Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP

Lukas Giner[1], Sudheendra Raghav Neela[1], and Daniel Gruss[1]

Graz University of Technology
{first.last}@tugraz.at

Abstract. Confidential computing platforms, e.g., AMD SEV-SNP, allow running mutually distrusting workloads on the same hardware with the protection of several isolation mechanisms: data is encrypted in RAM, and access to unencrypted data is architecturally prevented. Furthermore, access and cache line operations are restricted, mitigating attacks like Flush+Reload. The hypervisor can access the encrypted data of virtual machines, e.g., for migration purposes. This creates a coherency challenge around modifications between encrypted and decrypted cache lines. AMD enforces coherency between these two cache lines by removing one when the other is *accessed*.
In this paper, we present Cohere+Reload, a novel side-channel attack exploiting AMD's coherency for encrypted memory. We discover two types of leakage in the coherency mechanism: First, coherence conflicts leak victim operations on a spatial granularity of a 2 kB block. Second, the timing correlates with number and location of accesses the victim performed within the confidential virtual machine, allowing to infer how often or where within a coherence partition victim accesses were performed, with a maximum spatial resolution of 256 bytes. We evaluate Cohere+Reload in two synthetic and two real-world attacks: In synthetic attacks, we demonstrate that Cohere+Reload can observe the control flow and access locations in workloads within a confidential virtual machine. We present a real-world attack on mbedTLS RSA, leaking 4096 key bits in a single-trace attack, with 99.7 % of bits correct. We present another real-world attack on OpenSSL AES exploiting disalignments on a cache line granularity: In a first round T-table attack we achieve an accuracy of 100 % in only 1500 encryptions and with a novel correlation attack an accuracy of 92.81 % in 12000 encryptions. We conclude that the coherence approach for AMD SEV-SNP should be re-evaluated and discuss further potential mitigations.

1 Introduction

Modern processors have a multi-layered memory hierarchy for data, including code. Data can reside in registers, in cache lines in L1, L2, or L3 cache, or in the RAM. Some processors have even further cache layers, e.g., an L4 cache. While caches are crucial for the performance of modern computers, they also inherently introduce timing side channels that distinguish cached from non-cached data.

The most widely known attacks are Prime+Probe [34] and Flush+Reload [45]. Flush+Reload [45] works by constantly flushing a cache line from the cache, using the processor's flush instruction, and measuring how long it takes to reload the cache line. Flushing a cache line requires read access to the memory, e.g., read-only shared memory with the victim, which is typically not available across virtual machines or in the context of confidential computing. Prime+Probe does not require shared memory. Prime+Probe [34] works by measuring how much time it takes to constantly re-fill a specific cache set. If a victim access falls into the same cache set, the timing increases.

Confidential computing is an emerging compute paradigm where a confidential workload, running isolated inside a virtual machine, is isolated from all other workloads and from the host. More specifically, it is part of the threat model that the hypervisor can be malicious or compromised but the confidential virtual machine remains secure. Confidential computing platforms, e.g., Intel TDX and AMD SEV-SNP, still share the underlying hardware across mutually distrusting workloads running in virtual machines. However, vendors introduced several isolation mechanisms to protect workloads: For instance, data is encrypted in RAM, is decrypted on-the-fly when moved into the caches, and the processor prevents direct access to unencrypted data in caches or registers. The hypervisor cannot access unencrypted memory of the confidential virtual machine. Furthermore, cache line operations are restricted, mitigating attacks like Flush+Reload.

Despite the strong isolation, some functionality requires access from the hypervisor to the encrypted data, e.g., migration of virtual machines in the cloud. Consequently, the hypervisor can access the encrypted data of virtual machines. However, this implies that data can be twice in the caches: once encrypted for the host, and once unencrypted for the confidential virtual machine. Clearly, this creates a coherency challenge as virtual machine may modify a cache line, *i.e.*, the cache contains a modified unencrypted cache line and an outdated encrypted cache line. Google reported that there is a coherency mechanism on Intel TDX [1] for this purpose, where accesses with one key flush *all* other copies of the address with different keys from the cache. AMD pursued a different approach by enforcing coherency between the unencrypted and encrypted cache lines by removing one when the other is *accessed*. Still, AMD does not operate on the granularity of a cache line, as we show in this work.

In this paper, we present Cohere+Reload, a novel attack exploiting that AMD's coherency approach introduces a surprisingly powerful side channel. We thoroughly analyze AMD's coherence mechanism for encrypted memory and discover two properties that form the basis of our Cohere+Reload attack: First, there is a significant timing difference between cache hits and coherence conflicts on a spatial granularity of a 2 kB coherence partition, *i.e.*, half a page. This timing difference directly reveals whether a victim confidential virtual machine just accessed a specific memory location. Second, the amplitude of the timing coarsely correlates with the number of accesses the victim performed. It is also more finely correlated with the location of single victim accesses, allowing to infer which out of 8 alignments within a coherence partition the victim access

had, *i.e.*, we have a maximum spatial resolution of 256 bytes, which is in the same order of magnitude as Flush+Reload with a spatial resolution of 64 bytes.

We evaluate Cohere+Reload in two synthetic and two real-world attacks: The two synthetic attacks demonstrate that Cohere+Reload can observe the control flow in workloads within a confidential virtual machine, *i.e.*, identify the target of a jump; and that Cohere+Reload can observe which data locations a workload within a confidential virtual machine accessed, naturally within the limits of the spatial resolution of Cohere+Reload. We present an attack on mbedTLS RSA-4096 and show that we can leak all 4096 key bits in a single-trace attack, with a Levenshtein distance of less than 11 bits on average. Finally, we present a novel attack on OpenSSL AES that exploits disalignments on a cache line granularity. Based on this insight, we mount a first round attack with and accuracy of 100 % in only 1500 encryptions. We recover all upper nibbles of AES with a novel correlation attack with an accuracy of 92.81 % in 12000 encryptions. We conclude that the coherence approach for AMD SEV-SNP should be re-evaluated and discuss further potential mitigations.

Disclosure. We responsibly disclosed our results to AMD and are waiting for their response.

Contributions. In summary, our main contributions are:

- We introduce Cohere+Reload, a novel attack exploiting that AMD's coherency with a spatial granularity of a 2 kB per coherence partition, and 8 distinguishable alignments within a partition, yielding a maximum spatial resolution that is on par with Flush+Reload.
- We evaluate Cohere+Reload in two synthetic attacks demonstrating that we can leak control flow and data access from a confidential virtual machine on AMD SEV-SNP.
- We present an attack on mbedTLS RSA-4096 and show that we can leak all 4096 key bits in a single-trace attack, with a Levenshtein distance of less than 11 bits on average.
- We present a novel attack on OpenSSL AES that exploits disalignments on a cache line granularity, yielding an accuracy of 100 % in only 1500 encryptions in a first-round attack and an accuracy of 92.81 % in 12000 encryptions in a novel correlation attack.

