A Close Look at RMP Entry Caching and Its Security Implications in SEV-SNP

Alexis Bagia
Technische Universität Berlin
Berlin, Germany
a.bagia@tu-berlin.de

Vincent Quentin Ulitzsch
Technische Universität Berlin
Berlin, Germany
Massachusetts Institute of Technology (MIT)
Cambridge, MA, USA
viniul@mit.edu

Daniël Trujillo
Massachusetts Institute of Technology (MIT)
Cambridge, MA, USA
danieltr@mit.edu

Mengyuan Li
University of Southern California
Los Angeles, USA
mli49061@usc.edu

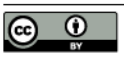
Mengjia Yan
Massachusetts Institute of Technology (MIT)
Cambridge, MA, USA
mengjiay@mit.edu

Jean-Pierre Seifert
Technische Universität Berlin
Berlin, Germany
Fraunhofer Institute for Secure Information Technology
Darmstadt, Germany
jean-pierre.seifert@tu-berlin.de

Abstract

AMD's Secure Encrypted Virtualization (SEV) technology is a pivotal component in AMD server processors that boosts cloud computing security. It achieves this by offering transparent memory encryption and managing keys for protecting virtual machines (VMs), independently of the hypervisor's trustworthiness. The latest iteration, SEV-Secure Nested Paging (SEV-SNP), introduces memory integrity protection through a data structure called the Reverse Map Table (RMP), which maps system physical addresses to guest physical addresses and tracks ownership of physical pages.

The RMP is maintained in a dedicated region in DRAM. As every memory write triggers a check against an RMP entry, caching RMP entries is crucial to alleviating the RMP's performance impact. However, caching may create new security challenges, as it can introduce new microarchitectural side-channels. In addition, maintaining cache coherence is crucial for the RMP's security guarantees. However, so far, neither the details of the RMP's caching behavior nor its security implications have been explored. This paper aims to fill this gap by conducting a systematic study of the RMP's caching behavior. Through reverse engineering, we identify that the RMP is not only cached in the TLB, but also in the L1D and L2 data cache. Interestingly, this caching depends on the access type on Zen 5. We also uncover the mechanisms by which cache coherence across the TLB is enforced. We find that each update to the RMP table triggers a global TLB flush across all cores. Finally, we present several potential security implications and demonstrate that an attacker can exploit RMP's caching to leak physical address information. A user process can leak 6 bits of the Physical Frame Number (PFN) of its pages via the L1D cache within 2.5 µs per page, with success rates of 97 % (Zen 4) and 99 % (Zen 3 and Zen 5).



HASP 2025, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2198-4/25/10
https://doi.org/10.1145/3768725.3768727

CCS Concepts

• **Security and privacy** → **Trusted computing**; *Virtualization and security*.

Keywords

confidential computing, trusted execution environment, virtual machine, microarchitecture, side channel, AMD SEV

ACM Reference Format:
Alexis Bagia, Vincent Quentin Ulitzsch, Daniël Trujillo, Mengyuan Li, Mengjia Yan, and Jean-Pierre Seifert. 2025. A Close Look at RMP Entry Caching and Its Security Implications in SEV-SNP. In *Hardware and Architectural Support for Security and Privacy 2025 (HASP 2025), October 19, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3768725.3768727

1 Introduction

The last decade saw a mass adoption of cloud computing, providing easy access to scalability and resilience. However, cloudification requires entrusting the cloud service provider with handling the cloud services, as the hypervisor is able to inspect and alter guest memory without the permission or knowledge of the end-user. This strong requirement inhibits cloud computing for sensitive data processing, as cloud service providers could potentially steal data processed in the cloud.

Trusted Execution Environments (TEEs) and Confidential Computing provide solutions to this problem. These technologies allow tenants to run workloads on untrusted hosts, promising confidentiality and integrity. One particular successful realization of Confidential Computing is AMD's Secure Encrypted Virtualization (SEV) technology, which allows users to run a virtual machine (VM) in a confidential manner, without needing to adapt the workloads to specifically support confidential computing. SEV realizes this technology by protecting a virtual machine's data in-use through memory encryption [15]. When the confidential virtual machine is active, its data are encrypted and protected from unauthorized access, ensuring that even the hypervisor cannot view or tamper with

the data. This robust security model provides strong assurances to tenants that their sensitive information remains confidential, regardless of the underlying host's security posture. Since its inception in 2016, AMD SEV has seen multiple improvements, mitigating attacks that exploit design or implementation flaws, or easing usability [1, 2, 7, 8, 13, 18–21, 29, 30]. The latest development, AMD Secure Nested Paging (SNP), represents a significant advancement by introducing robust protection mechanisms that effectively mitigate the majority of the known attacks. This iteration introduces a core new feature that is particularly intriguing from a security perspective, warranting detailed analysis.

AMD-SNP adds integrity protection to the VM by introducing the Reverse Mapping Table (RMP), mitigating attacks that exploit the hypervisor's ability to silently modify the VM's data [2, 18, 29]. The RMP is a data structure that maintains ownership information for system physical memory at page granularity. The Memory Management Unit (MMU) consults the RMP on each memory write, enabling it to block unauthorized modifications by a malicious hypervisor to a virtual machine's memory.

Implementing the RMP securely and efficiently is challenging. To avoid performance bottlenecks from frequent DRAM accesses, RMP entries must be cached. However, this introduces potential security risks, as caching can give rise to microarchitectural side-channels. Ensuring coherency and preventing race conditions further complicates the design. Despite its importance, RMP's caching behavior and its security implications remain unexplored in prior work — an oversight this paper aims to address.

Contributions. This paper conducts a systematic, security focused, analysis of the caching of RMP entries. Our analysis uncovers microarchitectural implementation details, new attack vectors, and illuminates the threat landscape of cached RMP entries. Our results apply across all microarchitectures that support SEV-SNP, including Zen 3, 4, and 5. In summary, our contributions and findings are as follows.

