RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP

Benedict Schlüter
benedict.schlueter@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Shweta Shinde
shweta.shinde@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Abstract

AMD SEV-SNP offers confidential computing in form of confidential VMs, such that the untrusted hypervisor cannot tamper with its confidentiality and integrity. SEV-SNP, the latest addition, ensures integrity via the Reverse Map Table (RMP) that stops the hypervisor from tampering guest page mappings. AMD uses RMP entries to protect the rest of the RMP, thus causing a Catch-22 during the RMP setup phase. To address this, SEV-SNP relies on AMD's Platform Security Processor (PSP), that resides next to the x86 cores executing SEV-SNP VMs, to perform the RMP initialization. During initialization, only PSP should be able to alter the RMP memory. All other memory accesses must be fenced, especially from the x86 cores. We present RMPocalypse, a novel attack that shows a critical gap in the security of RMP initialization, wherein the x86 cores maliciously control parts of the initial RMP state. Our analysis shows that the vulnerability arises due to the complex, but insufficient, interplay of multiple hardware components and distributed access controls. To show the impact of our finding, we exploit this gap to break confidentiality and integrity guarantees of SEV-SNP. We demonstrate RMPocalypse by enabling debug on production-mode CVMs, faking attestation, VMSA state replay, and code injection.

CCS Concepts

• **Security and privacy → Hardware attacks and countermeasures**.

Keywords

SEV-SNP, confidential computing, virtualization, computer architecture

ACM Reference Format:
Benedict Schlüter and Shweta Shinde. 2025. RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP. In *Proceedings of the 2025 ACM SIGSAC Conf. on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765233

1 Introduction

AMD SEV-SNP is deployed in production by major cloud vendors such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud, and has real-world deployments [1, 19, 27, 28]. The main guarantee that SEV-SNP provides is that the cloud provider who can control the privileged software on the cloud servers cannot abuse that privilege, intentionally or accidentally, to tamper with the cloud tenant's computation. In particular, SEV-SNP introduces the notion of confidential VMs, CVMs for short, that host the tenant workloads, i.e., code and data. SEV-SNP allows the hypervisor to perform its resource management tasks (e.g., scheduling, memory management) while ensuring that the hypervisor cannot leak the tenant's data or tamper with its execution.

SEV-SNP encrypts and integrity protects the CVM register state and memory to stop confidentiality attacks. For integrity, SEV-SNP introduces the concept of a reverse map table, RMP for short. It ensures 1:1 mapping between host and guest pages. This way, a hypervisor cannot create malicious page mappings (e.g., assigning the same physical page to two CVMs or sharing it with the hypervisor). In particular, SEV-SNP's RMP addition was motivated by integrity attacks on its predecessor SEV-ES [37]. Tampering with the RMP would result in a complete compromise of SEV-SNP. Knowing that the RMP is the linchpin of SEV-SNP, AMD has several checks and balances to safeguard the RMP throughout a server's lifecycle.

When an AMD server initializes SEV-SNP, it implicitly also initializes the RMP. It goes without saying that a good starting state of RMP is critical to enforce 1:1 mapping. Once initialized correctly, the hypervisor can launch CVMs by assigning physical memory to them. Now, since the RMP keeps track of the page mappings, any change in the ownership of physical pages must be tracked in the RMP. This tracking is done on a per-page basis, where each physical address is represented by a 16-byte RMP entry. Modern servers have large DRAM capacities with 1-4 TiB being standard for cloud deployments [17]. This results in a sizeable RMP (16GiB). Maintaining such a large data structure on-chip is infeasible, so AMD stores the RMP in the DRAM. This leads to a classic dilemma: *who protects the protector?* AMD has an elegant solution: since the RMP is meant to protect DRAM pages, it can also protect itself. Given a well-formed RMP state, SEV-SNP can stop the hypervisor from tampering with the physical page that contains RMP entries—by simply refusing the hypervisor to map an RMP-owned physical page to itself. Similarly, any changes to the memory mappings (e.g., launching a new CVM) that result in an RMP update can be mediated by a trusted hardware/firmware entity. The only missing piece of this puzzle is bootstrapping, i.e., how to setup the initial RMP state when no RMP exists.

AMD SoCs with SEV-SNP support have several x86 cores that perform most of the workload computation and a secure co-processor called the PSP, short for Platform Security Processor. The PSP is not a general-purpose processor but is strictly for enforcing security onto the x86 cores and the memory subsystem, as well as tasks such as preparing attestation reports. AMD uses the PSP to perform the RMP initialization. Specifically, the hypervisor does several preparatory steps (e.g., creating a physical region of memory that

CCS '25, Taipei

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765233

will hold the RMP). The main goal of the PSP is to check the correctness of the hypervisor-provisioned configuration. Since this is a critical task, once the hypervisor requests a RMP initialization, the PSP then accesses diverse platform state registers to perform its checks (e.g., MSRs configuration). At this point, there is no way for the hypervisor to write to RMP memory directly, thus allowing a secure initialization.

We present a novel attack RMPocalypse to show that, despite its best efforts, the RMP initialization has a severe flaw—the PSP does not properly protect the memory containing RMP during initialization. Due to the PSP's incomplete protection, we find that there is nothing that stops the hypervisor from writing to the RMP memory. Once we corrupt one RMP entry, we can compromise all SEV-SNP guarantees. We experimentally confirm RMPocalypse on the Zen 3, Zen 4, and latest Zen 5 processors. We analyze CVM-specific metadata and use it as corruption targets. Building from this, we show four distinct ways how RMPocalypse can be used to fully break SEV-SNP with attestation forgery, enabling debug, arbitrary code injection, and replay of the CVM register state.

Contribution. RMPocalypse is the first attack on AMD SEV-SNP that demonstrates and exploits a critical gap in RMP initialization to fully break confidential computing guarantees.

Coordinated Disclosure. We reported the vulnerability to AMD on the 3rd of February, 2025. AMD acknowledged the vulnerability, thanked us for the disclosure, and issued CVE-2025-0033. Our code is public at https://github.com/RMPocalypse.

2 Background

AMD SEV enables CVMs, which are managed by the hypervisor (e.g., memory management, scheduling). SEV encrypts the guest memory but does not ensure integrity or confidentiality of the register state. Encryption works by setting a bit, called the C-bit, in the physical address. The C-bit propagates to the memory controller and informs it that the memory access should happen in an encrypted form. SEV is susceptible to attacks (e.g., observing/-manipulating the unencrypted register state) [20]. Subsequently, AMD introduced SEV-ES with register state encryption [22]. This eliminated the attacks that were possible on the previous generation of SEV CVMs. But crucially, SEV-ES lacked memory integrity guarantees, rendering the CVMs vulnerable to attacks based on malicious page mappings [29, 30, 37].

Integrity Attack on SEV-ES. The hypervisor remains responsible for memory management and programs the second-level address translation (SLAT) page tables, which map guest physical address (GPA) to host physical address (HPA). In Figure 1, function `auth` located at GPA `0x1000` and function `dummy` (`dummy` always returns 0) located at GPA `0x2000` are on 2 different pages. The memory encryption prevents the hypervisor from writing meaningful data directly to the pages. The decryption process will result in random values. But the hypervisor can program the SLAT tables to switch the 2 pages, i.e., map the GPA that was pointing to `auth` to point to `dummy` like in Figure 1 (a). Now, when the victim code calls auth, it will invoke the dummy function, thus always succeeding in authentication. To fix the flaw, AMD introduced SEV-SNP with memory integrity protection along with SEV-ES's register state

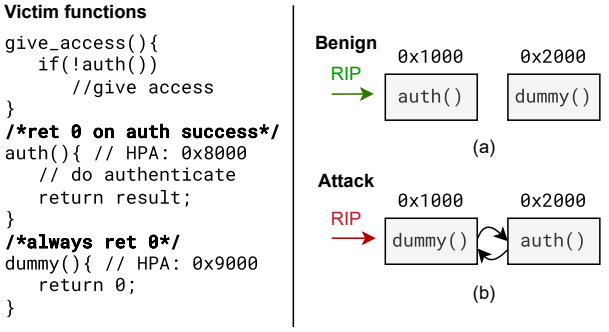

Figure 1: (a) Normal execution of CVM (b) Hypervisor swaps the guest page tables and tricks the VM into executing code on another page. SEV-ES is susceptible to such attacks.

encryption. To ensure memory integrity, SEV-SNP uses the RMP. Next, we explain how RMP works and how it is initialized.