Outline. We provide background in Section 2. We define our threat model in Section 3. We present our novel Cohere+Reload attack in Section 4. We template target pages in Section 5. We present an attack on mbedTLS RSA in Section 6 and an attack on OpenSSL AES T-Tables in Section 7. We present an attack on control flow and data accesses in Section 8. We conclude in Section 10.

2 Background

In this section, we provide background on trusted-execution environments, side-channel attacks, and coherence in the context of memory encryption.

2.1 Trusted-Execution Environments

The goal of trusted-execution environments (TEEs) is to provide confidentiality and integrity for code and data on a system even on a compromised system [22,2,23,5]. Older TEEs often focus on personal and mobile computers, e.g., Intel Software Guard Extensions (SGX) [22]. The TEE runs a small trusted workload in a signed enclave [22]. These enclaves run on the same CPU as regular applications. To prevent access from a compromised host system, SGX prevents access to the encrypted enclave memory and register state.

More recent TEEs focus on cloud use cases and virtual machines (VMs). Instead of protecting a small workload, the idea is to move entire VMs into the TEE, which are then called confidential virtual machines (CVMs) and protect them from a malicious or compromised host [9], e.g., AMD Secure Encrypted Virtualization (SEV) [3] and Intel Trust Domain Extensions (TDX) [21].

AMD SEV protects memory contents of CVMs by encrypting any data moved out of the CPU, e.g., to DRAM or disk [26]. Still, there are many attacks on SEV. In particular the basic SEV design was demonstrated to provide too little protection for the guest state [18,42] and memory [18,11,32,43]. AMD addressed this issue with with the *Encrypted State (ES)* and *Secure Nested Paging (SNP)* SEV extensions, protecting guest state and memory integrity.

Like AMD SEV-SNP, Intel supports CVMs through their Trust Domain Extensions (TDX) [23]. Guest memory and state are encrypted and managed by the TEE. The host can only interact with the guest through well-defined secure interfaces. For fast inter-process communication, the memory has both private encrypted parts and shared parts that are equally accessible to the host.

2.2 Side-Channel Attacks

Side channels can be used to attack systems even if there are no software or hardware vulnerabilities or they are not known. Side channels instead exploit side effects of the implementation such as timing [27], power consumption [28], or radiation [36]. Older works focused on cryptographic primitives [27,6,8], leaking keys of vulnerable cryptographic implementations of e.g., AES [6,34], RSA [45,8], or ECDSA [44]. More recent works often focus on larger systems to leak information from one system component, e.g., kernel information [20], user input [38,17,31], and system activity [15].

Many side-channel attacks target caches as they can be probed without privileges at a high temporal (*i.e.*, nanosecond to microsecond range) and spatial resolution (*i.e.*, 64 B) while being comparably robust against noise. Most importantly, they allow for use generic attacks that are not tailored to specific applications and victim programs. Consequently, the community developed a set of generic attack techniques that follow a uniform naming pattern based on the attack components, e.g., Prime+Probe and Flush+Reload. One of the first generic attack techniques was Evict+Time [34], in which an attacker runs and times a victim process twice, once with evicting a target cache line from the cache by performing a larger number of memory accesses that collide in the

cache, e.g., due to set associativity, and once without. A statistically higher execution time means the cache line was used by the victim. Instead of timing the victim, Prime+Probe [34] times the (evicting) memory accesses, *i.e.*, they time how long it takes to (re-)prime a cache set. If it takes more time, more cache lines were replaced by the victim execution. Prime+Probe is one of the most widely used attack techniques besides Flush+Reload [45]. In the Flush+Reload attack, an attacker flushes a cache line using a dedicated flush instruction, and measures the time it takes to reload the memory location in order to decide whether or not the victim used it. Variations of Flush+Reload include Evict+Reload [14], which substitutes eviction for flushing, and Flush+Flush [16], which measures the timing of the flush instruction instead of the reload, thereby revealing a similar timing difference. Similarly, for Prime+Probe, several attack techniques and variations have been presented more recently, such as Prime+Abort [10], Prime+Scope [35], and Spec-o-Scope [19].

2.3 Coherence Between Ciphertext and Plaintext

Modern CPUs feature memory encryption technology, such as Intel's Total Memory Encryption (TME) [24] and AMD's Secure Memory Encryption (SME) [26]. Memory encryption is a crucial aspect of trusted execution environments, including Intel TDX and AMD SEV, designed to safeguard sensitive data. These built-in memory encryption systems encrypt data before it is written to the main memory and decrypt it when loaded into the CPU caches.

Given the nature of trusted computing, the untrusted hypervisor must interact with ciphertexts to facilitate operations such as migrating the guest machine to another server. To enable direct access to encrypted data, AMD's encryption unit includes a short-circuit path that forwards the data without decrypting it. Each data access's physical address encodes a so-called encrypted bit (C-bit), which indicates whether this short-circuit path should be utilized. This leads to an important question: how is coherence maintained when the hypervisor actively requests ciphertext while the guest is processing plaintext?

AMD states that coherency between the ciphertext and plaintext depends on the hardware [4] as some hardware enforce coherency while others do not. In the systems where coherence is not enforced, the hypervisor must flush the encrypted data from all CPU caches. In other systems, hardware supports coherency across encryption domains and software does not have to flush encrypted data, and the presence of this feature can be determined by the CPUID bit "CoherencyEnforced" (see AMD Architecture Programmer's Manual [4], 7.10.6).

In our systems, all AMD EPYC CPUs support SEV-SNP and include this coherence feature. Consumer Ryzen CPUs only support SME [29] without automatic hardware coherence between ciphertext and plaintext. For SEV systems without SNP, not having hardware coherence poses significant risks, allowing potential fault-attack-like exploitation during the write-back of ciphertext.

With the introduction of AMD SEV-SNP, the hypervisor can no longer directly write to an encrypted page [2]. Each guest page undergoes a procedure to assign it in a reverse page map, indicating its ownership to a given guest VM.

Once a guest accepts a page, the hypervisor retains only read access, which is ciphertext. Despite these advancements, maintaining coherence between ciphertext and plaintext remains essential. On Intel TDX, it is known that accesses to a ciphertext flushes all other copies of the address from the cache [1]. In Section 4, we present our initial analysis of how coherence is managed on AMD systems.