- RMP entries are cached not only in the TLB, but also in the L1D and L2.
- Updates to a single RMP entry require multiple memory operations, enabling other cores to observe and load RMP entries while being updated.
- A `lock` bit and a global TLB flush across all cores ensure cache coherence and semantic atomicity when RMP entries are updated.
- We show a user-space attacker can exploit the caching of RMP entries in the L1D and L2 cache to leak physical address bits of its virtual address space. In the process, we also reverse engineer AMD's custom L2 hash functions.

2 Background

2.1 AMD SEV

AMD's Secure Encrypted Virtualization (SEV) is a hardware-based TEE for confidential computing, protecting a virtual machine's data-in-use. SEV and its variants are supported by AMD's server-level EPYC processors. To protect virtual machines from an untrusted or malicious hypervisor, hardware-based memory encryption ensures confidentiality, while the Reverse Map Table (RMP) enforces integrity of guest memory. SEV therefore acts as a Trusted Execution Environment (TEE) protecting a full VM. This is in contrast to Intel SGX, which protects dedicated applications instead.

The SEV technology provides two features to this end. First, a VM's user can verify the VM's correct deployment through the *remote attestation* procedure. Second, the *runtime protection* of a VM, achieved through its memory encryption.

Through remote attestation, a user can request an attestation report for their VM, which contains so-called *measurement* data. This measurement contains, for example, the hash of the VM's code and provides a proof of the execution platform's authenticity. This report is generated by a dedicated security subsystem, the AMD Platform Security Processor (AMD-PSP), an ARM based system on chip. The transparent memory encryption utilizes an AES memory encryption engine, embedded into the MMU.

2.2 Caches in x86

Modern x86 processors incorporate multiple types of caches, but the principal cache hierarchy is formed by the three-level data- and instruction-caches that serve memory access latency reduction. The L1 cache is split into instruction (L1I) and data (L1D) caches, while the L2 and L3 cache is unified. The L1 and L2 are private to a physical core and shared between SMT threads, while the L3 is shared across cores.

2.2.1 TLB. The Translation Lookaside Buffer (TLB) is a specialized cache for virtual-to-physical address translations, designed to reduce the overhead of frequent page walks. It is typically organized into two levels: a small, fast L1 TLB and a larger, slower L2 TLB. On AMD systems, the TLB hierarchy is fully split by access type [26]: the L1 and L2 DTLBs cache data page translations, while the L1 and L2 ITLBs cache instruction page translations.

2.2.2 Cache coherency. In modern x86 multicore processors, cache coherence across data and instruction caches is automatically maintained. AMD uses the MOESDIF protocol [5], an enhanced MOESI variant that supports efficient multi-core data sharing. This protocol ensures that cache lines residing in private caches of different cores remain synchronized, maintaining a coherent view of shared data and instruction streams across the system.

However, the Translation Lookaside Buffer (TLB) lacks hardware-managed coherence; coherence following page table modifications is enforced through software-managed TLB shootdowns, typically initiated by the hypervisor or operating system using privileged instructions such as `INVLPG`.

3 The Reverse Map Table

This paper provides an in-depth analysis of the Reverse Map Table (RMP), a data structure recently introduced with AMD-SNP technologies. The RMP's purpose is to safeguard the integrity of virtual machine memory. To this end, the RMP assigns each physical page an owner, e.g., the hypervisor or a specific VM. The CPU enforces that only the owner of a physical page can write to that page.

RMP data structure. The RMP is stored in a dedicated, contiguous region of physical memory, with entries laid out sequentially as a single table. The kernel can read out the RMP's physical address through dedicated Model Specific Registers (MSRs). The RMP itself consists of a sequence 16-byte RMP entries, each storing the state

of one 4KB physical page. Each RMP entry is comprised of multiple fields, which define a page's state. For instance, the *assigned* field determines if a page is owned by the hypervisor or not. Further, RMP entries contain two reserved, undocumented fields. The AMD-PSP firmware source code labels one of these undocumented fields as `lock` [4]. We reverse engineered the semantics of this lock bit and present the results in Section 6.

RMP checking. To enforce integrity protection, the hardware performs the following upon each write access [2].

- Translates the virtual address to the system physical address (SPA).
- Uses the SPA as an index for the RMP.
- Checks if the writing entity is allowed to perform the write. For example, the hypervisor is not allowed to write to guest-owned pages.

RMP state transitions. The RMP table can only be updated via the dedicated x86 instructions `RMPUPDATE` and `PVALIDATE` or via firmware commands to the PSP. This ensures that only valid state transitions take place. For example, assigning a page owned by the hypervisor to the PSP can only be done through the `RMPUPDATE` instruction, and will automatically place the page in the firmware state (by setting the `assigned` and `immutable` bit). The hypervisor can later reclaim the page by first sending the `SNP_PAGE_RECLAIM` command to the PSP, which transitions the page to the reclaim state by clearing the `immutable` bit. Using `RMPUPDATE` the hypervisor can then transition the page to the hypervisor state by clearing the `assigned` bit.

4 Key Research Questions

SEV-SNP introduces the Reverse Map Table (RMP) to enforce memory integrity by tracking ownership of physical pages. However, its interaction with the CPU's caching hierarchy introduces subtle microarchitectural complexities with potential security implications. Our work is driven by three central research questions.

Question 1: Where and when are RMP entries cached? Performance considerations require caching RMP entries, yet the architectural documentation provides only vague references, such as that RMP entry are cached in "the CPU TLB and related structures" [2, p. 9].

We reverse engineer RMP's caching behavior. We confirm that RMP entries are cached in the TLB, and additionally discover that they are cached in the L1D and L2 cache as well. Furthermore, we find that these entries are always loaded during a page walk on Zen 3 and Zen 4, whereas on Zen 5 they are loaded only when a check is required (Section 5).

