Load+Reload attack on the AMD Ryzen Threadripper 1920X on the first key byte. As the first key byte is set to zero, the diagonal shows a higher number of cache hits than the other parts of the table. We repeated every experiment 1000 times. With Collide+Probe, we can successfully recover with a probability of 100 % ($\sigma_{\bar{x}} = 0$) the upper 4 bits of each k_i with 168 867 ($\sigma_{\bar{x}} = 719$) encryptions per byte in 0.07 s ($\sigma_{\bar{x}} = 0.0003$). With Load+Reload, we require 367 731 ($\sigma_{\bar{x}} = 82388$) encryptions and an average runtime of 0.53 s ($\sigma_{\bar{x}} = 0.11$) to recover 99.0 % ($\sigma_{\bar{x}} = 0.0058$) of the key bits. Using Prime+Probe on the L1 cache, we can successfully recover 99.7 % ($\sigma_{\bar{x}} = 0.01$) of the key bits with 450 406 encryptions ($\sigma_{\bar{x}} = 1129$) in 1.23 s ($\sigma_{\bar{x}} = 0.003$).



(a) Collide+Probe **(b) Load+Reload**

Figure 5: Cache access pattern with Collide+Probe and Load+Reload on the first key byte.

6 DISCUSSION

While the official documentation of the way prediction feature does not explain how it interacts with other processor features, we discuss the interactions with instruction caches, transient execution, and hypervisors.

Instruction Caches. The patent [23] describes that AMD's way predictor can be used for both data and instruction cache. However, AMD only documents a way predictor for the L1D cache [8] and not for the L1I cache.

Transient Execution. Speculative execution is a crucial optimization in modern processors. When the CPU encounters a branch, instead of waiting for the branch condition, the CPU guesses the outcome and continues the execution in a transient state. If the speculation was correct, the executed instructions are committed. Otherwise, they are discarded. Similarly, CPUs employ out-of-order execution to transiently execute instructions ahead of time as soon as their dependencies are fulfilled. On an exception, the transiently executed instructions following the exception are simply discarded, but leave traces in the microarchitectural state [17]. We investigated the possibility that AMD Zen processors use the data from the predicted way without waiting for the physical tag returned by the TLB. However, we were not able to produce any such results.

Hypervisor. AMD does not document any interactions of the way predictor with virtualization. As we have shown in our experiments (cf. Section 5.2), the way predictor does not distinguish between virtual machines and hypervisors. The way predictor uses the virtual address without any tagging, regardless whether it is a guest or host virtual address.

7 COUNTERMEASURES

In this section, we discuss mitigations to the presented attacks on AMD's way predictor. We first discuss hardware-only mitigations, followed by mitigations requiring hardware and software changes, as well as a software-only solution.

Temporarily Disable Way Predictor. One solution lies in designing the processor in a way that allows temporarily disabling the way predictor temporarily. Alves et al. [4] evaluated the performance impact penalty of instruction replays caused by mispredictions. By dynamically disabling way prediction, they observe a higher performance than with standard way prediction. Dynamically disabling way prediction can also be used to prevent attacks by disabling it if too many mispredictions within a defined time window are detected. If an adversary tries to exploit the way predictor or if the current legitimate workload provokes too many conflicts, the processor deactivates the way predictor and falls back to comparing the tags from all ways. However, it is unknown whether AMD processors support this in hardware, and there is no documented operating system interface to it.

Keyed Hash Function. The currently used mapping functions (Section 3) rely solely on bits of the virtual address. This allows an attacker to reverse-engineer the used function once and easily find colliding virtual addresses resulting in the same μTag. By keying the mapping function with an additional process- or context-dependent secret input, a reverse-engineered hash function is only valid for the attacker process. ScatterCache [77] and CEASAR-S [61] are novel cache designs preventing cache attacks by introducing a similar keyed mapping function for skewed-associative caches. Hence, we expect that such methods are also effective when used for the way predictor. Moreover, the key can be updated regularly, e.g., when returning from the kernel, and, thus, not remain the same over the execution time of the program.