3 Threat Model

Exploiting the Cohere+Reload mechanism requires ciphertext view and a plaintext view on the same memory region. Outside of SME, this can only happen when a hypervisor maps an SEV guest page (ciphertext view), as guests have no option to map pages outside their allocated memory. Our threat model is therefore a malicious hypervisor trying to extract information from an encrypted SEV, SEV-ES or SEV-SNP guest. In this scenario, the hypervisor has control over all parts of the CPU that are not part of the attestation. This includes control over CPU frequency, disabling hardware prefetching and selecting a suitable DRAM interleaving setting at boot (see Section 4.1). While Cohere+Reload attacks can be performed even without stabilizing the frequency or disabling prefetching, like all cache attacks, it is enhanced by these settings and they will be used throughout the paper.

4 Cohere+Reload

In this section we examine the behaviour of coherence for AMD memory encryption. In all of our tests, hardware cache coherence works the same in SME as it does in SEV, SEV-ES or SEV-SNP. Therefore we will conduct all basic experiments in SME for simplicity, unless specifically mentioned.

As a first step, we configure our systems (cf. Table 1) for transparent secure memory encryption (TSME). This means all pages will be encrypted by default, denoted by a bit in the physical address, e.g., bit 51.To get a ciphertext view of a page, we create a second mapping where this bit is not set. When we now measure access times to a cache line in the ciphertext mapping, we can clearly distinguish three cases: hits, misses and coherence conflicts (see Figure 1). We cause a normal miss by flushing the line with `clflush` before measuring it, and a coherence conflict by accessing the same line in the plaintext mapping. We attribute the latency increase to the fact that when there is a plaintext line to evict, this has to happen before the load is completed, to ensure coherency. We also observe, as expected, that this effect is entirely symmetrical; it does not matter which mapping is used as the observer. The coherence also holds for code pages that were cached through code execution.

This basic hit/conflict behaviour constitutes the first part of the Cohere+Reload primitive (see Section 4.2 for the second).

Table 1. Test systems.

CPU	Architecture	SME	HW Coherence	SEV	VM page flush MSR
2x AMD EPYC 7443	Zen 3	✓	✓	SNP	✓
AMD EPYC 7313P	Zen 3	✓	✓	SNP	✓
AMD EPYC 8024P	Zen 4c	✓	✓	SNP	X

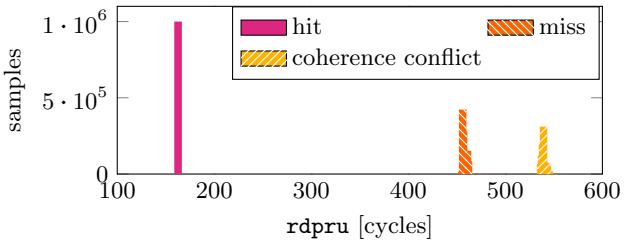


Fig. 1. Access timing histogram for accesses that are hits, misses (flushed) or conflicts caused by SME coherence.

4.1 Eviction Pattern

Contrary to Intel TDX, AMD memory encryption does not enforce its coherence with single line granularity. Instead, we find that any access *always* triggers the eviction of 32 out of 64 cache lines on a 4 kB aligned section of memory. We notice that in all of our machines' default configurations, the page is not simple split into two contiguous 2048 B halves, but instead shows an alternating pattern of 256 byte (4 cache lines) *coherence blocks* between the two *coherence partitions* (see Figure 2a). Concretely, this means that an access to one or multiple plaintext (or ciphertext) addresses in the first (or second) coherence partition of a page will always trigger an eviction of *all* ciphertext (or plaintext) addresses in the first (or second) partition of a page. This limits the channel's spatial resolution compared to Flush+Reload, though it speeds up page profiling (see Section 5). We run this experiment on different physical and virtual pages, different page sizes (4 kB and 2 MB) and between different cores. We find that the pictured pattern is always the same. However, two of our machines' (EPYC 7443 and EPYC 7313P) mainboard menus expose a boot setting for "*DRAM interleaving size*". When we change it from its default of 256 B to 512 B, 1024 B or 2048 B, we can see the coherence eviction pattern changing to match (see Figure 2), except for "off", 1024 B in the case of the EPYC 7443 system, and 4096 B (Table 2). While we do not know why the coherence mechanism is implemented as it is, we suspect some form of load balancing consideration w.r.t. DRAM.

4.2 Access Delay Time

Since we have seen in Section 4 that the presence of a single plaintext cache line increases the access time for a ciphertext line in the same coherence partition, it stands to reason that more cache lines in the same partition might take even longer to evict. Indeed, we find that the access delay on the evicting party's

Table 2. DRAM interleaving size and coherence pattern block size on our two systems.

DRAM interleaving setting	off	256	512	1024	2048	4096
block size EPYC 7443	2048	256	512	1024	2048	256
block size EPYC 7313P	2048	256	512	2048	2048	256