Question 2: How does AMD maintain cache coherence of RMP entries in the TLB and prevent race conditions during their updates? Unlike data caches, the TLB is not hardware-coherent across cores. Software-managed TLB invalidation is insufficient, since the hypervisor is untrusted in the SEV-SNP threat model. To prevent stale permissions and break isolation guarantees, updates to RMP entries need to propagate consistently. We show that AMD mitigates this at the microarchitectural level by (1) issuing a global TLB flush on all cores during each RMP update and (2) making use of the RMP entry's `lock` field to avoid race conditions in the process.

Question 3: What security implications arise from our results? Our reverse engineering reveals a number of potential security implications stemming from the microarchitectural realization of the RMP. As a case study, we demonstrate how RMP's caching can be exploited to leak physical address bits by an unprivileged user-space program.

We present an attack that exploits that the cache set in which an RMP entry is placed depends on the physical address being accessed. Our proof-of-concept leaks physical address bits within 2.5 µs for a particular page on Zen 5, with a 99 % success rate (Section 7).

5 Reverse Engineering the RMP's Caching Behavior

This section investigates when RMP entries are loaded and in which caches they are stored. According to official documentation, RMP entries are loaded after the virtual to physical address translation of a page walk and results of the RMP check are cached in "the CPU TLB and related structures" [2, p. 9]. When operating in hypervisor mode, every memory write triggers a check of the RMP, while a read does not [5, p. 609]. However, the documentation does not clarify (1) whether RMP entries are cached in the data cache hierarchy (2) whether RMP entries are *always* loaded after a page walk or only when an RMP check is required, and (3) which specific RMP entry is checked for a huge page.

To answer these questions, we conduct experiments that monitor cache traces of the page walker in the L1 and L2 data cache [1] (Section 5.1). To properly associate the L2 cache traces with memory locations, we also reverse engineer AMD's custom L2 hash function (Section 5.2).

Experimental Setup. We conduct the experiments on the AMD EPYC 7431 (Zen 3), 9184X (Zen 4), and 9115 (Zen 5) processors, each equipped with 16GB of RAM. We run the Ubuntu operating system with a Linux kernel compiled directly from AMD SEV's GitHub repository[2].

5.1 RMP Caching

Since RMP entries are accessed during the page walk, observing the page walker's behavior is key to determining whether these accesses are cached. To capture its cache activity, we combine TLB flushing with the Prime+Probe technique from cache side-channel attacks. However, because we can only observe the page walker's cache accesses as a whole — rather than isolating individual memory accesses — we must correlate specific cache sets with known memory locations to determine which accesses are being cached.

Specifically, our technique operates as follows. Given a target virtual address, we begin by explicitly evicting the TLB entry associated with the target address to force a page walk on subsequent accesses. Next, we prime a cache set using an eviction set—a set of congruent addresses that map to the same cache set. We then access the target address. Since its TLB entry was evicted, the CPU performs a page walk followed by a memory access to the target. Subsequently, we re-access our eviction set and measure a performance counter that counts cache misses. If no cache misses are

[1] we exclude the LLC in our analysis due to its complexity
[2] `6.11.0-rc3-snp-host-85ef1ac03941`

counted, it indicates that the eviction set remained in the cache, implying that no conflicting accesses occurred during the page walk. Conversely, one or more cache misses suggest that one or more cache lines from the eviction set were evicted, indicating that the page walker cached a memory access to an address mapping to the same cache set.

We conduct this experiment on both the L1 and L2 data caches, evaluating three types of memory operations: read, write, and prefetch. For read and prefetch operations, we perform the experiment on both read-only and read-write pages. Our results vary across microarchitectures. On Zen 3 and Zen 4, RMP entries are always loaded during a page walk—regardless of page permissions or the triggering instruction—meaning the RMP entry is loaded even if no RMP check is ultimately performed. By contrast, on Zen 5, RMP entries are loaded only during page walks initiated by write accesses, i.e., when an RMP check is actually needed.

If an RMP entry is loaded, we observe six cache misses in the L1 and eight in the L2. In both cases, the evicted cache sets match with the cache sets of the page table entries and the RMP entry of the accessed memory location. The two additional L2 cache evictions match with the cache sets of the experiment code itself. Figure 1 shows a measured L1 and L2 cache trace with RMP access. Using color annotations, we map each relevant cache set to its associated memory location.

The RMP also supports tracking the ownership of 2MB huge pages, for which ownership can be changed through one `RMPUPDATE` instruction. Changing the ownership of a huge page will cause updates to all 512 4KB encompassing entries (cf. Section 6.1). Naturally, the question arises how these entries are consulted and cached.

We repeat the prime and probe experiment for 2MB pages. The experiments reveal that upon a 2MB page memory access, only the first of the 512 RMP entries associated with the 2MB address range is consulted. We find that, in line with AMD's documentation [6, p. 447], only this first RMP entry contains page state information. The remaining 511 entries merely have the `assigned` flag set. We refer to this first entry as the `authoritative` RMP entry.

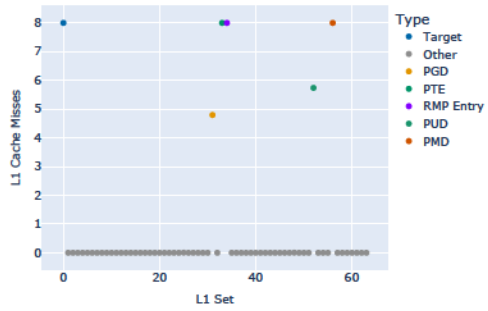
RMP entries only support page sizes of 4KB and 2MB. AMD states that "accesses to 1GB pages only install 2MB TLB entries when SEV-SNP is enabled" [5, 608]. Our experiments confirm that accessing 1GB pages at various offsets results in an access to the respective 2MB page RMP entry.