2.1 Reverse Map Table (RMP)

RMP is a reverse map table from the HPA to GPA that ties the GPA and HPA. For Figure 1, this would tie HPA `0x8000` to GPA `0x1000` and HPA `0x9000` to GPA `0x2000`. SEV-SNP checks the RMP before any memory access from the x86 cores. Importantly, unlike the SLAT tables, the hypervisor cannot arbitrarily change the RMP. SEV-SNP strictly controls how the RMP is set up and what operations the hypervisor is allowed to perform on it. With SEV-SNP, the hypervisor still manages the SLAT tables and can perform the remapping of the pages like in Figure 1 (b). However, this will not compromise the CVM. When the CVM invokes the `auth` function at GPA `0x1000` that now maps to HPA `0x9000`, the RMP will detect that the translation resulted in a different HPA and will abort the memory request, stopping the attack. While tying the HPA to the GPA stops the hypervisor from mounting page-swapping attacks, it alone cannot prevent the hypervisor from directly writing to the CVM's pages. To prevent this, the RMP entries also store information about which pages belong to a CVM. The RMP checks stop any hypervisor writes to CVM pages.

Updating RMP. For security, it is crucial to ensure that the untrusted hypervisor cannot change RMP directly. However, the hypervisor needs to update the RMP for certain operations (e.g., CVM memory management operations). To allow this functionality, AMD introduced the RMPUPDATE instruction that the hypervisor uses. RMPUPDATE is subject to strict checks from the SEV-SNP hardware; the hypervisor is only allowed to perform specific actions (e.g., creating pending pages a CVM needs to accept). The instruction blocks all security-undermining actions. If the hypervisor wishes to perform operations not allowed by the RMPUPDATE instruction (e.g., create valid CVM pages), it has to invoke AMD's platform security processor (PSP) APIs. The PSP is a privileged co-processor on the AMD chipset and can modify RMP entries. Furthermore, the PSP can apply additional security checks to the modification request (e.g., check that the CVM exists by comparing it with the internal state). Therefore, with SEV-SNP, the RMP can be changed from the x86 cores (using RMPUPDATE) and from the PSP.

RMP Initialization: Bootstrapping Security of RMP with RMP. The RMP contains entries that protect it to prevent the hypervisor from arbitrarily writing to the RMP directly. However, during RMP initialization, the RMP is not fully bootstrapped, so it does not have self-protection. But for self-protection, the RMP must be fully initialized. This leads to a Catch-22 problem where the RMP relies on the RMP for protection. The PSP has to ensure that only the PSP can modify the RMP during initialization, and any writes from x86 cores are blocked.

2.2 Platform Security Processor

The AMD Platform Security Processor (PSP), also known as AMD Security Processor (ASP) or Security Processor (SP), is an Arm Cortex A5 processor embedded into the x86 platform [9, 16]. The PSP acts as the root of trust for SEV-SNP and is the most privileged instance on an AMD chipset. The interacts with the x86 cores and can write directly to the x86 DRAM. The interface connecting the platform with the PSP and the exact location of the PSP are not publicly documented. The hypervisor requests PSP services through a set of MMIO registers at runtime. Should the hypervisor not have sufficient permissions to change an RMP entry through RMPUPDATE, it requests the PSP to execute the transition. Since the PSP is more privileged and has more capabilities, it can enforce strict requirements that might be necessary to perform the RMP update securely. For instance, one of these requirements might be that a global write-back and invalidation of all caches must be executed before certain RMP state transitions are allowed. Besides RMP lifecycle management, the PSP also initializes the RMP. Initialization is particularly challenging since the RMP uses itself to protect itself from malicious hypervisor writes to its memory. During initialization, while the RMP is not yet fully initialized, the PSP relies on additional platform security controls to block the x86 cores from writing to the RMP.

2.3 Memory Encryption

AMD SEV encrypts the memory of both the guest and the host. Encryption takes place at the memory controller [6]. This implies that all data on the bus between the memory controller and the caches is unencrypted. Since individual guests use different encryption keys, the CPU core must inform the memory controller about which key to use. SEV-SNP uses the upper bits of the physical address to transport the key ID to the memory controller [6]. On AMD EPYC Gen 4 and 5, this results in an effective reduction of the physical address space to 46 bits [7]. Memory requests whose physical address has the so-called C-bit set will be encrypted by the memory controller. Zen 4 and Zen 5 ensure cache coherency between memory requests with the C-bit set and those without. For Zen 3, the cache coherency for the encrypted and unencrypted domains must be ensured manually by the OS or the PSP [7].

3 AMD SEV-SNP Initialization

We analyze the SEV-SNP initialization flow by sourcing information from official documentation, source code, and patents. We first explain our understanding of how the PSP initializes SEV-SNP and continue with insights about the AMD EPYC platform.

3.1 PSP Firmware

The SEV firmware runs on the PSP and is responsible for several lifecycle tasks of SEV-SNP. Most importantly for our analysis, it initializes SEV-SNP and the RMP in memory. The hypervisor triggers the initialization routine through an API call, `SNP_INIT_EX`, to the PSP.

What does the hypervisor do before SNP initialization? The hypervisor ensures that the platform is in a pre-defined state for successful command execution. The PSP does not enable SNP if any of the prerequisites are not fulfilled [8]. We group the prerequisites into 3 categories, based on our source code analysis of the PSP firmware:

(1) *MSRs monitoring the current processor state*. During SEV-SNP initialization, all x86 cores are required to execute in normal execution mode. Specifically, the cores should not execute in System Management Mode (SMM) or already run a Virtual Machine. The PSP reads an undocumented MSR to validate the x86 core execution states [6].
(2) *MSRs defining the memory routing*. Similar to the checks for the undocumented MSR above, the PSP checks x86 MSRs that define how memory accesses are routed. Depending on those registers, certain memory requests might be routed to MMIO space rather than DRAM space. The PSP needs these registers (e.g., TOM, TOM2) to be locked and equal across all cores. This ensures a uniform memory view and allows the PSP to exclude MMIO pages from the range of valid guest pages within the RMP.
(3) *MSRs defining system and SEV-SNP features*. The PSP checks SEV-SNP-specific registers for their value and ensures they are identical across all cores. The BIOS must enable SEV-SNP through an MSR and set the RMP start and end registers equally between all cores. Furthermore, the memory controller must have the encryption enable flag set so that it can receive the encryption keys and perform the encryption. The PSP continues with preparation for the RMP initialization, once it confirms that the hypervisor has met all the prerequisites.

RMP Initialization. The PSP starts the RMP initialization by locking writes from x86 cores to the RMP memory. The x86 core lock of RMP memory acts as a first barrier to prevent x86 cores from creating memory requests targeting the RMP memory region. All x86 cores must acknowledge the setting before the SEV firmware continues [6]. Subsequently, the PSP creates a second memory barrier at the memory controller (see TMRs in Section 3.2.2), preventing x86 cores and devices from writing to RMP in the DRAM. The two barriers, as depicted in Figure 2 (b), prevent the x86 cores from overwriting the RMP entry. Ultimately, the PSP writes the initial RMP configuration to DRAM.

Finalization. Once the PSP finishes writing the RMP, it modifies the memory controller barriers such that x86 cores are allowed to write to the RMP again. Note that at this point the RMP checks are in place. As a last step, the SEV firmware performs another call to the PSP's operating system to enforce SEV-SNP semantics globally. After initialization, x86 cores can manipulate the RMP through specific RMP instructions, like RMPUPDATE, while microcode or hardware blocks direct writes to RMP memory.

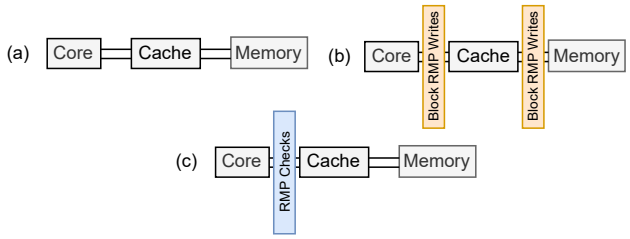

Figure 2: (a) SNP is not yet initialized; all RMP memory is writable. (b) The PSP creates two barriers to prevent x86 cores and other system entities, such as DMA devices, from writing to RMP memory. (c) RMP Init is executed successfully, and all x86 cores now enforce RMP semantics.

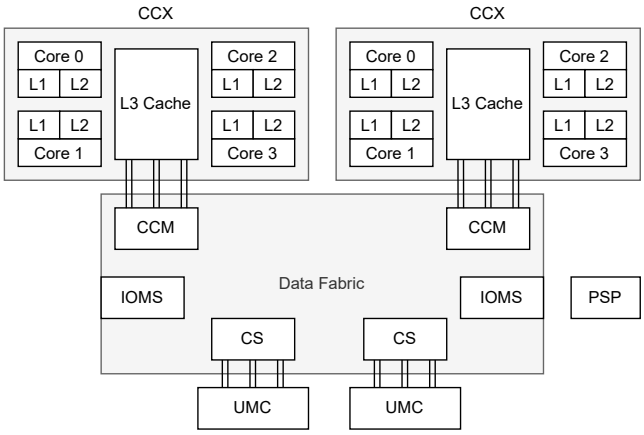

Figure 3: Overview of AMD EPYC Data Fabric interconnect.