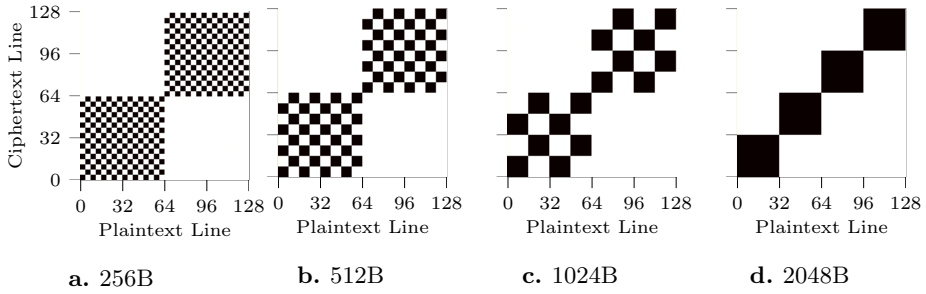


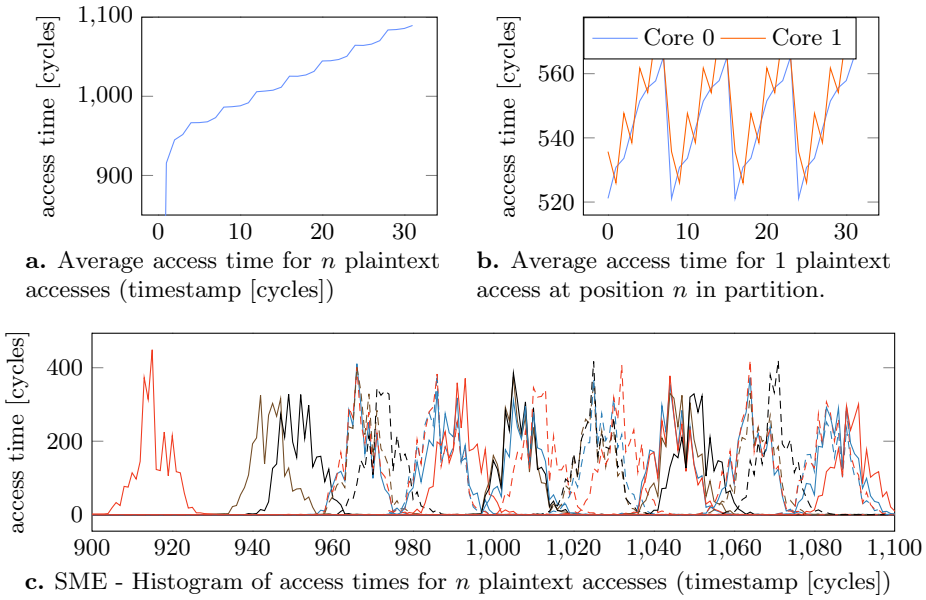
Fig. 2. Eviction Pattern for different *DRAM interleaving size* setting over 8 kB physically contiguous memory. A plaintext cache line is evicted by (and evicts) all corresponding ciphertext cache lines in black.

side is related to the number of accessed lines in the coherence partition. When we access 0-32 lines of plaintext in the same coherence partition and measure a ciphertext access we see a monotonically increasing access time (Figure 3a). But we don't observe a strictly monotonic increase, instead we see plateaus every 4 cache lines. When we look at the individual distribution of access times for each number of accesses (Figure 3c), these groupings are fairly clear.

Investigating further, we can see that Figure 3 is actually a special case of timings when the accessed cache lines are contiguous, *i.e.*, we do not skip addresses within a coherence partition up to our chosen number of accesses. Measuring the ciphertext access times for single plaintext evictions of different plaintexts we find the cause of this behaviour: different offsets within a coherence block have distinct timings.

As Figure 3 shows, when there is only one plaintext access coherence domain, the access time of the ciphertext depends on the position of the plaintext access withing the coherence block and repeats from block to block. This pattern depends on the core number *within a core complex* that loaded the plaintext address, but is the same between core complexes. We believe the pattern comes from the topology of the core complexes. When the block size is increased, the timing pattern also expands, though only up to 8 lines. For pattern sizes of 1024 B and larger, the timing pattern begins to repeat after 512 B, *i.e.*, 8 cache lines.

This timing behaviour is the second aspect of the Cohere+Reload primitive. We will further explore this effect in an attack in section Section 7.2.



a. Average access time for n plaintext accesses (timestamp [cycles])

b. Average access time for 1 plaintext access at position n in partition.

c. SME - Histogram of access times for n plaintext accesses (timestamp [cycles])

Fig. 3. Ciphertext conflict access times for prior plaintext accesses on congurent adresses.

4.3 Cohere+Reload Compared to Other Cache Attacks

In this section, we compare Cohere+Reload with two cache attacks: Flush+Reload and Flush+Flush. Our results, presented in Table 3, show that Cohere+Reload is a fast attack, comparable to the two Flush-based cache attacks across three metrics: hit time, miss time (conflict time), and blind spots. Using the methodology presented in prior work [37], we measure each metric 100 000 times on three systems: 2x EPYC 7443 (Zen 3), EPYC 7313P (Zen 3), and EPYC 8024P (Zen 4c). We disabled the hardware prefetchers and fixed the frequency on all three machines. To measure the metrics, we spawn two threads on different physical cores: a victim thread which randomly accesses a predetermined memory location, and an attacking thread that mounts the attack, measuring the metric.

On all three systems, we notice that Flush+Reload has a large blind spot — between 65-75% of victim accesses were missed by the attacker. Flush+Flush has a much smaller blind spot, with only 1.5-3.5% of victim accesses being missed by the attacker. Cohere+Reload has a minuscule blind spot, with less than 0.5% of victim accesses being missed by the attacker.

To measure the attack time, we consider both the hit and miss (conflict) timings. We see that the hit timings of Cohere+Reload are comparable to the hits of Flush+Reload, and the conflict timings of Cohere+Reload are comparable to Flush+Flush. This shows that Cohere+Reload is as fast as comparable attacks with a much smaller blind spot size, making it a very reliable side-channel attack.

Table 3. Comparison of Cohere+Reload with Flush+Reload and Flush+Flush.

	Flush+Reload			Flush+Flush			Cohere+Reload		
System	Hits [cycles]	Misses [cycles]	Blind Spots [%]	Hits [cycles]	Misses [cycles]	Blind Spots [%]	Conflict [cycles]	Hit [cycles]	Blind Spots [%]
EPYC 7443	122 σ=13	389 σ=130	65.1%	482 σ=96	367 σ=90	3.2%	428 σ=103	125 σ=130	0.06%
EPYC 7313P	150 σ=0	658 σ=173	73.2%	833 σ=124	632 σ=31	3.5%	800 σ=167	151 σ=0	0.53%
EPYC 8024P	145 σ=0	497 σ=117	74.8%	623 σ=92	484 σ=59	1.6%	623 σ=131	146 σ=1	0.02%

We compare Cohere+Reload with Flush+Reload and Flush+Flush across three metrics: hit time, miss/conflict time, and blind spots. The measurement for all three metrics is repeated 100 000 times on each system on different physical cores. Cohere+Reload has a much smaller blind-spot size and comparable hit and conflict times.

5 Target Page Templating

In a standard SEV-SNP scenario, the host has no knowledge about where the guest maps which data in its virtual memory range. For an attack, the first step is therefore to locate pages of interest in the guest. The same result could be achieved with page access flags, though this is an alternative approach that doesn't require the flushing of TLB entries. The only requirement for this step is that a victim page access can be reliably triggered (e.g., establishing a connection that causes an RSA encryption).

We implement this for RSA by using a network call to the guest that triggers an encryption. In our test, we filter 4 GB of VM virtual memory with a sieve of sorts. Starting with all pages, we repeatedly cause the guest to access or not access the page of interest while measuring each page from different core with Cohere+Reload. We access each page with a single split load to detect coherence eviction in both coherence partitions at once, as we find that the penalty for accessing both partitions is only $\approx$ 30 cycles. Pages that do not show the expected hits or conflicts are discarded from the list and the next step operates on the reduced list. After only 4 sieve steps, 1 048 510 pages can be reduced to an average of 7.36 pages in 10.6 s in 100 experiments, with the two RSA pages of interest always being among them. Finding the correct page from there is trivial, as only those two pages show the expected access pattern during an encryption (see Section 6).