Result 1: RMP entries are loaded during the page walk when an RMP check is required (Zen 5) or always (Zen 3 and 4), and are subsequently cached in the TLB, L1D, and L2 caches.

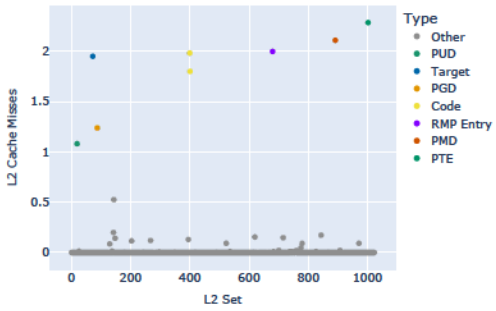
5.2 L2 Hash Function

We now give more details about the L2 hash function, which was required to reverse engineer the RMP's caching behavior. To reverse engineer the L2 hash function, we assume it follows a linear form, which is typical for microarchitectural hash functions [14, 22]. Formally, we model the hash function $H(p) = s = s_m || \dots || s_0$ as having the following linear form:

$$H_i(p) = s_i = \sum_{j=0}^{n} h_{i,j} p_j \mod 2$$



(a) L1 sets accessed during a page walk.



(b) L2 sets accessed during a page walk.

Figure 1: Measurements showing L1 and L2 cache misses during our probe step, using the performance counters for L2 hits and L2 misses, respectively. Each set is measured separately, with values representing the average over 2,000 iterations. The data indicate the accesses conducted during the page walk of the target, where variance in cache misses can be explained by the cache's eviction policy.

, where

- p_j is bit j of the physical address p
- $s_i \in \{0, 1\}$ is bit i of the L2 cache set p is mapped to
- $h_{i,j} \in \{0, 1\}$ describes the hash function
- $i \in 0, \dots, m$, with $m = 9$ or 10 depending on the number of L2 sets
- $j \in \{0, \dots, n-1\}$ with $n = 34$ as our test machine has 16GB of main memory

Reverse engineering the L2's cache set hash function is a matter of reverse engineering the values for each $h_{i,j}$. To this end, we utilize the following key idea [22]. Consider two physical addresses p and p' that differ in exactly one bit j, and that p is mapped to the L2 set s and p' is mapped to L2 set s'. The physical address bit at position j only has an influence on the i'th bit of the resulting cache set if $s_i \neq s'_i$. Otherwise, we can conclude that the j'th position of the physical address bit at position i does not influence the i'th cache set bit.

5.2.1 Results. On our test system with 16 GB of main memory, we reverse engineered the following hash functions. Concurrent and independent work by Wang et al. [27] confirms our results for Zen

3, reporting an equivalent hash function. However, our labeling yields a more compact description.

$$H_i^{\text{Zen3}}(p) = s_i = \begin{cases} p_{i+6} & \text{if } 0 \le i < 3 \\ p_{i+6} \oplus p_{i+18} & \text{if } 3 \le i < 10 \end{cases}$$

$$H_i^{\text{Zen4}}(p) = s_i = \begin{cases} p_{i+6} & \text{if } 0 \le i < 3 \\ p_{i+6} \oplus p_{31-i} \oplus p_{i+26} & \text{if } 3 \le i < 8 \\ p_{i+6} \oplus p_{31-i} & \text{if } 8 \le i < 11 \end{cases}$$

$$H_i^{\text{Zen5}}(p) = s_i = \begin{cases} p_{i+6} & \text{if } 0 \le i < 4 \\ p_{i+6} \oplus p_{30-i} \oplus p_{i+23} & \text{if } 4 \le i < 10 \end{cases}$$

Result 2: AMD uses a complex xor hash function for the L2 cache.

6 Cache Coherence of RMP Entries

Two challenges present themselves as a result of RMP entry caching. First, the issue of *cache coherence* for both data caches and the TLB. Although AMD automatically ensures cache coherence of the data caches using the MOESDIF protocol [5, p. 192], cache coherence of the TLB must be manually ensured by the hypervisor. As the hypervisor is not trusted in the SEV-SNP threat model, this approach is not suitable for RMP entries cached in the TLB. Consequently, the CPU hardware now needs to be responsible for ensuring the coherence of RMP entries cached within the TLB, as described in an AMD white paper: "When required, hardware automatically performs TLB invalidations to ensure that all cores in the system see the updated RMP entry information" [2, p. 9].

Second, updating RMP entries atomically is inherently challenging, as atomic memory operations on x86 are only guaranteed up to a single quadword [5, p. 256], while each RMP entry spans two quadwords. While we demonstrate that partially updated RMP entries are indeed observable from other cores, we also show how AMD mitigates potential race conditions arising from the non-atomic nature of these updates.

6.1 RMP Entry State Transitions

To gain a deeper understanding on how the RMPUPDATE instructions works, we conduct two experiments that closely monitor RMP entries while being updated for 4KB and 2MB pages respectively.

To observe the 4KB page RMPUPDATE process, we set up two threads. One thread executes an RMPUPDATE instruction and another thread, scheduled on a different core, polls the corresponding RMP entry. To not miss any fast occurring updates to the RMP entry, we slow down the core that updates the RMP entry and speed up the core that polls the RMP entry by setting the frequencies of the respective cores appropriately.

We conduct this experiment for all four state transitions away from the Hypervisor state, monitoring the upper quadword of the RMP entry. For each state transition, we observe changes in following chronological order: (1) the lock bit is set (2) fields of the RMP entry are updated according to the triggered state transition (3) the lock bit is cleared if transitioning to the Reclaim or Guest-Invalid state.

Table 1: Timeline of changes on the RMP table when a RMPUPDATE is performed on a 2MB page with page frame number pfn. The change of the value of V_{AB} indicates the TLB flush. For 4KB pages, we observe the same timeline without the assigned bits being set on other RMP entries.