3.2 Memory & Interconnect

Figure 3 shows a high-level overview of the components that work together during the RMP initialization.

3.2.1 Data Fabric. It transports data between x86 cores, memory controllers, and I/O devices. AMD's interconnect is called the Infinity Fabric (IF) and consists of the Data Fabric (DF), and Control Fabric (CF) [10]. Throughout this paper, we exclusively look at the Data Fabric.

AMD Zen makes use of a chiplet architecture, meaning Core Complex (CCX) blocks are individually created and later connected to the I/O Die. This allows for better yields with increasing core count, as multiple chiplets combined form a high-core-count processor. CCX blocks host the main computing resources of the platform, i.e., the x86 cores. Each CCX block can host between 4 and 8 physical Zen cores, depending on the architecture [3, 5]. Attached to each CCX core is an L1 and L2 cache. The L3 cache, although shared between all cores, is sliced into the different CCX blocks on the platform. Core Coherent Master (CCM) units connect the CCX to the Data Fabric and manage memory routing on their behalf.

Next, we have Coherent Slaves (CS) blocks on the Data Fabric. CS blocks connect to one Unified Memory Controller (UMC) instance. Most importantly, CS blocks are responsible for ensuring memory coherence. If data does not reside within the local cache (L1 / L2 / L3 slice) of a CCX, a memory request is issued that eventually lands at the CS. If the CS has already sent the data to another CCX, the memory request is forwarded to the respective CCX. The Data Fabric itself caches data while it is in transit through queues.

3.2.2 Memory Access Checks. SEV-SNP uses two means of memory barriers to protect its integrity.

1. Trusted Memory Regions (TMRs). They offer complementary memory access filtering at the memory controller. Each memory request has an originating Data Fabric component (e.g., CCM). TMRs filter memory requests based on the source of the request and may read or write-protect the underlying memory regions. Only the PSP can install and remove TMRs as they ensure the security of SEV-SNP and other components (e.g., protect MSR memory). TMRs do not differentiate between x86 and microcode access [6]. Both accesses originate from the same CCM and are treated equally by the TMRs. Thus, if the microcode can write to a memory region, the hypervisor can potentially also write to the memory region. During SEV-SNP initialization, TMRs block all writes to the RMP memory. If a memory request does not pass the TMR checks, it is silently discarded. After SEV-SNP and RMP initialization, the PSP removes the TMR that was placed to protect the RMP. As the RMP is in place to protect the memory, it now allows read and write access from the CCM units connecting the x86 cores.

2. RMP / x86 Access Checks. Apart from TMR protection, SEV-SNP uses another protection mechanism to block memory requests. Instead of enforcing access checks at the UMC boundary, the RMP access checks are performed at the x86 cores. Each memory write and certain reads are subject to RMP checks. Microcode or hardware consults the RMP to check whether the x86 memory access is allowed. In case of a policy violation, the hardware raises a page fault and stops the memory access before it leaves the core [7]. The hypervisor cannot change the RMP, as all instructions are subject to the RMP access checks. Every instruction attempting to access RMP memory is blocked by the RMP. For functionality, AMD adds multiple instructions (e.g., RMPUPDATE) that bypass the aforementioned checks. The added instructions offer a highly restricted API to change the RMP. AMD implements the security checks for the RMP-modifying instructions in microcode [7]. Furthermore, AMD employs undocumented protection barriers on so-called `MP` elements to safeguard the RMP during initialization [6]. The PSP activates these barriers using the same semantics to enable SEV-SNP globally. Section 8 discusses additional details and security considerations of the RMPUPDATE instruction.

3.2.3 Memory Coherency. Modern systems must ensure memory coherence between different cores. AMD uses the `MOESDIF` protocol for this purpose [7]. x86 cores might opt out of the coherency on purpose by mapping the same memory region with different caching attributes. However, this is a direct threat to SEV-SNP security. The hypervisor might perform split-view attacks on guest memory, where the guest has dirty data in the caches, but the non-coherent memory type forces the access to DRAM and does not access the caches with stale data. AMD defines two non-coherent memory types capable of bypassing memory coherence. Uncachable (UC) memory bypasses the caches and directly accesses DRAM

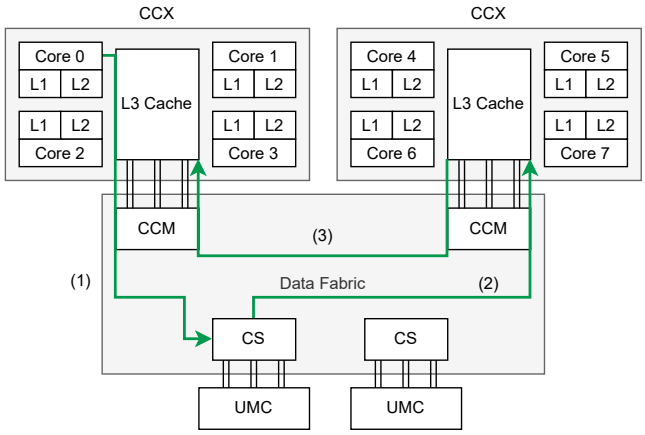

Figure 4: Coherent memory request flow if cache line resides in another CCX. (1) The core sends the memory request to the memory controller. (2) The memory controller sends a snoop request to the CCX holding the cache line. (3) The CCX holding the cache line responds with the correct cacheline.

without respecting coherency; and Write-Combining (WC) memory, where the memory subsystem uses a private write buffer to buffer memory that is invisible to other cores modifying the same memory. Apart from two non-coherent memory types, all x86 memory types are coherent. Hardware transitions the two non-coherent types into coherent types for SEV-SNP guest access [7]. Thus, coherency issues from the x86 core pertaining to SEV-SNP guest memory access should not be possible. AMD also ensures coherence at the CS blocks. Whenever these blocks receive a memory request, they either let it pass to DRAM or send a broadcast request to probe the state of all x86 core caches. Figure 4 depicts a snoop request.

AMD Zen 3 EPYC CPUs do not offer automatic coherency between the encrypted and unencrypted domains (C-bit set and not set). The CPU can have the plaintext and the ciphertext versions dirty in its caches with different values. An overwrite of the plaintext or ciphertext data does not cause the eviction or update of the other version. For instance, if the PSP writes to the RMP memory and at the same time the CPU has dirty cachelines of RMP memory but in the ciphertext domain, the accesses do not snoop the caches and the dirty cachelines remain in the cache. This behavior is only present on Zen 3, as Zen 4 and 5 automatically ensure coherence even between encryption domains. These details become important when we discuss the countermeasures for the RMPocalypse attack and what care must be taken.

4 RMPocalypse Attack

Analyzing the security of the RMP is not straightforward, as many details on how the internal protection mechanisms work are proprietary and not publicly available. To this end, our goal is to study how SEV-SNP protects the RMP, specifically during initialization. In combination with the PSP source code, we analyze what occurs at AMD SEV-SNP initialization and hypothesize about the lack of implemented protection mechanisms. Our experiments testing the security of SEV-SNP initialization flow show that we can overwrite the RMP and corrupt its internal state. We confirm the existence of the vulnerability on the processors in Table 1. Our attack, RMPocalypse, bypasses all deployed protections to corrupt the initial RMP configuration. However, the root cause of the vulnerability is not immediately obvious when studying the PSP source code and the documentation, and requires further experiments.

Table 1: AMD EPYC processors vulnerable to RMPocalypse

Generation	Test CPU	Launched	SEV Firmware	Microcode
Zen 5/Turin	EPYC 9135	10/10/2024	1.55 build 44	0x0b002116
Zen 4/Genoa	EPYC 9124	10/11/2022	1.55 build 43	0x0a101154
Zen 3/Milan	EPYC 7313	15/03/2021	1.55 build 23	0x0a0011d5

Thread Model. We operate in the default confidential computing threat model. The hardware, especially the CPU and all components provided by the CPU manufacturer, and the firmware are trusted. The hypervisor is untrusted and can operate maliciously. Furthermore, we assume the hypervisor can create and run CVMs.

4.1 Experiments

Since the exact initialization steps are not publicly documented, we start with the initial goal of reverse engineering the protection checks. To test this, we perform the following experiment on a Zen 5 machine: we create a kernel thread running on the x86 host that repeatedly writes to RMP memory as the PSP activates the barriers asynchronously (see Section 3.2.2). Thus, the kernel thread writes continuously to RMP memory in a while loop and stops once SNP initialization is done or it encounters an exception. We isolate the kernel thread by pinning it to a CPU core. We further synchronize the SNP activation call with the kernel thread. The step is necessary to ensure that our thread is running during SNP initialization. Without it, the thread races with the SNP initialization, and we encountered cases where SNP activation has already finished before the thread reached the while loop. We modify the Linux exception handler to jump to the `exit_jump` label on Line 5 in Listing 1 if it encounters an exception in our code. Listing 1 shows our code.

```
1 [...]
2 atomic_set(overwriting_thread_initialized,1);
3 local_irq_save(flags);
4 while (!kthread_should_stop()){
5     WRITE_ONCE(rmptable[0],0x4);
6 }
7 exit_jump:
8 local_irq_restore(flags);
```

Listing 1: RMPocalypse Exploit code.