6 High Frequency Code Attack - RSA

We attack the square-and-multiply `mbedtls_mpi_exp_mod` implementation of RSA in Mbed-TLS v3.0.0. For the purposes of this demonstration, we configure it to use a maximum window size of 1. While the specific version is not crucial as long as the algorithm is the same, note that because of the coherence pattern, Cohere+Reload requires a suitable code layout. That is, the code that can be

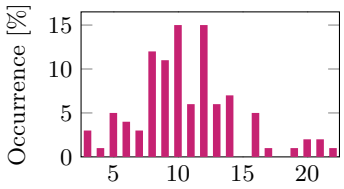


Fig. 4. Levenshtein distances over 100 single-trace RSA 4096 bit key recoveries.

attacked needs to be aligned suitable within a coherence partition, while code that would hinder the attack needs to fall into the other partition. In this example, we find that a 256 B pattern is unsuitable, but switching to a 512 B pattern with the DRAM interleaving setting results in a working attack.

The attacker is a program on the host system, while our victim is an RSA encryption service in an SEV-SNP guest triggered by the attacker. For the attack, the host program records traces of two code locations. First, the second partition in the `mbedtls_mpi_exp_mod` function. This contains the beginning of the loop that iterates over each key bit. Second, we trace the `mpi_montmul` function that does the squaring *and* multiplication operations. Our attack traces starts when the `mbedtls_mpi_exp_mod` is called for the first time and records long enough to capture the entire encryption (240000 samples). Figure 5 shows a section of such a trace. The `mbedtls_mpi_exp_mod` signal carries most of the key information, as it ideally detects an eviction precisely once per processed bit. This lets us infer whether or not `mpi_montmul` was called in addition to the square function (indicating that the key bit was 1) by the time delay to the next detection.

As we can see in Figure 6, the time difference between two conflicts in this signal is about twice as long when a '1' bit is processed. While this alone allows us to recover most of the key, we can correct some mistakes with the signal in `mpi_montmul` (pink). As the algorithm spends most of its time in this function, we detect almost all conflicts. However, when the algorithm is paused for any reason (e.g., scheduling), we see periods of hits on this address that we can then use to correct the primary signal. Figure 5 shows one occurance of this near the end. In minor post-processing we also detect the precise start and end of the encryption and remove double detections for single bits that are too close together (see Figure 6). Over 100 runs, our attack recovers randomly generated 4096 bit keys with a Levenshtein distance of 10.7 ± 3.94 (μ, σ) with a single trace (see Figure 4). In terms of attack performance, this is on par with related works attacking RSA [39,33,41,12,13].

7 AES T-Tables

In this section, we evaluate Cohere+Reload on the AES T-table implementation of OpenSSLv3.4 with 128 bit keys. Specifically, the `AES_encrypt` function which uses T-tables in lieu of hardware support (*i.e.*, AES-NI). Similar to the RSA attack (Section 6), we choose this implementation as it has been used extensively

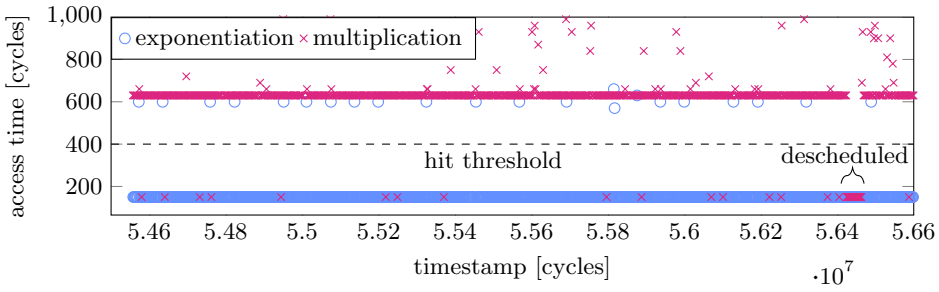


Fig. 5. Part of a raw Cohere+Reload trace of an RSA encryption. When *exponentiation* shows a conflict, a new exponentiation loop was started. When *multiplication* shows many in a row, the victim was most likely descheduled.

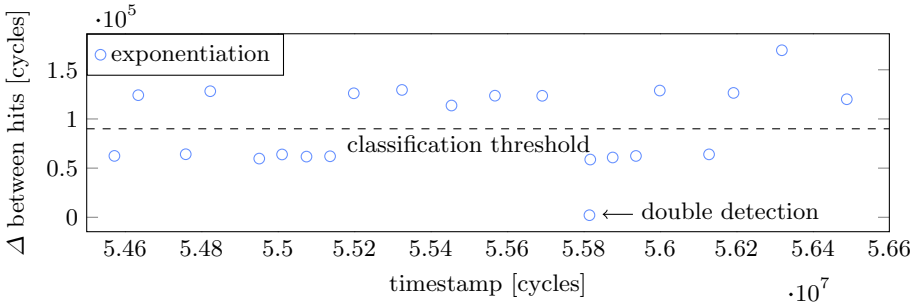


Fig. 6. Time difference between hits on *exponentiation*. The two bands show where a multiplication was executed (large difference) and the key bit is 1 or where it was not and the bit is 0 (small difference).

to evaluate prior side-channel attacks and is therefore a well-understood attack target [39,46,33,30,41,12,13].

The well-known first- and last-round attacks on AES T-tables [6,7,25] are both based on access probabilities. In the case of a first-round cache attack, each of the four T-tables' cache lines (16 per table with 16 entries each) are measured as hits or conflicts after an entire encryption. As an encryption consists of 40 total accesses to each table (10 rounds with each 4 accesses), over a sufficient number of random plaintexts the probability for each line to be accessed at the end of an encryption is $1 - \frac{15}{16}^{40} = 92.43\%$. This can be distinguished from lines that are *always* accessed. In the first round of the encryption, the tables are accessed according to the result of $P_n \oplus K_n$, where K_n is byte n in the original key and P is the plaintext. Fixing one plaintext byte therefore allows an attacker to control which line is always accessed. After measuring enough encryptions, only this line will show no conflicts, hence we can infer the upper nibble of each key byte.