State Flushing. With Collide+Probe, an attacker cannot monitor memory accesses of a victim running on a sibling thread. However, μTag collisions can still be observed after context switches or transitions between kernel and user mode. To mitigate Collide+Probe, the state of the way predictor can be cleared when switching to another user-space application or returning from the kernel. Every subsequent memory access yields a misprediction and is thus served from the L2 data cache. This yields the same result as invalidating the L1 data cache, which is currently a required mitigation technique against Foreshadow [74] and MDS attacks [16, 68, 75]. However, we expect it to be more power-efficient than flushing the L1D. To mitigate Spectre attacks [41, 44, 51], it is already necessary to invalidate branch predictors upon context switches [17]. As invalidating predictors and the L1D cache on Intel has been implemented through CPU microcode updates, introducing an MSR to invalidate the way predictor might be possible on AMD as well.

Uniformly-distributed Collisions. While the previously described countermeasures rely on either microcode updates or hardware modifications, we also propose an entirely software-based mitigation. Our attack on an optimized AES T-table implementation in Section 5.4 relies on the fact that an attacker can observe the key-dependent look-ups to the T-tables. We propose to map such secret data n times, such that the data is accessible via n different virtual addresses, which all have a different μTag. When accessing the data, a random address is chosen out of the n possible addresses. The attacker cannot learn which T-table has been accessed by monitoring the accessed μTags, as a uniform distribution over all possibilities will be observed. This technique is not restricted to T-table implementations but can be applied to virtually any secret-dependent memory access within an application. With dynamic software diversity [19], diversified replicas of program parts are generated automatically to thwart cache-side channel attacks.

8 CONCLUSION

The key takeaway of this paper is that AMD's cache way predictors leak secret information. To understand the implementation details, we reverse engineered AMD's L1D cache way predictor, leading to two novel side-channel attack techniques. First, Collide+Probe allows monitoring memory accesses on the current logical core without the knowledge of physical addresses or shared memory. Second, Load+Reload obtains accurate memory-access traces of applications co-located on the same physical core.

We evaluated our new attack techniques in different scenarios. We established a high-speed covert channel and utilized it in a Spectre attack to leak secret data from the kernel. Furthermore, we reduced the entropy of different ASLR implementations from native code and sandboxed JavaScript. Finally, we recovered a key from a vulnerable AES implementation.

Our attacks demonstrate that AMD's design is vulnerable to side-channel attacks. However, we propose countermeasures in software and hardware, allowing to secure existing implementations and future designs of way predictors.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their comments and suggestions that helped improving the paper. The project was 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. It was also supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work also benefited from the support of the project ANR-19-CE39-0007 MIAOUS of the French National Research Agency (ANR). Additional funding was provided by generous gifts 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] Andreas Abel and Jan Reineke. 2013. Measurement-based Modeling of the Cache Replacement Policy. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2018. Port Contention for Fun and Profit. In *S&P*.
[3] Alex Christensen. 2015. Reduce resolution of performance.now. https://bugs.webkit.org/show_bug.cgi?id=146531
[4] Ricardo Alves, Stefanos Kaxiras, and David Black-Schaffer. 2018. Dynamically disabling way-prediction to reduce instruction replay. In *International Conference on Computer Design (ICCD)*.