When we run the code in Listing 1, we change the value in `rmptable[0]` from `0x0` to `0x4`. This shows that we can corrupt the initial RMP configuration and write arbitrary values to RMP memory. Note that, no matter what the root cause might be, we can use the above primitive to perform the attack described in Section 6 to undermine AMD SEV-SNP security guarantees.

Initial Assumption. According to the source code and comments in the documentation, we could only explain the behavior by assuming a memory coherency issue: the PSP writes in a non-coherent manner to the x86 DRAM; dirty x86 cachelines pointing to the RMP are untouched during init and overwrite the RMP once the PSP

lifts the TMR barriers. As described in Section 3.2.2, it seems two barriers protect the RMP during init, and the only way to write to the RMP is before the PSP installs those two barriers. The SEV-SNP ABI specification [8] states: "Before invoking SNP_INIT_EX with INIT_RMP set to 1, software must ensure that no CPUs contain dirty cache lines for the memory containing the RMP". Combined with comments in the PSP source code for Zen 4 that discuss the missing coherency of PSP writes [6], we have a strong reason to assume that the PSP and x86 cores probably do not enforce coherency.

4.2 Root Cause Analysis

Based on the RMP initialization lifecycle summarized in Section 3, we perform a sequence of experiments based on reverse engineering and hypotheses to identify the root cause of our observation.

Hypothesis 1: Lack of Coherency. We start with the assumption that the PSP is non-coherent and that the hypervisor can evict dirty cachelines in the x86 caches after RMP initialization completes. To test our hypothesis, we need a CPU that indeed has missing cache coherency, to rule out other factors that might influence our experiments (e.g., timer-based cache flushes [4]). Recall that Zen 3 does not enforce coherence between the encrypted and unencrypted domains.

```
1  local_irq_save(flags);
2  WRITE_ONCE(rmptable_nocbit[0],0x4);
3  wbinvd();
4  ncbit = READ_ONCE(rmptable_nocbit[0]);
5  cbit = READ_ONCE(rmptable_cbit[0]);
6  while (!kthread_should_stop()){
7      tmpNC = READ_ONCE(rmptable_nocbit[0]);
8      tmpC = READ_ONCE(rmptable_cbit[0]);
9      if (tmpNC != ncbit){
10         if(tmpC == cbit){
11             pr_info("PSP most likely coherent\n");
12         }
13     }
14 }
15 exit_jump:
16 local_irq_restore(flags);
```

Listing 2: Code to Validate the Cache Coherency Hypothesis.

Listing 2 shows a slightly modified version of our initial Listing 1 exploit, which we run on our Zen 3 system. Specifically, we access the encrypted and unencrypted pages in the same pattern to make sure they are most likely subject to the same eviction policy in the caches. If PSP writes are memory coherent, we will only see an update in the unencrypted domain, as the PSP writes memory requests to the RMP without the C-bit set/ If PSP writes are non-coherent, we will either see no change in both values, or changes in both (encrypted and unencrypted), as the fabric has automatically flushed the caches [4]. In our experiments, we repeatedly see the print statement on line 11. From this, we deduce that the PSP is more likely to be memory coherent and that coherency is probably not the underlying issue.

Hypothesis 2: Missing Barrier. Next, we hypothesize that a barrier, which should be present according to the PSP source code [6], might be missing. As the PSP performs the RMP initialization asynchronously with respect to the x86 cores, we cannot precisely time the writes. We wait until the PSP writes a certain RMP entry, and when we observe a change in the entry, we simply overwrite it again, as shown on Line 7 in Listing 3.

```
1  local_irq_save(flags);
2  WRITE_ONCE(rmptable[0],0x4);
3  wbinvd();
4  while (!kthread_should_stop()){
5      tmp = READ_ONCE(rmptable[0]);
6      if (tmp != 0x4)
7          WRITE_ONCE(rmptable[0],0x4);
8  }
9  exit_jump:
10 local_irq_restore(flags);
```

Listing 3: Code to Validate the Missing Barrier Hypothesis.

If our write goes through, we know that one or both barriers must be missing. When executing the code, the final RMP entry is 0x4, showing our hypothesis is true. Next, we need to narrow down which of the barriers are missing. Note that the code without the RMP overwrite in the while loop does not change the RMP after initialization.

Possibility 1: Missing TMR Barrier. To check the existence of the TMR barrier, we repeat the experiment from Listing 3, but map the page as uncachable on the x86 host. This will ensure that our writes directly hit the memory controller and are not cached in the L1, L2, or L3 cache. If the TMR barrier is present, we will not see the value 0x4 as a final RMP entry. The experiment shows 0x0 as the final value, which proves that Line 7 in Listing 3 does not overwrite the RMP memory. From this, we deduce that the TMR barrier must be present and it protects the memory range from overwrites. The experiment also shows that the x86 core barrier is either non-existent or inactive.

Possibility 2: Inactive x86 Barrier. According to the source code, the PSP activates the barrier in the same way it activates the RMP checks [6]. SEV firmware performs a syscall to its PSP operating systems (no source code available), but with different flags. Since x86 cores cache RMP access permissions in the TLB, we suspect the barrier mentioned in the PSP source code might be TLB-based. If so, a flush after activating the barrier would be necessary by the PSP, which currently does not happen, according to the publicly available source code. To confirm if AMD simply forgot a global TLB flush initiated from the PSP, we added the TLB invalidation manually to the x86 core.

```
1  local_irq_save(flags);
2  WRITE_ONCE(rmptable[0],0x4);
3  wbinvd();
4  while (!kthread_should_stop()){
5      tmp = READ_ONCE(rmptable[0](;
6      if (tmp != 0x4){
7          asm ("invlpg(%0)" ::"r"(rmptable) : "memory");
8          WRITE_ONCE(rmptable[0],0x4);
9      }
10 }
11 exit_jump:
12 local_irq_restore(flags);
```

Listing 4: Code to Validate the Missing TLB Flush Hypothesis.

We add the TLB flush just before the second overwrite on Line 8 in Listing 4. The final RMP entry is still 0x4 after executing the code in Listing 4. We conclude that the issue is not a missing TLB flush.

Possibility 3: Non-Existent x86 Barrier. By elimination, we conclude that no x86 core barrier is active. This is why the x86 cores can create arbitrary cache entries pointing to RMP memory during

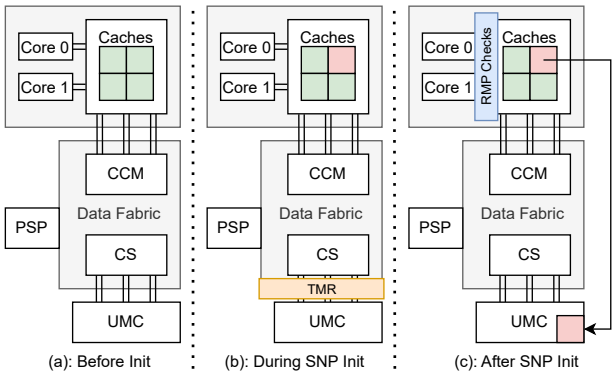

Figure 5: (a) No RMP protection exists prior to initialization. (b) The attacker calls `SNP_INIT_EX` to initialize SEV-SNP. During initialization, only a TMR barrier exists, allowing the attacker to create dirty cachelines pointing to RMP memory. (c) Once the PSP finishes SEV-SNP initialization, it destroys the TMR region. Furthermore, x86 cores now perform RMP checks on all memory accesses. However, the dirty cacheline pointing to the RMP still exists. The attacker flushes the cacheline and successfully overwrites the initial RMP configuration in DRAM.

initialization. Although the TMR prevents flushing these entries during initialization, after the initialization finishes, the cache entries overwrite the RMP and corrupt the initial RMP configuration. Figure 5 illustrates an overview of the attack and what happens on a macroarchitectural level.

5 RMP and States

RMPocalypse enables us to overwrite RMP entries on initialization. Next, we leverage RMPocalypse capabilities to break the security of SEV-SNP. We discuss all RMP entries in the system and the logic state they belong to. Then we discuss the impact of corrupting them with RMPocalypse.