The resolution of Cohere+Reload, however, is not a single cache line. The partition size of half a page is simply not enough to perform this attack merely

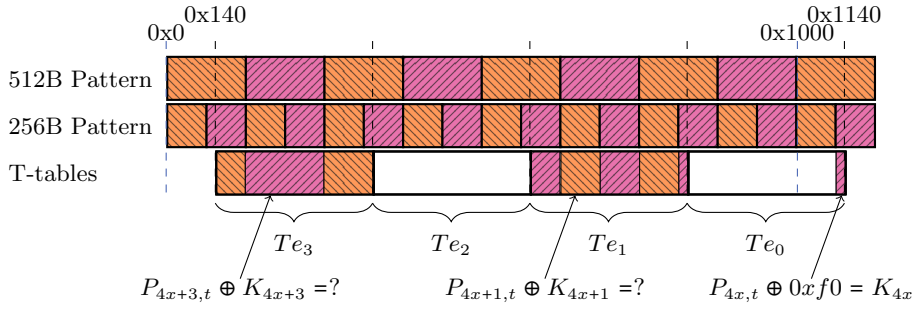


Fig. 7. AES T-table memory alignment in OpenSSL and memory coherence partition patterns for 256 B and 512 B. Annotations show where first round T-tables are accessed depending on the key- and plaintext bytes. Te_1 shows the 256 B pattern, Te_3 demonstrates 512 B and Te_0 shows the second partition on the next page.

by distinguishing hits from conflits, and even other tables on the same page influence each other. Even if one performed enough measurements to detect a non-accessed coherence partition (quite unlikely with $P_{accessed} = 1 - \frac{1}{2}^{160}$), this would only leak one bit per key byte.

However, we notice that while the default T-table placement in memory is aligned to cache lines with default compilation options, it is not necessarily aligned to a page. From this simple fact, we are able to conduct a limited standard first-round attack (Section 7.1) as well as a novel variation on the first-round attack based on a bias in the number of accessed cache lines instead of their location (Section 7.2).

7.1 Disaligned T-Table First-Round Attack

For this first attack, we only look at table Te_0, which deals with key/plaintext bytes 0,4,8 and 12 in the first round. As we can see in Figure 7 bottom right, Te_0 spans 5 cache lines into a new page. For a Flush+Reload attack, this would not make a difference, as the attack either works on all cache lines or it does not work at all (e.g., because the target cannot be accessed). For Cohere+Reload however, this enables an attack. With a coherence pattern size of 256 B, this means that the last line of Te_0 is the only one that accesses the second coherence partition on that page. With this disalignment we can convert the Cohere+Reload primitive into what is essentially a Flush+Reload primitive for this implementation, an improvement in the same vein as the attack described by Spreitzer et al. [40].

For random plaintexts, each cache line will be used by an encryption in 92 % of cases. In our case, the second coherence partition of the second page will measure this percentage for the last line of Te_0. The T-table access in the first round depends only on the XOR of the plaintext and key bytes, e.g., $Te_0 = P_{\{0,4,8,12\}} \oplus K_{\{0,4,8,12\}}$. The correct key nibble can thus be derived with $P_{x,t} \oplus 0xf0 = K_x$ from the test plaintext $P_{x,t}$ for which Cohere+Reload shows a 100 % hit ratio.

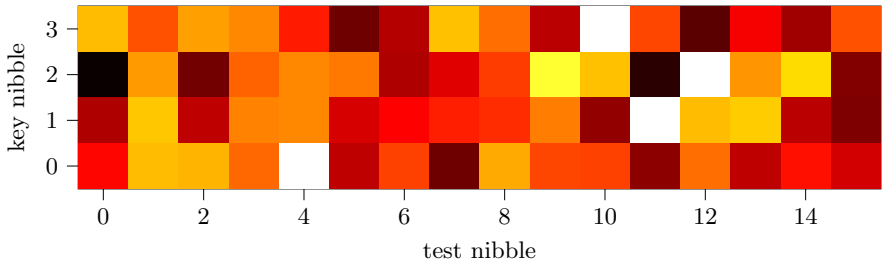


Fig. 8. Heatmap of conflicts in an AES Disaligned T-table attack on key bytes 0, 4, 8, 12 with a total of 1500 encryptions. Correct key nibbles are 4, 11, 12, 10.

We can recover all upper nibbles for these 4 key bytes in 1500 encryptions with 100 % accuracy. Figure 8 shows a heatmap of one such attack with a total of 1000 encryptions, clearly displaying the key correct key nibbles 0x4, 0xb, 0xc and 0xa.

7.2 First-Round Correlation Attack

For the standard T-table attack, we use the fact that there is a 92 % chance that any given cache line in a table will be accessed with a random plaintext. We have established above that the spacial resolution of Cohere+Reload is not enough to us this in a standard first-round cache attack. However, we know each key- and plaintext byte combination accesses a specific cache line with 100 % certainty. This in turn means either the first or second coherence partition will be accessed for each plaintext byte in the first round. Looking at a single key byte at a time, we can easily calculate a pattern of partition 1 and partition 2 accesses for each plaintext and each key. Concretely, we can create a template vector for each possibly key byte value of the form $\{0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0\}$. Each number denotes the partition that $P_n \oplus K_n$ will access, in this case for $K_n = 0$ and a pattern size of 256 B. We can now pick a fixed plaintext and change only one plaintext byte at a time and record the number of total accessed cache lines in the T-tables that fall within the one of the partitions. With this, we can now calculate the Pearson correlation coefficients between the templates and the access count vector. The highest correlation will show the template corresponding with the correct upper nibble of the tested key byte.

When we generate the template vectors, we find that they contain a different number of unique templates for the 256 B and 512 B coherence patterns. Depending on the cache line offset within a page, there are at most 8 unique templates for a 256 B pattern and 16 for the 512 B pattern. This means for all odd cache line offsets of the T-tables, we can recover 3 or 4 bits per key byte, depending on the chosen pattern size. The pattern size 256 B yields one bit less, since the disaligned coherence pattern within each table repeats once (see Figure 7) and we can therefore not distinguish the most significant bit.

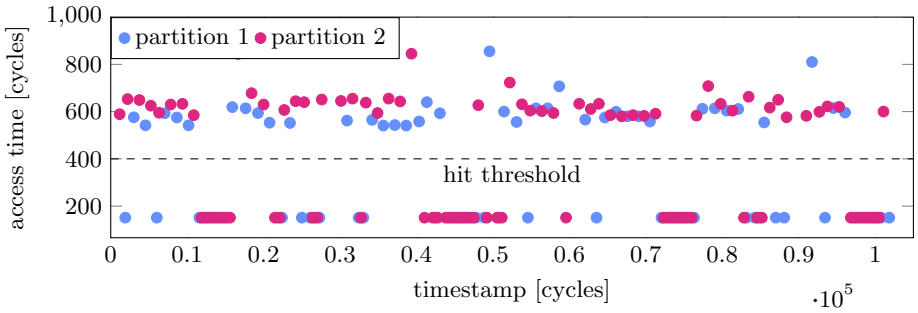


Fig. 9. Cohere+Reload access trace for a single AES encryption.