[5] AMD. 2013. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors.*
[6] AMD. 2014. *Software Optimization Guide for AMD Family 15h Processors.*
[7] AMD. 2017. AMD64 Architecture Programmer's Manual.
[8] AMD. 2017. *Software Optimization Guide for AMD Family 17h Processors.*
[9] AMD. 2018. Software techniques for managing speculation on AMD processors.
[10] AMD. 2019. 2nd Gen AMD EPYC Processors Set New Standard for the Modern Datacenter with Record-Breaking Performance and Significant TCO Savings.
[11] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. 2015. CAIN: Silently Breaking ASLR in the Cloud. In *WOOT*.
[12] Daniel J. Bernstein. 2004. Cache-Timing Attacks on AES.
[13] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*.
[14] Boris Zbarsky. 2015. Reduce resolution of performance.now. https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab
[15] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. 2016. Flush, Gauss, and Reload–a cache attack on the BLISS lattice-based signature scheme. In *CHES*.
[16] 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. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*.
[17] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*.
[18] Mike Clark. 2016. A new x86 core architecture for the next generation of computing. In *IEEE Hot Chips Symposium (HCS)*.
[19] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *NDSS*.
[20] Joan Daemen and Vincent Rijmen. 2013. *The design of Rijndael: AES-the advanced encryption standard.*
[21] Helder Eijs. 2018. PyCryptodome: A self-contained cryptographic library for Python. https://www.pycryptodome.org
[22] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*.
[23] W. Shen Gene and S. Craig Nelson. 2006. MicroTLB and micro tag for reducing power in a processor . US Patent 7,117,290 B2.
[24] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*.
[25] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*.
[26] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing* (1996).
[27] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *ESSoS*.
[28] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*.
[29] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*.
[30] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.
[31] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*.
[32] Shay Gueron. 2012. Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01.
[33] Richard W Hamming. 1950. Error detecting and error correcting codes. *The Bell system technical journal* (1950).
[34] Joel Hruska. 2019. AMD Gains Market Share in Desktop and Laptop, Slips in Servers. https://www.extremetech.com/computing/291032-amd
[35] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*.
[36] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. 1999. Way-predicting set-associative cache for high performance and low energy consumption. In *Symposium on Low Power Electronics and Design*.
[37] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *S&P*.
[38] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross processor cache attacks. In *AsiaCCS*.
[39] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*.
[40] Richard E Kessler. 1999. The alpha 21264 microprocessor. *IEEE Micro* (1999).
[41] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.
[42] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*.
[43] Robert Könighofer. 2008. A Fast and Cache-Timing Resistant Implementation of the AES. In *CT-RSA*.
[44] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*.
[45] Marcin Krzyzanowski. 2019. CryptoSwift: Growing collection of standard and secure cryptographic algorithms implemented in Swift. https://cryptoswift.io
[46] Linux. 2019. Complete virtual memory map with 4-level page tables. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt
[47] Linux. 2019. Linux Kernel 5.0 Process (x86). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/kernel/process.c
[48] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.
[49] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.
[50] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.
[51] G. Maisuradze and C. Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*.
[52] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Complex Addressing Using Performance Counters. In *RAID*.
[53] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*.
[54] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies The Power of Cache Attacks. In *CHES*.
[55] Richard Moore. 2017. pyaes: Pure-Python implementation of AES block-cipher and common modes of operation. https://github.com/ricmoo/pyaes
[56] Louis-Marie Vincent Mouton, Nicolas Jean Phillippe Huot, Gilles Eric Grandou, and Stephane Eric Sebastian Brochier. 2012. Cache accessing using a micro TAG. US Patent 8,151,055.
[57] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*.
[58] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.
[59] Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*.
[60] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*.
[61] Moinuddin K Qureshi. 2019. New attacks and defense for encrypted-address cache. In *ISCA*.
[62] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. 2006. Bitslice Implementation of AES. In *Cryptology and Network Security (CANS)*.
[63] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*.
[64] David J Sager and Glenn J Hinton. 2002. Way-predicting cache memory. US Patent 6,425,055.
[65] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. 2019. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725* (2019).
[66] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*.
[67] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*.
[68] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.
[69] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*.
[70] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*.

Table 3: `rdtsc` increments on various CPUs.

Setup	CPU	μ-arch.	Increment
Lab	AMD Athlon 64 X2 3800+	K8	1
Lab	AMD Turion II Neo N40L	K10	1
Lab	AMD Phenom II X6 1055T	K10	1
Lab	AMD E-450	Bobcat	1
Lab	AMD Athlon 5350	Jaguar	1
Lab	AMD FX-4100	Bulldozer	1
Lab	AMD FX-8350	Piledriver	1
Lab	AMD A10-7870K	Steamroller	1
Lab	AMD Ryzen Threadripper 1920X	Zen	35
Lab	AMD Ryzen Threadripper 1950X	Zen	34
Lab	AMD Ryzen Threadripper 1700X	Zen	34
Lab	AMD Ryzen Threadripper 2970WX	Zen+	30
Lab	AMD Ryzen 7 3700X	Zen 2	36
Cloud	AMD EPYC 7401p	Zen	20
Cloud	AMD EPYC 7571	Zen	22

[71] Mark Seaborn. 2015. How physical addresses map to rows and banks in DRAM. http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html

[72] Raphael Spreitzer and Thomas Plos. 2013. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *COSADE*.