5.1 RMP States

AMD defines ten page states in SEV-SNP. Each state defines specific access attributes. Microcode uses the state to allow or deny memory requests. Table 2 maps all publicly documented page states for SEV-SNP to the RMP entry bits defining their state. RMPocalypse allows an attacker to write arbitrary data to the RMP, effectively introducing new transitions between the states [8]. Furthermore, with RMPocalypse, the hypervisor can execute state transitions that are strictly reserved for the PSP. To understand the capabilities of our attack, we must introduce the implications of different page states. We denote the name of the state in *italics* font, since some page state names are overloaded.

Hypervisor. *Hypervisor* pages have no RMP restrictions. The hypervisor may use them for regular operation (e.g., as data memory) or promote them to other pages through RMPUPDATE. *Hypervisor* pages are readable and writable, and may be encrypted in DRAM depending on the system setting.

Table 2: Documented SEV-SNP RMP page states.

State	Assigned	Validated	ASID	Immutable	GPA	VMSA
Hypervisor	0	0	0	0	0	0
HV-Fixed	0	0	0	1	0	0
Reclaim	1	0	0	0	0	0
Firmware	1	0	0	1	0	0
Context	1	0	0	1	0	1
Metadata	1	0	0	1	0	0
Pre-Guest	1	0	guest	1	GPA	0
Guest-Invalid	1	0	guest	0	-	0
Pre-Swap	1	1	guest	1	GPA	0
Guest-Valid	1	1	guest	0	GPA	0

HV-Fixed. *HV-Fixed* pages have no RMP restrictions, and the hypervisor can read and write to them. However, as the name suggests, they cannot be promoted into any state other than *HV-Fixed*. The SEV firmware marks security-critical pages as *HV-Fixed* (e.g., MMIO memory). The hypervisor may additionally supply a list of pages it wants to mark as *HV-Fixed* to the SEV-SNP firmware initialization call.

Reclaim. *Reclaim* pages serve no active purpose and are the resulting page state of pages that the SEV firmware no longer needs. Once a page is in the *Reclaim* state, it can be reclaimed by the hypervisor through RMPUPDATE. The hypervisor may ask the firmware to transform *Firmware* pages into the *Reclaim* state by using the SEV API `SNP_PAGE_RECLAIM`.

Firmware. *Firmware* pages are reserved for SEV firmware use. The hypervisor cannot write to, nor can it directly change, the RMP entry of *Firmware* pages. The SEV firmware may use *Firmware* pages freely for its operation or transition them to *Context* or *Metadata* pages. The hypervisor can create *Firmware* pages using RMPUPDATE on a *Hypervisor* page. Most notably, the memory backing the RMP is also in the *Firmware* state, meaning the RMP uses itself for protection.

Context. *Context* pages hold the internal guest context used by the PSP for managing the CVM. *Context* pages contain the most sensitive information (i.e., the attestation value and SEV-SNP optional security features). The PSP encrypts the *Context* page inline and does not rely on the memory controller to perform the encryption. Performing inline encryption ensures the context information remains encrypted even while passing through the Data Fabric. Only the PSP can decrypt the memory in a *Context* page. The RMP protects *Context* pages from hypervisor writes.

Metadata. *Metadata* pages hold security-relevant information, such as the hash of swapped-out pages in the system. The hypervisor may swap active guest pages out of DRAM to disk. *Metadata* pages ensure the integrity of the swapped-out pages by storing the hash and other state information about the pages. The RMP protects *Metadata* pages from hypervisor writes.

Pre-Guest. *Pre-Guest* pages are non-active pages and ensure one invariant: neither the guest nor the hypervisor can write to the page. The PSP can securely operate on *Pre-Guest* pages as no x86 entity can change the page content. The hypervisor creates *Pre-Guest* pages using the RMPUPDATE instruction. Operations such as `SNP_MOVE_PAGE` require both the destination and the source page to be in the *Pre-Guest* State. Without the invariant, the PSP might

move the page, and the guest is still writing to it. *Pre-Guest* pages prevent race conditions between the PSP and x86 cores.

Guest-Invalid. *Guest-Invalid* pages are the ones that are about to be handed over to a guest. The hypervisor creates *Guest-Invalid* pages by using RMPUPDATE. Subsequently, a guest may use the `PVALIDATE` instruction to accept the page from the hypervisor. Accepting the page results in the *Guest-Invalid* page being transitioned into a *Guest-Valid* page. However, the content is not re-encrypted with a key. *Guest-Invalid* pages may only be used to supply the guest with more memory, but not to supply the guest with the initial state memory. The PSP supplies the initial memory to the guest using a special SEV firmware command, `SNP_LAUNCH_UPDATE`. *Guest-Invalid* pages are non-active pages.

Pre-Swap. *Pre-Swap* pages serve a similar purpose to *Pre-Guest* pages. Before swapping a guest page to disk, the hypervisor uses RMPUPDATE to transition a *Guest-Valid* page to *Pre-Swap*. *Pre-Swap* pages can only be modified by SEV firmware as they have the validated and immutable bit set. Furthermore, *Pre-Swap* pages also serve as a temporary page state for other SEV firmware commands (e.g., `SNP_PAGE_MOVE`).

Guest-Valid. *Guest-Valid* pages represent SEV-SNP guest memory. *Guest-Valid* pages store guest data and are encrypted in DRAM. The hypervisor can read encrypted *Guest-Valid* pages but cannot write to them.

Undocumented Page States. Apart from the publicly documented RMP states, we observe at least two undocumented states. Pages holding the VMSA register state of a CVM vCPU have two different unique RMP entries depending on if the VMSA is in use or not.

- Idle VMSA RMP entry `0x6010ffffffff001`
- In-Use VMSA RMP entry `0xfff0ffffffff001`

The official SEV-SNP ABI documentation does not include any information regarding the states [8]. Most likely, AMD uses these RMP entries to mark a VMSA in use such that the same register state cannot be reentered twice. Further, the PSP sets the ASID of a page state temporarily to `0x3FF`, an unused ASID, to lock a 2MiB sub-region from updates [6]. AMD may define other undisclosed states with internal meaning in the RMP.

5.2 RMP Structure in Memory

Two Model Specific Registers (MSRs) define the RMP in memory. MSR `0xC0010132` marks the beginning and MSR `0xC0010133` the limit of the RMP. According to public documentation, the first 16KiB of the RMP are used for bookkeeping without specifying it further. We suspect it might be used for synchronizing RMP updates. The remaining memory in the RMP is used to hold RMP page state information for all memory (optionally a subset of the available memory). Memory includes DRAM but also MMIO, such as PCIe device BAR regions and the PCIe configuration space. Each RMP entry is 16-byte in size for a 4KiB page. Thus, the RMP incurs a 0.4% memory footprint.

Single Point of Failure. Depending on the alignment, there is one or two RMP entries where the RMP entry points to the page it resides on (e.g., the red entry in Figure 6). To find the RMP page protecting itself, we need to solve Equation 1 for x.

$$\texttt{RMP\_START} + \texttt{0x4000} + (x >> 8) = x \tag{1}$$

Table 3: Target RMP Pages of Impact for SEV-SNP. Attacks show what RMPocalypse can do by transitioning the pages into a writable state.

State	Usage	Attacks
Firmware	protects *Firmware* pages (e.g., RMP)	make RMP writable & exploit possible race conditions between PSP and hypervisor
Context	protects CVM management data	forge attestation report & Forge ASID & enable debug of a production ready CVM
Guest-Valid	protect common CVM memory from hypervisor	rollback page & overwrite with random data & inject arbitrary data
Guest VMSA	protect the register state of a CVM	rollback attack
HV-Fixed	protects MMIO & other special memory from being used for any SEV-SNP operations	-

The first four pages of the RMP are used for bookkeeping. We add `0x4000` to account for the offset. Since each RMP entry is exactly 16 bytes, we need to shift the address by 8. Pages are `0x1000` bytes in size, which equals $1 << 12$ bytes. 16 bytes equal $1 << 4$ bytes. To get the RMP index of an arbitrary page, we need to shift it by $12 - 4 = 8$. For example page `0x0000` has index `RMP_START + 0x4000`, page `0x1000` index `RMP_START+0x4000+(0x1000 >> 8)` = `RMP_START+0x4010` and so on.

We visualize the RMP entry protecting itself in red in Figure 6. After solving for x and making the entry hypervisor-owned, we can overwrite all RMP entries on the page protected by the RMP entry. Subsequently, we transition all adjacent pages around x into hypervisor writable pages. All adjacent pages are now writable. We overwrite all RMP entries on their pages and transition them from *Firmware* to *Hypervisor*. We continue inductively until the entire RMP is marked as hypervisor writable memory. For the case that x is the first or last RMP entry, we need to additionally overwrite an entry on the page before or after. In summary, by having the capability of overwriting one RMP entry, we gain the privilege of overwriting the entire RMP.

6 Using RMPocalypse

We discuss the page states of interest to RMPocalypse and introduce primitives to compromise CVMs on SEV-SNP.