While this attack can benefit from chosen plaintexts (as described above), a sufficient number of (mostly) random known plaintexts will work just the same. By adding the number of partition accesses to a bucket for each plaintext byte value and for every byte of a random plaintext, each encryption can contribute to the recovery of all 16 key bytes instead of just one. With many encryptions, this creates a 16x16 matrix with one correlation vector for each key byte. After enough encryptions, the bias in the average number of accessed cache lines in a coherence partition outweighs the initial noisiness of random plaintexts and they key bytes can be recovered. Therefore we consider this a known-plaintext attack.

Cohere+Reload provides for two methods of measuring the number of accesses. Firstly, the access time for a single read, as described in Section 4.2. In theory, over enough measurements with randomized plaintexts, the average access time should provide a proxy measure for the number of cache lines that were accessed within a partion. Unfortunately, we could not make this method work with AES. Fortunately, we can make use of the minimal blind spot and high frequency of Cohere+Reload and mount a trace attack.

Unlike in the case of RSA, there is very little time between accesses to the T-tables in the OpenSSL implementation. When we try to trace an encryption normally, we only observe 10-15 accesses per partition, for a total of $\approx 20 - 30$ accesses out of the $\approx$ 135 expected accesses (less than 160, as some accesses fall on the second page). Since in our threat model (cf. Section 3) we are the hypervisor, we can however employ a little trick to slow down our victim. Even in SEV-SNP, the hypervisor has control over the bits in the page table entries, including the uncacheable bit. We find that by making both the function code and the stash page uncacheable, we can slow down our victim considerably. This allows us to record around 100 total memory accesses for a single encryption, as we can see in Figure 9. Though still shy of 160, we can not reliably see all 16 individual accesses in the first round and infer the table accesses directly. We can, however use the number of hits to the partitions as a proxy. Even though some hits will contain two or more accesses, on average the number of accesses in a partition will be biased by the first round. To reduce the noise, we only look

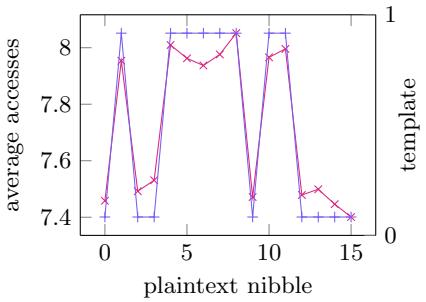


Fig. 10. Correlation attack template vector for key nibble 0xa vs. average access counts for correct key nibble guess with 8000 encryptions. $\rho = 0.99$.

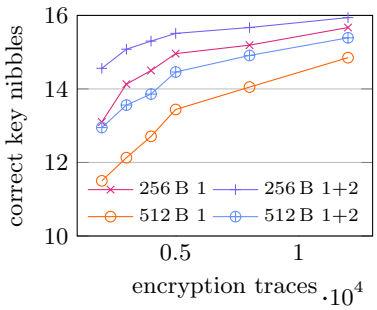


Fig. 11. Average number of correct key nibbles for a given number of observed AES traces for either the top guess (1) or the top two guesses combined (1+2). $n = 100$ per point.

at the first 16 accesses to both partitions, and extract the number of accesses to one of them as our signal. We choose 16 as it is the upper limit to how many hits we can see within the first round, and even if accesses from the second round are included, they only add noise. We can see this signal plotted together with the correct template in Figure 10. In this case, we achieve a very high correlation coefficient of $\rho = 0.99$.

Figure 11 shows our results for both pattern sizes. We can see that recovering 3 bits is more robust, as the templates are more different to each other. With 4000 traces, we recover an average of 14.5 times 3 bit in our first guess. Adding second guesses, this rises to 15.3. Using the 512 B pattern, we can recover an average of 14.85 nibbles per byte with 12000 encryptions with first guesses and 15.4 when we also include second guesses.

Since this is a correlation based attack, we can identify weakly correlating key nibbles, or those where several candidates are close, and record more traces only in these cases to minimize overall traces.

Tracing AES with this time resolution (without repeating encryptions) is something we do not believe can be easily achieved *across cores or even sockets* with other cache attacks like Flush+Reload or Flush+Flush, as Cohere+Reload can monitor an entire page with only two addresses (cf. Section 4.3). Though compared to the standard our attack takes longer (e.g., Flush+Reload can work with only a few hundred to low thousands of encryptions as shown in Section 7.1), it also has slight advantages. Firstly, as a trace-based attack it is not hindered by software prefetching, as each read resets the cache line. Secondly, this attack functions the same for 128 bit keys as it does for 192 bit or 256 bit keys, as the additional two or four rounds do not affect the beginning of the attack, whereas for other attacks the probabilities become less favorable.

8 Load-time based attacks

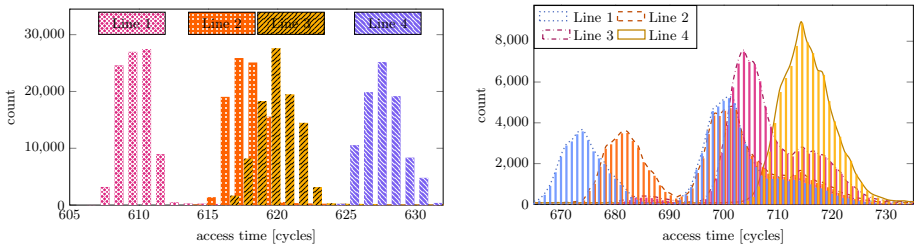
In Section 4.2, we observed that Cohere+Reload-timings to different cache lines in the same coherence block have discernible timing differences, *i.e.*, an attacker can conclude which cache line was accessed by measuring the time taken to access a cache block. In this section, we use this observation to mount two synthetic attacks: the first reveals which part of an array is accessed while the second detects which case of a `switch` statement is taken. We test these attacks on an AMD EPYC 8024P (Zen 4c) with hardware-prefetching disabled.

Our experimental setup consists of two processes, an attacker and a victim, which can access the same physical page as a ciphertext or plaintext mapping respectively. This page is either a dynamically allocated array storing values (the first attack), or it the victim's code (the second attack). We assume that the region of interest is one cache-block large (256 B) and assume that the physical address is aligned to this value. For the attack on code execution, we ensure that each case of the switch is one cache-line long and all four cases are within the same cache block. By measuring the access time with Cohere+Reload, we can now infer which line in a coherence block was accessed by the victim, which in turn can let us infer control- or data flow. With the attack on code, a source of noise is the (speculative) fetching of instructions from the next case by the instruction prefetcher. To overcome this, we use the `ret` instruction at the end of every case to indicate that the next set of instructions will not be executed.