Time		0	21942	24937	26845
rmp[pfn].lo	.lock	1	1	1	0
	.asid	0	0	0	1
	.assigned	0	0	0	1
rmp[pfn + 511].lo	.assigned	0	1	1	1
V_{AB}		'A'	'A'	'B'	'B'

To observe the 2MB page RMPUPDATE process, we extend the previous experiment by having the second thread also monitor a non-authoritative RMP entry, e.g., the last of the 512 RMP entries belonging to the 2MB page. Our experiment reveals that for 2MB pages, the update process resembles that of a 4KB page; however, between steps (1) and (2), the assigned bit is set on all non-authoritative RMP entries. The precise measurements of our experiment are shown in Table 1.

Result 3: Updating a single RMP entry involves multiple memory operations, including setting and potentially clearing a lock bit.

6.2 Cache Coherence via Global TLB Flush

To understand how cache coherence of RMP entries cached in the TLB is implemented, we perform an experiment similar to the previous ones performed in Section 6.1, with one addition. To precisely monitor at what point in time the TLB is invalidated on a core different from the core performing the RMPUPDATE, we make use of the TLB desynchronization technique introduced in [26].

The monitoring thread now further polls a special virtual address V_{AB}. This virtual address is cached in the TLB but is out of sync with the page table, that is, according to the TLB, V_{AB} translates to a physical page A, but according to the processes page table, V_{AB} translates to a different physical page B. By polling this virtual address, we can precisely monitor at what point in time the TLB entry of V_{AB} is invalidated and reloaded. While we poll V_{AB}, we initially read the value of page A, because the out-of-sync TLB entry is still in the TLB, once the TLB entry is invalidated and reloaded from the page table, we read the value of page B.

We perform this experiment on 4KB and 2MB pages. Our results are as follows. On 4KB pages, we observe that the TLB entry of V_{AB} is reloaded after the lock bit is set and before the fields RMP entry gets updated. For 2MB pages, the TLB entry of V_{AB} is reloaded after the assigned bit of the last of 512 RMP entries is set, and before the fields of the authoritative RMP entry are updated. As the physical pages A and B, and the virtual address pointing to them, is independent of the page on which the RMPUPDATE instruction is performed and because both threads run on different cores, we come to the conclusion that the RMPUPDATE instruction triggers a global TLB flush on all cores.

Result 4: Cache coherence of RMP entries in the TLB is maintained via a global TLB flush on all cores in the system.

6.3 Lock Bit Semantics

Clearly, the lock bit is used to prevent concurrent updates to the same RMP entry from multiple cores. This interpretation is supported by AMD's documentation, which specifies that the RMPUPDATE instruction returns a FAIL_INUSE error if the targeted entry is already being modified by another core [6, p. 448].

We can experimentally validate this behavior: as shown in Section 6.1, certain state transitions leave the lock bit set even after the RMP fields are updated. Attempting a subsequent RMPUPDATE on such an entry yields a FAIL_INUSE error, despite no concurrent update being issued — indicating that a set lock bit triggers this error.

We now show how the MMU interprets an RMP entry while lock bit is set. This detail is critical for the security of SEV-SNP as it may undermine the cache coherence of RMP entries cached in the TLB.

Consider the following scenario: One thread initiates an RMPUPDATE that transitions a page from hypervisor ownership to firmware control. If a second thread accesses the corresponding RMP entry after the lock bit is set and after the global TLB flush, but before the rest of the RMP fields are updated, it may load an transitional state into its TLB—an RMP entry where the lock bit is set, but the remaining documented fields indicate hypervisor writeability.

If the MMU interprets this transitional state as hypervisor-writable, this would violate the isolation guarantees of SEV-SNP: the hypervisor could write to a page that has already been reassigned to a protected domain. Moreover, because this TLB entry was loaded after the global TLB flush, it will not be invalidated when the RMP update completes, making the exposure persistent.

To assess whether such a race condition is security-relevant, we test whether lock-only RMP entries—those where only the undocumented lock bit is set—are treated as hypervisor-writable. We design a two-threaded experiment:

- Thread A performs an RMPUPDATE to convert a 2MB page from hypervisor to firmware ownership, requiring updates to 512 RMP entries.
- Thread B continuously polls the target RMP entry and, upon observing the lock bit set, immediately invalidates the pages TLB entry and attempts to write to the page.

This experiment setup provides a large enough window for observation, as the update process for 2MB pages is slow. Our results show that a write attempt in this state triggers an RMP violation, indicating that the MMU treats lock-only RMP entries as non-hypervisor-writable.

Our experimental evidence suggests that the undocumented lock bit not only synchronizes RMP updates, but also influences MMU access decisions. As a result, transitional RMP states do not pose a security risk, preserving the security guarantees expected by SEV-SNP—even in the presence of microarchitectural race windows.

Result 5: The lock bit serves two purposes: it prevents concurrent RMPUPDATE operations to the same RMP entry and blocks writes to the associated page during the update process.

6.4 Summary

We summarize the behavior of the RMPUPDATE instruction with respect to cache coherence and race conditions in Algorithm 1: The update begins by atomically testing and setting the lock bit, likely using a mechanism similar to the Bit Test and Set (BTS) instruction [6, pp. 126–127]. This bit is respected by both the RMPUPDATE instruction and the memory management unit (MMU), ensuring that race conditions during the subsequent non-atomic modifications to the RMP entry do not compromise security.

Further, in the case of 2MB pages, the assigned bit is set in all non-authoritative RMP entries. A full TLB flush is then broadcast to all cores to ensure coherency of RMP entries cached in the TLBs.

Finally, the fields of the authoritative RMP entry are updated to reflect the new state, and depending on the transition, the lock bit may be cleared.