6.1 Target States of Impact

After initialization, the RMP contains only 3 different states, *HV-Fixed*, *Hypervisor*, and *Firmware*. The PSP marks all RMP entries pointing to RMP memory as *Firmware* such that the hypervisor cannot write to the RMP. Further, it marks potentially dangerous memory regions, such as System Management Mode memory and all types of MMIO memory, as *HV-Fixed*. *HV-Fixed* pages may be freely written to by the hypervisor, but their RMP state cannot be altered. Lastly, *Hypervisor* pages can be promoted to SEV-SNP-related pages through RMPUPDATE and the PSP API.

As Section 5.2 discusses, we can overwrite one *Firmware* RMP entry to make the entire RMP writable. By using this primitive, we can not only overwrite the 3 pages on init, but all possible states during the RMP lifecycle. For scope, we exclude all non-active page

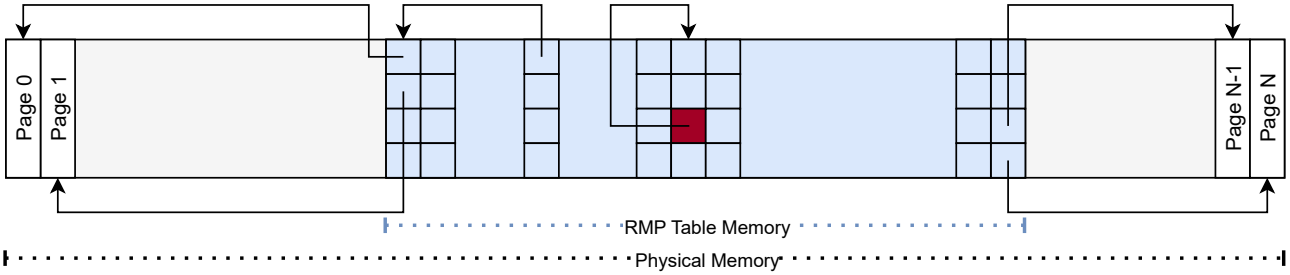

Figure 6: Memory structure of the RMP. The arrows show the page that an entry in the RMP protects. The entry marked in red protects the page it resides on. The red entry is the single point of failure of the RMP; if an attacker can overwrite the red entry, the RMP protection can be disabled.

states (e.g., *Pre-Swap*, *Pre-Guest*, *Metadata*). Table 3 summarizes the possible attacks. We discuss the most promising targets.

Overwriting *Hypervisor* Pages. We can transform *Hypervisor* pages into any other state. For instance, we could overwrite the *Hypervisor* page such that it becomes a *Guest-Valid* page with a specified GPA. However, the exploitation of this primitive is challenging. The main problem is that the content of the page will be read decrypted by the CVM. Thus, no matter which value the attacker has put into the *Hypervisor* page before transitioning it into a *Guest-Valid* page, the guest will read pseudo-random text. The same applies if we were to transition a *Hypervisor* page into any other encrypted page state. However, transitioning other page states into the *Hypervisor* state is particularly interesting since it allows the attacker to write to previously write-protected pages.

Overwriting *HV-Fixed* Pages. We may submit pages we want to mark as *HV-Fixed* to `SNP_INIT_EX`. Additionally, the SEV firmware obtains a memory map of the platform and marks security-critical pages as *HV-Fixed*. Security-critical pages include, for instance, the System Management Mode (SMM) memory region and all MMIO pages. Setting MMIO pages to *HV-Fixed* is vital for SEV-SNP security. The UMC blocks perform the encryption of the memory content. If the memory targets MMIO regions, no encryption happens. Furthermore, MMIO regions are otherwise accessible to the untrusted hypervisor (i.e., PCIe config space, PCIe device BAR region). It would trivially allow us to read and write arbitrary CVM data. Consequently, marking *HV-Fixed* pages as *Guest-Valid* pages seems to be a promising target. However, our experiments show MMIO pages marked as guest memory result in hardware-generated #VC exceptions on access. We conclude *HV-Fixed* pages are not an interesting target for RMPocalypse.

Overwriting *Firmware* Pages. The PSP marks the RMP memory as *Firmware* pages during SEV-SNP initialization. By changing the *Firmware* state to *Hypervisor* state, we make the pages writable. Transitioning all *Firmware* pages that protect the RMP to *Hypervisor* pages is the first step of RMPocalypse. Furthermore, pages that might be promoted to other states by the PSP are temporarily in the *Firmware* state, such that x86 cores cannot write to them anymore. Overwriting the page state while the PSP uses the pages may lead to race conditions. We do not explore the attack possibility further.

Overwriting *Context* Pages. *Context* pages hold all management data related to an SEV-SNP CVM. Each CVM gets its unique *Context* page. The SEV-SNP debug enable bit is part of the *Context* page (Section 7). Overwriting the bit transitions the CVM into debug mode and allows an attacker to trivially read and write CVM memory. An attacker may use RMPocalypse to make the *Context* page writable by transitioning its RMP entry into *Hypervisor* state. Furthermore, the *Context* page contains the attestation hash and the ASID (unique identifier of a CVM). We can forge the attestation report and impersonate another CVM's ASID by replaying the *Context* page. All the attacks are possible because *Context* pages are inline encrypted by the PSP AES engine, but without integrity protection (see Section 2.3).

Overwriting VMSA Pages. AMD documents do not publicly define the RMP state of VMSA pages. However, we observe their layout by profiling the VMSA page in a CVM (see Section 5.1). VMSA pages are only writable by microcode and the guest. However, the hypervisor can read and snapshot its encrypted content. RMPocalypse enables us to overwrite the VMSA pages at any point in time with arbitrary data. Writing saved VMSA page content to the actual VMSA page allows us to enter a CVM with an old register state we previously saved, effectively breaking the integrity.

Overwriting *Guest-Valid* Pages. *Guest-Valid* pages host code and data used by CVMs. We can read the encrypted content of *Guest-Valid* pages, but cannot overwrite them. By changing the RMP entry from *Guest-Valid* to *Hypervisor*, we get the capability of writing arbitrary data to the pages. We gain arbitrary code injection by combining the primitive to replay the data with the SEV-SNP ABI `SNP_PAGE_MOVE` and an I/O interface of the CVM. We discuss this attack vector in detail in Section 6.2.

Remaining Page States. The remaining page states may be used to compromise SEV-SNP security. However, they are not active pages (i.e., not actively used by the PSP hypervisor or the Guest) and are thus used to temporarily ensure some invariants (i.e., block x86 core access such that the SEV firmware can securely operate on a page). Most attacks unique to non-active page states include split-view attacks where the PSP thinks it has exclusive access, but the hypervisor/guest can also write to it. Race conditions induced by split view attacks are challenging to exploit, and finding suitable cases requires a detailed analysis.

6.2 Primitives

RMPocalypse can be used to corrupt many targets. We pick the following four to show the richness of our attack.

Enabling Debug Mode. SEV-SNP allows debugging of CVMs. The hypervisor may enable debug mode by setting a bit during guest creation. Since it is part of attestation, an attacker cannot create debug-enabled CVMs and trick the victim into believing it is a secure CVM. The PSP stores the debug-enable bit in the *Context* page belonging to the guest. Production CVMs do not have the debug-enable bit set. Our goal is to toggle the bit and enable debug mode. The PSP inline encrypts the *Context* page with its private key. The RMP protects the *Context* page from malicious overwrites from the x86 cores. RMPocalypse allows us to remove the write protection of *Context* pages by transitioning them to *Hypervisor* pages. We use this primitive to overwrite the ciphertext of the *Context* page at the 16-byte block hosting the debug-enable bit. Since we have no control over the plaintext when the PSP decrypts the *Context* page, we need to get lucky to cause the bitflip. AES decryption of a random ciphertext behaves like a pseudo-random function. Subsequently, by changing the ciphertext, we have a 50% chance of flipping the debug-enable bit. By repeating the process, we achieve a 100% success rate. Once we flip the debug-enable bit, we use the SEV-API functions `SNP_DEBUG_DECRYPT` to leak the entire guest memory and `SNP_DEBUG_ENCRYPT` to write arbitrary data into guest memory. The victim has no possibility of detecting the attack since the attestation report states that the debug-enable bit is not set. Further, we can always reset the Guest *Context* page to its original value after executing our attack.

Attestation Forgery. Breaking the attestation mechanism renders confidential computing useless. We use RMPocalypse to forge an attestation report and corrupt the initial CVM state. The PSP stores the attestation report encrypted in a *Context* page belonging to the CVM. The PSP decrypts the *Context* page but does not check its integrity. The lack of integrity enables us to perform a ciphertext rollback attack at block granularity. Fortunately, the PSP freshly generates the *Context* page encryption key once during every platform boot. The static key allows us to perform a replay attack between two *Context* pages. We create a benign CVM and snapshot its ciphertext attestation value in the *Context* page. Subsequently, we create a malicious CVM and replay the ciphertext attestation value we previously obtained. If the victim connects to the newly created malicious CVM and queries the attestation value from the PSP, it gets the benign attestation report, despite the loaded image being malicious.