For the purposes of the experiment, the attacker and victim alternately access the physical address. The attacker records the access times and the victim accesses only one random cache line. Each cache line is accessed 100 000 times, resulting in 400 000 measurements. Figure 12a shows the access time distribution to the coherence block when the monitored address corresponds to a dynamically allocated array, *i.e.*, *data*. In Figure 12b, we see the access times to the coherence block when the physical address corresponds to a switch and each case is on a different cache line, *i.e.*, *code*.

With these measurements we can make several observations. First, access times to code (Figure 12b) is noisier than data (Figure 12a), even in our example with a `ret` at the end of every case. In further tests we learn that the distributions become visually separable when the attacker can choose to repeat the same input. Since the `ret` instruction also improves the result but does not make it perfect, we believe this is a combination of misspeculation and a race condition between how fast the front end can fetch data vs. how soon the `ret` is decoded.

In a real attack, exploitabilty depends heavily on the control an attacker has over the target branch. If a single secret bit can be reliably repeated, it only takes a few samples to train predictors and receive a clean signal (cf. Section 4.2). If a sequence of bits can reliably be repeated in the same order (e.g., a key that is used bit by bit), Figure 12b shows that by combining repeated measurements we can produce distinct distributions, even when attacking instructions.



a. Data **b.** Code

Fig. 12. Cohere+Reload access times to a coherence block when it contains victim data (Figure 12a) and victim code (Figure 12b).

9 Mitigation

AMD disabling the coherence feature will undoubtedly mitigate Cohere+Reload, though `VMPAGE_FLUSH`, if present, may lead to similar leakage. The SEV-SNP attestation flag `CiphertextHidingDRAM` disallows reading of the ciphertext by the hypervisor when active. As we have no hardware that supports this feature, we can only speculate that the coherence mechanism could be disabled when this setting is enabled depending on how it is implemented .

For hardened implementations, the memory type of critical sections (e.g., T-tables) could be set to *uncacheable*. While this slows down execution, our experiments show that *loads* do not trigger the coherence mechanism.

10 Conclusion

In this paper, we introduced Cohere+Reload, a novel side-channel attack exploiting AMD's coherency for encrypted memory. We exploit two types of leakage in the coherency mechanism: First, we exploit coherence conflicts, leaking victim operations on a spatial granularity of a 2 kB block. Second, we exploit timing correlations with the number and location of accesses, reaching a maximum spatial resolution of 256 bytes. In our synthetic attacks, we showed that Cohere+Reload can observe control flow and access locations in workloads within a confidential virtual machine. As a benchmark we mounted an attack on mbedTLS RSA, leaking 99.7 % of the 4096 key bits in a single-trace attack. We also mounted an attack on OpenSSL AES exploiting disalignments on a cache line granularity, achieving an accuracy of 100 % in only 1500 encryptions in a first round T-table attack and an accuracy of 92.81 % in 12000 encryptions with a novel correlation attack. Our work shows that coherence mechanisms can undermine the confidentiality of confidential virtual machines. Consequently, AMD should to re-evaluate their approach to the coherency challenge based on the mitigation directions that we discussed.

Acknowledgments

We want to especially thank Andreas Kogler for the initial idea and many productive conversations. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). 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. Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel Trust Domain Extensions (TDX) Security Review, 2023. URL: `https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf`.
2. AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, 2020. URL: `https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf`.
3. AMD. AMD Secure Encrypted Virtualization (SEV), 2024. URL: `https://developer.amd.com/sev/`.
4. AMD. AMD64 Architecture Programmer's Manual, 2024.
5. ARM. Arm Confidential Compute Architecture, 2024. URL: `https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture`.
6. Daniel J. Bernstein. Cache-Timing Attacks on AES, 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.
7. Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *CHES*, 2006.
8. Elad Carmon, Jean-Pierre Seifert, and Avishai Wool. Photonic Side Channel Attacks Against RSA. In *HOST*, 2017.
9. Confidential Computing Consortium. A Technical Analysis of Confidential Computing, 2022.
10. Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security*, 2017.
11. Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. *arXiv:1712.05090*, 2017.
12. Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In *S&P*, 2023.
13. Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In *NDSS*, 2025.
14. Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In *ESORICS*, 2015.
15. Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In *CCS*, 2019.

16. Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.
17. Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*, 2015.
18. Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. *ACM SIGPLAN Notices*, 52(7):129–142, 2017.
19. Gal Horowitz, Eyal Ronen, and Yuval Yarom. Spec-o-Scope: Cache Probing at Cache Speed. In *CCS*, 2024.
20. Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*, 2013.
21. Intel. Intel Trust Domain Extensions, 2021. URL: `https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf`.
22. Intel. Intel Software Guard Extensions (Intel SGX), 2024. URL: `https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html`.
23. Intel. Intel Trust Domain Extensions Module Base Architecture Specification, 2024. URL: `https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html`.
24. Intel. Intel Total Memory Encryption White Paper, 2025. URL: `https://www.intel.com/content/www/us/en/architecture-and-technology/vpro/hardware-shield/total-memory-encrpytion.html`.
25. Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *RAID*, 2014.
26. David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption, 2016.
27. Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.
28. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, 1999.
29. Tom Lendacky. What processors support SEV? #1, 2019. URL: `https://github.com/AMDESE/AMDSEV/issues/1#issuecomment-581426096`.
30. Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*, 2021.
31. John Monaco. SoK: Keylogging Side Channels. In *S&P*, 2018.
32. Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting AMD's virtual machine encryption. In *EuroSec*, 2018.
33. Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapotre, Muhammad Khurram Bhatti, and Guy Gogniat. Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA. *Information Systems*, 92:101524, 2020.
34. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.
35. Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *CCS*, 2021.
36. Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In *E-smart*, 2001.
37. Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A Systematic Evaluation of Novel and Existing Cache Side Channels. In *NDSS*, 2025.
38. Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*, 2009.

39. Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.
40. Raphael Spreitzer and Thomas Plos. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *COSADE*, 2013.
41. Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. MeshUp: Stateless cache side-channel attack on CPU mesh. In *S&P*, 2022.
42. Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In *AsiaCCS*, 2019.
43. Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity–Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *S&P*, 2020.
44. Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. *Cryptology ePrint Archive, Report 2014/140*, 2014.
45. Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.
46. Mark Zhao and G Edward Suh. FPGA-based Remote Power Side-Channel Attacks. In *S&P*, 2018.