Algorithm 1 Pseudocode illustrating how the RMPUPDATE instruction uses the lock bit and when it triggers a global TLB flush. Whether the lock bit remains set depends on the resulting state transition.

```
Require: new_rmp_entry, pfn
Ensure: RMP[pfn] updated or error
  if bit_test_and_set(RMP[pfn].lock) == 1 then
    EXIT(FAIL_INUSE)
  end if
  if new_rmp_entry.page_size == 2MB then
    for i = 1 to 511 do
      RMP[pfn + i].assigned ← 1
    end for
  end if
  global_tlb_flush_on_all_cores()
  RMP[pfn] ← conditional_set_lock_bit(new_rmp_entry)
  EXIT(SUCCESS)
```

7 Security Implications of RMP Caching

Our reverse engineering results facilitate future work in a number of potential directions. (A) The fact that RMP entries are cached (dependent on the access type on Zen 5) may have implications for traditional cache side-channels. (B) By triggering RMPUPDATEs (e.g., through a guest message to the AMD-SP), a malicious VM can trigger global TLB flushes, potentially impacting system performance. (C) The cache set in which an RMP entry is placed depends on the physical address being accessed, potentially leaking physical address information, which in turn facilitates a number of microarchitectural attacks, such as Rowhammer and cache attacks [23, 25, 27].

As a case study, this section presents an end-to-end attack leaking 6 bits of the PFN, implementing attack (C). We leave the other directions for future work, as initial results did not show promise (B), or the direction exceeded the scope of this paper (A).

7.1 Physical Address Bit Leakage

This section analyzes the information leakage risks from caching RMP entries in the data caches, demonstrating that a user-level process can leak physical address bits of its pages. Our work can be viewed as a reverse analog to the attack by Gras et al. [12]. They

demonstrate that a user-level process can leak virtual address information due to two main factors: first, the page walker's memory accesses during page walks are cached in the data cache hierarchy; second, these memory accesses depend on the virtual address being translated. Combined, these factors enable virtual address leakage through a cache side channel.

With the introduction of the RMP, a similar scenario emerges — but in the reverse direction. As we demonstrate, RMP entries are cached in the L1D and L2 caches, and crucially, the physical address of each RMP entry is deterministically derived from the physical address of the memory page it protects, which in turn determines its cache set.

In modern operating systems, the physical addresses of memory pages are intentionally hidden from user-space processes [25]. Revealing any bits of the physical address can enable or exacerbate several attacks. For example, such leakage can facilitate more effective Rowhammer attacks, covert and side channels [23, 27] by facilitating the identification of bank conflicts or assist in constructing eviction sets for LLC-based cache attacks [27]. More precisely, the address bits we leak are often involved in DRAM bank selection or LLC set indexing. While the leakage does not necessarily reveal all required bits, exposing a subset can reduce the search space and thereby accelerate such attacks.

7.2 Overview

Threat Model. We assume an unprivileged (user-level) malicious process running on an SEV-SNP enabled machine, such as a monitoring, logging, or health-check agent that runs alongside the hypervisor to support system operations. The attacker aims to obtain physical address bits of its own pages, bypassing Linux kernel protections [25].

Notation. The attack targets the relationship between a "target page" (a page the attacker owns and wishes to learn its physical address bits, denoted with physical address p) and its "target RMP entry" (the RMP entry of the target page with physical address r). We label bits such that r_0 is the least significant bit of a physical address, r_1 the next bit, and so on.

Attack Primitive. The attacker exploits the fact that RMP entries are cached in the CPU's data caches, as we confirmed for the L1D and L2 caches. First, using a Prime+Probe timing side-channel, the attacker determines which cache set is used to cache the RMP entry of the target page. Given this cache set, the attacker gains information about the RMP entry's physical address. In turn, this provides information about the target page's PFN.

7.3 Physical Bit Leakage

Leakage from L1 Cache Set. If an attacker can determine the L1 cache set of the target page's RMP entry, they can directly derive the six cache index bits (denoted $s_0 \dots s_5$) and map them to physical address bits of that RMP entry. In particular, those index bits correspond to bits $r_6 \dots r_{11}$ of the RMP entry's physical address. According to AMD's documentation [3, 5], the RMP table must be aligned to a 1MB boundary and consists of a 16KB bookkeeping region followed by the RMP entries (each 16 bytes in size). Using this knowledge, the attacker can map the bits $r_6 \dots r_{11}$ of the RMP entry

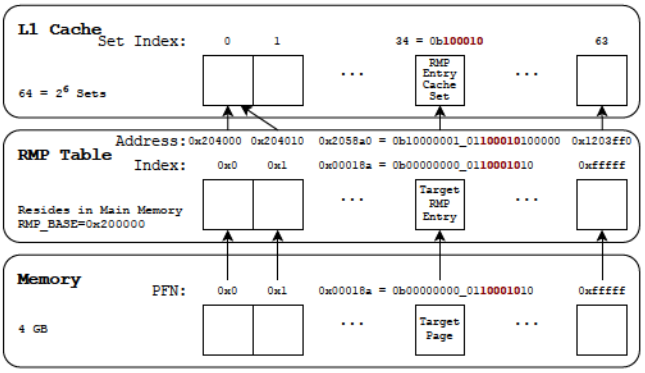


Figure 2: Illustration and example of the bit leakage using the L1 cache. Each physical page has one RMP entry. Due to the alignment of the RMP table, the lower 10 bits of the page frame number match the lower 10 bits of the index of the RMP table (here marked with an underscore). The bits of the PFN we are able to leak are colored in dark. It can be seen how these bits propagate from the PFN to the RMP table index and finally to the L1 cache set index.

directly to the six physical address bits $p_{14} \dots p_{20}$ of the target page (since those bits of the page's address determine the position of its RMP entry in memory). We illustrate an example of the leakage in Figure 2.