VMSA State Rollback. The state loaded into CPU registers on `VMRUN` is stored in the VMSA. By saving the VMSA and rewriting it to the VMSA page at a later time, we reenter the CVM with the saved register state. We can stop the CVM precisely by using page-fault in combination with SEV-Step [38]. After we interrupt the CVM, we save its VMSA page. While the CVM is not running, we can transition the VMSA page to *Hypervisor* and write the saved VMSA to the VMSA page. The next `VMRUN` instruction using the VMSA page will enter the CVM, but at the old saved state.

Arbitrary Code Injection. We inject arbitrary code into the victim CVM by only manipulating *Guest-Valid* pages. Every CVM needs an interface to communicate with the outside world. We use this I/O interface to inject arbitrary attacker controlled data into the CVM [30]. On SEV-SNP, we cannot replay ciphertext from physical page A to page B because different pages are encrypted with different tweak values. This ensures that even if page A and page B have the same plaintext, they encrypt to different ciphertexts. Unfortunately, if we try to use RMPocalypse to replay the ciphertext from page A to page B, this will lead to pseudo-random data. For successfully overwriting page B with page A's plaintext, we must first ensure the content of page A is encrypted with the tweak value of page B. The SEV-SNP API `SNP_MOVE_PAGE` gives us this primitive. We move the content of page B to a temporary page, move the content of page A to page B and back, and lastly, restore the original content of page B from the temporary page. Furthermore, we snapshot the ciphertext of page A's content on page B. The ciphertext corresponds to page A's content encrypted with the tweak value of page B. Until this point, we did not use any RMPocalypse primitive. The actual attack is to use RMPocalypse to remove the write protection of page B and effectively replay page A's ciphertext we just snapshotted. Since page A contains attacker-controlled data, we effectively overwrite page B with arbitrary data. In our attack, page B contains kernel code, and we use the I/O channel to inject arbitrary code. Alternatively, we can execute the same attack using PSP APIs exposed in debug mode.

7 Implementation & Evaluation

We perform our experiments on an AMD EPYC 9135 16-core processor with 32 GiB RAM. The Zen 5 processor runs the latest microcode `0xb002116` and PSP firmware version `1.55` build 44 as of 3rd February 2025. We use Linux `6.12` as a hypervisor and guest kernel for our exploits. For SEV-SNP initialization, we use the SNP ABI command `SNP_INIT_EX`. We set the additional flags `INIT_RMP` and `LIST_PADDR_EN` as additional arguments for `SNP_INIT_EX`.

RMP Overwrite. We change the Linux function `__sev_snp_init_locked` to omit the first wbinvd call on all CPUs. Furthermore, we add our page table mapping code and create a kernel thread, `execute_attack`, to overwrite the RMP. These changes account for 180 LoC changes. `SNP_INIT_EX` takes 234 ms to execute. Linux is occasionally unable to recover from the RMP page fault during initialization. The RMP page faults happen because our kernel thread constantly writes to RMP memory, which will eventually become read-only, causing the fault. We add recovery logic within the page fault handler to mitigate the RMP-induced crashes. We change the RIP to point to the label `exit_jump` within the Listings in Section 4.1. This ensures the stack frame is properly recovered upon exit, and we are not crashing the kernel.

7.1 Enabling Debug

We enable debug mode on fully attested production CVMs. The debug mode allows us to stealthily exfiltrate and inject arbitrary data. The hypervisor may enable the debug mode of a CVM by setting a bit during creation (or disable debug by omitting to set it). Eventually, the PSP writes the debug-enable bit into the *Context* page as part of the `snp_policy` field. The `snp_policy` is part of the attestation report. The guest checks the attestation report after the CVM has been initialized. However, RMPocalypse changes the

snp_policy after the guest attests the CVM, and thus, the change remains undetected by the guest.

Finding the Offset. The exact offset of the debug-enable bit within the management structure in the PSP is undocumented. The *Context* page offsets in the SEV-SNP specification are outdated [2]. Extracting the location by replicating the struct guest_context_page does not work reliably since not all struct fields are properly defined (i.e., some fields are missing a size, and we do not know which compiler flags have been used to compile the SEV firmware code). Furthermore, the open-source SEV firmware code is for Zen 4 and is almost 2 years old. In the meantime, the code has been updated 19 times (FW version 1.55.25 vs. 1.55.44). We extract the offset of the snp_policy field dynamically. To find the location, we snapshot the *Context* page of a guest under our control at time t. Subsequently, we send a guest message to the PSP. The PSP increments the msg_count0 that directly follows the snp_policy. We dump the guest context page again and compare it to the one at timestamp t. We observe a change in the encrypted AES block starting at offset 0x200. Since msg_count0 is naturally aligned, we assume snp_policy is either in the same 16-byte block or in the previous one.

Experiments. To test our hypothesis, we use our RMP overwrite primitive to make the *Context* page writable and increment the ciphertext at offset 0x200 / 0x19F by 1. Incrementing offset 0x200 does not affect the debug state. However, after incrementing offset 0x19F, we observe a debug-enable bit flip. We monitor the flip by executing SNP_DBG_DECRYPT and observing the error code. Error code 0x7 indicates a policy violation (disabled debug mode), and code 0x0 shows success. The PSP inline decryption of the modified *Context* page ciphertext results in a pseudo-random number. A single overwrite has a 50% probability of flipping the debug-enable bit in snp_policy. After 10 tries, we have an average success probability of $1 - \frac{1}{2^{10}} = 99.9\%$. Changing the ciphertext and validating if the bit has flipped costs between 699μs to 905μs. The lower end is if the PSP does not perform the actual decryption and aborts due to a policy violation, and the upper end is if the PSP decrypts the page and our attack succeeds.

```
while(fail){
    tmp_value = __rmptable[gctx_paddr >> 12].lo;
     __rmptable[gctx_paddr >> 12].lo = 0x4;
     gctx_virt[0x1FF]++;
    __rmptable[gctx_paddr >> 12].lo = tmp_value;
    fail = do_debug_decrypt(gctx_paddr,hpa_p1,0x1000);
}
```

Listing 5: Flipping the debug-enable bit in the *Context* page.

Listing 5 shows our kernel module code to flip the bit. We change the RMP entry of the *Context* page twice. First, to make the *Context* page writable by the hypervisor and second, to pass the correct page state to SNP_DBG_DECRYPT. The end-to-end attack with a 99.9% success probability takes at most 7.2 ms. For a 99.9999% success probability, the attack needs at most 14.2 ms. We implement our attack in a Linux kernel module in 496 LoC.

7.2 Attestation Forgery

The guest measurement hash is 48 bytes long and stored in the *Context* page. The PSP calculates the hash over the initial state of a CVM. We obtain the necessary *Context* page offset of measurement from prior work [12]. We snapshot a 64-byte block starting from offset 0x460. Due to the non-16-byte alignment, the snapshot includes a larger 0x40 byte memory area. In the offline phase, we boot the benign image using the correct configuration parameters. We capture the encrypted measurement from the *Context* page once the PSP finishes CVM initialization. In the online phase, we boot a malicious image (e.g., a slightly modified image with a backdoor). The PSP calculates the measurement hash over the malicious image. When the guest requests an attestation report, we overwrite new *Context* pages measurement with the hash of the benign image. The guest requesting an attestation report requires it to forward the request through the hypervisor. Thus, the hypervisor can change the measurement value before making the call to the PSP to request the attestation value. To ensure the attack works, we modify the Linux kernel to use the same physical address of the *Context* page for both boots. The guest in the malicious image receives the attestation report for the benign image and assumes the environment is set up correctly. Appendix A provides a detailed explanation of AMD SEV-SNP attestation procedure.

7.3 VMSA State Rollback

We snapshot the vCPU's VMSA in the svm_vcpu_run function. After a few seconds, we intercept the CVM execution in the same function and replay the saved VMSA. We replay the saved VMSA page by transitioning the active VMSA page to *Hypervisor*, writing the saved VMSA to the active page, and transitioning it back into VMSA page state. The register state of the CVM has been successfully reset. We experiment with a monotonously incrementing counter and observe a counter rollback if we overwrite the register state. An attacker may use the privilege to snapshot arbitrary states and reenter the CVM with a chosen snapshot.

7.4 Arbitrary Code Injection

We perform arbitrary code injection in three steps:

1. Breaking Physical KASLR. We use techniques from previous works to break physical KASLR [30, 34, 37]. The last address of deterministic execution is page 0x4fb6000. The page fault following that page is the startup_64 function and marks the beginning of the vmlinux image.