[73] Junko Takahashi, Toshinori Fukunaga, Kazumaro Aoki, and Hitoshi Fuji. 2013. Highly accurate key extraction method for access-driven cache attacks using correlation coefficient. In *ACISP*.

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

[75] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[76] VMWare. 2018. Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing (2080735). https://kb.vmware.com/s/article/2080735

[77] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*.

[78] Felix Wilhelm. 2016. PoC for breaking hypervisor ASLR using branch target buffer collisions. https://github.com/felixwilhelm/mario_baslr

[79] Henry Wong. 2013. Intel Ivy Bridge Cache Replacement Policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/

[80] John C Wray. 1992. An analysis of covert timing channels. *Journal of Computer Security* 1, 3-4 (1992), 219–232.

[81] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *S&P*.

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

[83] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-oriented flush-reload side channels on arm and their implications for android devices. In *CCS*.

[84] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS*.

[85] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*.

A RDTSC RESOLUTION

We measure the resolution of the `rdtsc` instruction using the following experimental setup. We assume that the timestamp counter (TSC) is updated in a fixed interval. This assumption is based on the documentation in the manual that the timestamp counter is independent of the CPU frequency [8]. Hence, there is a modulus x and a constant C, such that TSC $\mod x \equiv C$ iff x is the TSC increment. We can easily find this x with brute-force, *i.e.*, trying all different x until we find an x, which always results in the same value C. Table 3 shows a `rdtsc` increments for the CPUs we tested.

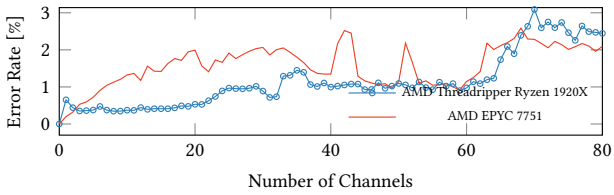


Figure 6: Error rate of the covert channel.

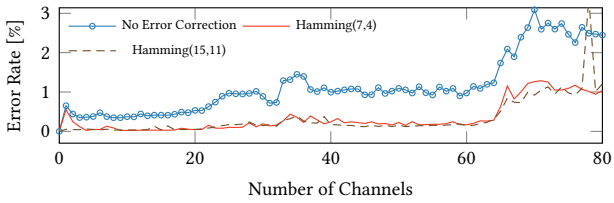


Figure 7: Error rate of the covert channel with and without error correction using different Hamming codes.

B COVERT CHANNEL ERROR RATE

Figure 6 illustrates the error rate of the covert channel described in Section 5.1. The error rate increases drastically when transmitting more than 64 bits in parallel. Thus, we evaluated different hamming codes on an AMD Ryzen Threadripper 1920X (Figure 7).

C USERSPACE ASLR

Linux also uses ASLR for user processes by default. However, randomizing the code section requires compiler support for position-independent code. The heap memory region is of particular interest because it is located just after the code section with an offset of up to 32 MB [47]. User programs use 4 kB pages, giving an effective 13-bit entropy for the start of the brk-based heap memory.

It is possible to fully break heap ASLR through the use of μTags. An attack requires an interface to the victim application that incurs a victim access to data on the heap. We evaluated the ASLR break using a client-server scenario in a toy application, where the attacker is the malicious client. The attacker repeatedly sends benign requests until it is distinguishable which tag is being accessed by the victim. This already reduces the ASLR entropy by 8 bits because it reveals a linear combination of the address bits. It is also possible to recover all address bits up to bit 27 by using the μTags of multiple pages and solving the resulting equation system.

Again, a limitation is that the attack is susceptible to noise. Too many accesses while processing the attacker's request negatively impact the measurements such that the attacker will always observe a cache miss. In our experiments, we were not able to mount the attack using a socket-based interface. Hence, attacking other user-space applications that rely on a more complex interface, e.g., using D-Bus, is currently not practical. However, future work may refine our techniques to also mount attacks in more noisy scenarios. For our evaluation, we targeted a shared-memory-based API for high-speed transmission without system calls [26] provided by the victim application. We were able to recover 13 bits with an average success rate of 88.9 % in 1.94 s (σ = 1.76 s, n = 1000).