Leakage from L2 Cache Set. The L2 cache has more sets (and thus uses more address bits for indexing) than L1D. A natural question is whether these additional index bits allow leaking even more physical address bits. The complication is that the L2 cache uses a hash function involving address bits above the page offset, which makes it harder for an attacker to know exactly which L2 set an address maps to when they only control 4KB pages. In other words, an attacker building eviction sets on 4KB pages only has partial knowledge of each eviction set's "label" or index. They definitely know the page-offset bits (which influence the cache index) and, if the L2 attack builds on the L1 attack, they also know the physical bits $p_{14} \dots p_{20}$ from the L1 leakage. However, they do not know the higher-address bits that the L2's indexing function XORs in.

To evaluate the potential for additional bit leakage via the L2, we analyze the conditions under which a candidate eviction address e conflicts with an RMP entry at address r, i.e., when $H(e) = H(r)$. Analyzing the resulting system of linear equations reveals that, even when incorporating the L1 leakage, AMD's use of higher-order physical address bits in its L2 indexing function prevents leakage of any additional bits beyond those already exposed through L1. Specifically, on Zen 3 and Zen 4, bits p_{14} through p_{15} can be leaked, and on Zen 5, bit p_{16} as well—overlapping entirely with the bits already observable via L1.

7.3.1 Locating the RMP Entry Cache Set. To exploit the *bit-leakage* primitives discussed earlier, an attacker must first discover which L1/L2 cache set stores the RMP entry for a given page. We begin with an approach for Zen 5 systems that applies to both the L1 and L2 caches. We then present an alternative method for Zen 4 and Zen 5, which exhibit different RMP entry loading behavior. Finally,

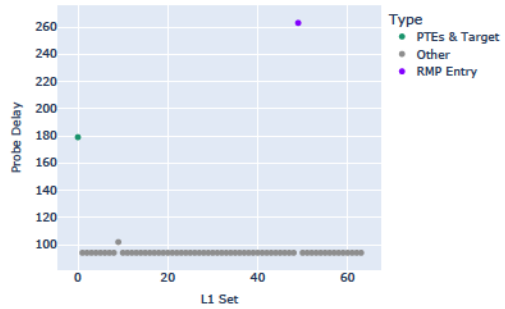


Figure 3: Prime+Probe measurements that reveal the L1 cache set holding RMP entry. Each set is measured separately, with values representing the average over 2,000 iterations.

we introduce an optimized isolation strategy tailored specifically for the L1 cache.

Zen 5. To isolate the RMP entry cache set on a Zen 5 system, an attacker performs two observations of page walks: First, they monitor the cache sets accessed during a write-induced page walk, recording the set S_W, which includes the RMP entry. Then, they observe the cache sets S_R accessed during a read-induced page walk to the same page. Since Zen 5 only loads RMP entries on writes, the difference $S\_\mathsf{RMP} = S_W \setminus S_R$ reveals the cache set corresponding to the RMP entry.

Zen 3/4. On Zen 3 and 4, an attacker can make use of two mechanisms available to unprivileged processes: (i) the `PREFETCH` instruction, which touches memory without raising faults, and (ii) the `mprotect` syscall, which can temporarily revoke page permissions. First, the attacker determines the cache set of the target page s using a direct Prime+Probe, without flushing the TLB. Next, they trigger a page walk to the same page in a normal read-write state and record the accessed cache sets S_{RW}. Then, they repeat the page walk using `PREFETCH` with the page in `PROT_NONE` state, obtaining S_{NONE}. Since `PROT_NONE` clears the `present` bit in the PTE, the page walk terminates at the PTE level, and S_{NONE} includes only cache sets of PTEs. The RMP cache set is therefore revealed by computing $S\_\mathsf{RMP} = (S_{\mathsf{RW}} \setminus S_{\mathsf{NONE}}) \setminus \{s\}$.

In both aforementioned strategies, collisions arise when the RMP cache set overlaps with cache sets of the PTEs or the page itself, yielding an empty S_{RMP}. The likelihood of such a collision depends on the number of cache sets and is on the order of $5 \times 2^{-10} \approx 0.5\%$ for the L2 cache on our Zen 5 machine.

L1 Cache. An attacker targeting the L1 RMP entry cache set can leverage controlled collisions to isolate the RMP set. Using the `mremap` syscall, the attacker maps the target page to a virtual address such that the page and all PTEs of the page map to the same cache set, for example, L1 cache set 0. They then monitor a page walk triggered by a write access to this page. If a collision occurs, it must be with cache set 0, revealing the RMP set as 0. If no collision occurs, two cache sets are observed, and the non-zero one corresponds to the RMP entry cache set. Figure 3 depicts the cache activity of the page walker for this optimized attack.

7.4 Experimental Results

We implemented the Prime+Probe attack described in this Section to leak physical address bits through a userspace process running directly on the Hypervisor. The experiments were conducted using the same systems and configuration described in Section 5. Our results demonstrate the efficacy of our attack. On SEV-SNP enabled machines, an attacker can leak six physical address bits beyond the page offset using our L1 RMP cache attack. On our Zen 5 machine, recovering six physical address bits for 100,000 pages took 24.9 s (2.5 µs per page) with a success rate of 99 %. Zen 4 and Zen 3 achieve success rates of 97 % and 99 % respectively, both with an average time of ~1.1 µs per page.

Result 6: RMP entries cached in the L1 data cache leak 6 physical address bits to user-level processes. The leak can be exploited fast and effectively.

7.5 Discussion

Cross Address Space Leakage. In this paper, we demonstrate the attacker's ability to leak physical address bits of addresses within its own address space. However, an attacker may also be interested in leaking physical address bits *across* address spaces (for example, physical kernel address bits). While we conjecture this to be possible using similar techniques as described in this paper, an attacker would need to overcome additional challenges. First, the attacker would need to leak the target page's TLB set across a context switch for an unknown virtual address (when ASLR is enabled). This has been demonstrated in prior work [11, 16]. Furthermore, the attacker must be able to distinguish between regular page table accesses and the RMP access, which reveals the PFN bits. For this, the attacker cannot leverage the methods described in 7.3.1, since they have no control over the victim program. Similarly, TLB misses of other accesses made by the victim program or context switching routines may introduce noise that the attacker would need to overcome.

Countermeasures. The attack uncovered in this paper leverages traditional cache side-channel techniques to infer the accessed RMP set. As such, general cache side-channel countermeasures apply to our attack as well. Common approaches include disabling caching, using fully associative caches, or employing randomized cache indexing [24]. These techniques can be applied to the entire data cache or, potentially, to a dedicated cache for RMP entries. Notably, disabling caching for RMP entries is a straightforward solution and may have limited performance impact, as RMP entries remain cached in the TLB.

Entropy-based solutions, such as randomizing the RMP's entries cache sets, could complicate the attack presented in this paper; however, this countermeasure is challenging to implement securely. For instance, simply randomizing the RMP's base address is susceptible to a de-randomization attack: if the attacker has access to 2MB huge pages, they can trivially de-randomize an RMP entry's cache set placement.

8 Related Work

Microarchitectural and side-channel analyses have repeatedly exposed fundamental weaknesses in AMD SEV's security guarantees.

Early work demonstrated that SEV's memory encryption lacked integrity protection and could not defend the CPU state—leading to attacks that recover unprotected registers [29] and manipulate ciphertext undetected [30]. Subsequent studies showed that even encrypted memory leaks information via TLB poisoning [21] and ciphertext-based side channels [17, 20], underscoring that encryption alone is insufficient. In response, SEV-SNP introduced RMP-based integrity checks to guard against such manipulations; however, the security implications of its RMP table have not been thoroughly examined, which motivated this work. Moreover, recent work has revealed that SEV-SNP still succumbs to high-resolution cache attacks [10], performance-counter side channels [9], and power-analysis attacks [28].

9 Conclusion

This work presents the first comprehensive analysis of RMP entry caching in AMD SEV-SNP systems spanning all microarchitectures that support SEV-SNP — Zen 3, 4, and 5. Our reverse engineering reveals that RMP entries are cached not only in the TLB but also in L1D and L2 caches. We demonstrate that a global TLB flush guarantees coherence of RMP entries cached in the TLB and that the inclusion of a `lock bit` within RMP entries prevents race conditions in their update process. Our novel findings enable us to construct a cache-based side-channel attack that allows an unprivileged process to leak 6 physical address bits reliably and fast. Furthermore, we point out several other security implications of our results, that can benefit both future defenses and attacks.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by the Air Force Office of Scientific Research (AFOSR) under grant FA9550-22-1-0511; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. It was also funded by the Federal Ministry of Education and Research of Germany (BMBF) through the program "Souverän. Digital. Vernetzt.", joint project 6G-RIC, project identification number 16KISK030, the program PQ-CCA, 16KIS2109, and the ALPAKA Grant, 16KIS1843. This work was also supported by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate (NDSEG) Fellowship Program.

References

[1] AMD. 2017. Protecting VM register state with SEV-ES. *White paper* (2017).
[2] AMD. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. *White paper* (2020).
[3] AMD. 2022. *SEV Secure Nested Paging Firmware ABI Specification.*
[4] AMD. 2023. SEV FW source code. https://github.com/amd/AMD-ASPFW. Commit 3ca6650, file `sev_rmp.h`, line 53.
[5] AMD. 2024. *AMD64 Architecture Programmer's Manual.* Volume 2: System Programming.
[6] AMD. 2024. *AMD64 Architecture Programmer's Manual.* Volume 3: General-Purpose and System Instructions.
[7] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. 2019. Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1087–1099.
[8] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. 2017. Secure Encrypted Virtualization is Unsecure. *arXiv preprint arXiv:1712.05090* (2017).
[9] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. 2025. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In *Network and Distributed System Security Symposium 2025.*
[10] Lukas Giner, Sudheendra Raghav Neela, and Daniel Gruss. 2025. Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 191–212.
[11] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. 955–972.
[12] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *Proceedings 2017 Network and Distributed System Security Symposium*. San Diego, CA.
[13] Felicitas Hetzelt and Robert Buhren. 2017. Security Analysis of Encrypted Virtual Machines. In *ACM SIGPLAN Notices*. ACM.
[14] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *2015 Euromicro Conference on Digital System Design*. 629–636.
[15] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).
[16] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 309–321.
[17] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (SP)*. 337–351.
[18] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2021. CrossLine: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event Republic of Korea). ACM, 2937–2950.
[19] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium*. 1257–1272.
[20] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 717–732.
[21] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *Annual Computer Security Applications Conference*. 609–619.
[22] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses*, Herbert Bos, Fabian Monrose, and Gregory Blanc (Eds.). Springer International Publishing, Cham, 48–65.
[23] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 565–581.
[24] Moinuddin K Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 775–787.
[25] Kirill A. Shutemov. 2015. *pagemap: do not leak physical addresses to non-privileged userspace.* Linux Kernel. Committed by Linus Torvalds on 2015-03-17 (ab676b7).
[26] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. 2022. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 989–1007.
[27] Han Wang, Ming Tang, Quancheng Wang, Ke Xu, and Yinqian Zhang. 2025. ZenLeak: Practical Last-level Cache Side-Channel Attacks on AMD Zen Processors. In *Proceedings of the 62nd Annual Design Automation Conference (DAC)*. San Francisco, CA, USA.
[28] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2023. PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer Nature Switzerland.
[29] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (Auckland, New Zealand) *(Asia CCS '19)*. Association for Computing Machinery, New York, NY, USA, 73–85.
[30] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1483–1496.