2. Getting Malicious Data Into the Kernel. We use a Python script to craft a network packet with a data payload. We observe Linux copying the packet into CVM memory using virtio queues. Eventually, Linux converts the packet into an skb struct in the Linux kernel, and our payload resides on a 4KiB page.

3. Swapping Pages. We implement the SNP_PAGE_MOVE API call since Linux does not have a wrapper at the time of writing. Our implementation consists of 637 LoC. Swapping one page takes 1.6 ms. Swapping 3 pages, changing the guest page tables, and writing the shellcode takes 5.03 ms.

We leak the guest's physical address of the network packet to locate the injected code. This allows us to easily swap the pages and perform the replay attack. In an end-to-end attack, an attacker has to dynamically find the page. Prior works show the feasibility of finding dynamic pages in SEV-SNP CVMs [23, 33, 34, 37].

8 RMP Lifecycle Management

Similarly to the initialization, we also analyze the security of the RMP lifecycle from the x86 and PSP perspectives.

8.1 RMPUPDATE

RMPUPDATE exposes a restricted interface to the hypervisor to update RMP entries. It performs various microcode checks to ensure secure lifecycle management of the RMP. TLB entries cache the RMP permissions. Thus, after an update to the RMP, RMPUPDATE must flush TLB entries resolving to the page whose RMP entry has been updated. Our experiments confirm that after executing an RMPUPDATE, previous TLB entries resolving to the page do not exist anymore. Listing 6 depicts the experiment we perform to validate the flushing operation with 4KiB as well as 2MiB pages.

```
u64 *ptr = 0xfffffffffff00000;
map_phys_to_virt(0x3000000,0xfffffffffff00000)
WRITE_ONCE(ptr[0]); /* create TLB entry */
rmpupdate_firmware(ptr);
WRITE_ONCE(ptr[0]);
```

Listing 6: RMPUPDATE TLB flushing check.

SEV-SNP disallows the existence of 1GiB TLB entries, as this potentially undermines SEV-SNP security since the RMP operates only on 4KiB / 2MiB entries. Translations from 1GiB pages only insert 2MiB entries for the respective submapping [7]. An insertion of a 1GiB TLB must first check the permissions of all 512 2MiB subpages, and if any of those restrict write access (e.g., any of the 2MiB pages contain a *Firmware* page), the 1GiB entry would only allow reads. AMD decided to skip this case and only insert 2MiB entries for simplicity.

To further test if we can circumvent the TLB-based protection checks, we use the Sinkclose vulnerability to gain System Management Mode (SMM) code access [31]. When entering SMM, the processor executes in real mode. Real mode directly accesses physical memory and does not use virtual addresses. We suspect that by circumventing the page walk, we can bypass the RMP protection checks. However, when writing physical memory protected by the RMP directly, we encounter a system freeze with a reset. We suspect x86 cores have additional protections besides the TLB to enforce SEV-SNP memory access checks. The freeze we are seeing is most likely the result of an unhandled page fault, which leads to a double and then a triple fault.

To update the RMP, RMPUPDATE must read and write the RMP entry. To prevent x86 and the PSP from accessing and modifying the same data, AMD introduced the SNP state machine [8]. Each page state may only be modified by the PSP or by x86 at any given point in time. There is no page state that the x86 core and PSP can update simultaneously. The PSP can exclusively transform pages into a different state with the immutable bit set (Table 2), which prevents x86 and the PSP from concurrently updating the same RMP entry in memory.

However, x86 and PSP may race to update a sub-entry in a 2MiB range and the 2MiB RMP entry. While x86 accesses the RMP, the PSP may access the same 2MiB region concurrently. As the first 2MiB page contains metadata about the 512 4KiB mappings within, care must be taken when updating it, as x86 might access other pages concurrently. To prevent such races, the PSP SEV firmware source code shows how the PSP ensures that the PSP and x86 cannot race in performing the update to the 2MiB RMP region. AMD uses a special invalid ID, `0x3ff`, to lock the 2MiB range to prevent microcode access [6].

Lastly, we study how RMPUPDATE accesses the memory and if we can break the coherency of the RMPUPDATE instruction itself. We use AMD IBS to trace the micro-ops of RMPUPDATE and discover that RMPUPDATE accesses the RMP entry using its physical address. Using the physical address directly means we cannot use page tables to influence the caching behavior of the access. From the recent Entrysign discovery, we know that the CPU microcode has multiple addressing modes [18]. We suspect that the RMPUPDATE microcode uses one of the addressing modes that directly accesses physical memory. Further, we use the Sinkhole technique to overlap the APIC register with the range of RMPUPDATE memory accesses, to force the RMPUPDATE instruction to read MMIO memory instead of DRAM [13]. However, despite the overlap, RMPUPDATE accesses DRAM and updates the RMP entry correctly. This further strengthens the hypothesis that microcode uses a special read and write mode to access the RMP.

8.2 PSP RMP Updates

The PSP is an Arm Cortex A5 processor with dedicated data and instruction caches. It caches accessed data, even if it comes from x86 DRAM. CS blocks do not snoop the caches of the PSP; therefore, the PSP must ensure coherency within its software. To resolve the issue, the PSP initiates a software cache flush after writing to the RMP to evict dirty cachelines. This resolves two issues: first, it ensures that the PSP always reads the latest data from DRAM, and second, it ensures x86 cores always get the latest data when reading RMP memory.

9 Discussion

RMPocalypse overwrites the RMP during initialization. From our analysis of AMD SEV firmware source code, we conclude that AMD enables mechanisms of protecting the RMP during initialization, but they are insufficient. The TMR protection (see Section 3.2.2) blocks accesses, but it is too late and allows stale cache entries to overwrite protected RMP memory. The PSP operating system is closed source and operates on MMIO memory to execute platform configuration commands, thus making binary analysis infeasible. Thus, we are unable to validate or falsify any hypothesis. No matter what the cause, we are able to corrupt the RMP from x86 cores. So we propose defenses to stop this behavior.

We propose two ways to mitigate the underlying vulnerability and suggest one firmware defense to make exploitation difficult. The underlying issue originates from the positioning of the RMP memory protection at the CS blocks. An activated RMP system performs the RMP checks at a core level before reaching the cache hierarchy. Thus, when switching from TMR protection to RMP semantics, all data that is in the caches at that time is not subject to either checks.

Mitigation 1: Aligning the Barriers. To mitigate the attack, the protection mechanism during initialization and runtime (RMP checks) should be in the same position on the platform. Thus, the

initialization barrier should be installed at a core level, also protecting the caches, and not at the memory controller in the form of TMRs. Moving the RMP checks on the memory controller is infeasible, since data in the caches must be subject to RMP checks. It remains to be seen if the affected AMD hardware can enforce such checks at the core boundary. Such checks would likely cause changes to the microcode as well as the PSP. Depending on the hardware mechanism implementing such a barrier, special care must be taken when reasoning about the security. The PSP must also flush the TLBs of the x86 cores if the RMP overwrite protection during initialization is TLB-based (like the RMP checks). Further, all instructions directly accessing physical memory must be patched to not access RMP memory during initialization (e.g., `VMSAVE`).

Zen 3 Considerations. Zen 3 does not ensure cache coherence between encrypted and unencrypted memory. Thus, when deploying a mitigation, AMD must make sure to pay close attention to Zen 3. PSP writes to unencrypted RMP memory do not invalidate dirty cacheline entries of encrypted RMP memory within the x86 caches. Thus, if the x86 core can hold those entries dirty in the caches for the duration of the initialization, it can flush the cache entries after the TMRs and the barrier on the x86 core lifts. To prevent the attack, the PSP must invalidate all cache entries in the encrypted domain pointing to RMP memory. This can be done by setting a special platform bit and reading the memory from the PSP [6].

Mitigation 2: Flushing the Caches. Instead of moving the initialization barrier, the PSP could force all x86 cache entries to be flushed to the memory controller after activating RMP checks. At that time, the TMR barrier must still be in place, such that the TMR blocks all dirty cache entries pointing to RMP memory. This would be the easiest fix, as it simply requires the PSP to flush the caches. However, the current PSP source code never initiates an x86 cache flush and always implicitly requests the x86 cores to execute `wbinvd` and validates its execution on each core. We suspect that the PSP most likely does not have the capability of flushing the x86 caches, and thus, implementing our proposed solution would require API changes to the PSP initialization. Patching both the hypervisor and the PSP, and requiring version alignment, makes implementing this defense challenging.

Our proposed solution only works if the x86 cores perform RMP access checks by accessing the RMP directly in memory and ignoring stale data in the L1/L2/L3 cache. As RMPUPDATE directly accesses physical RMP memory, we assume the x86 cores do the same when accessing the RMP entry to validate the permissions of the memory request. Otherwise, the proposed defense is susceptible to at least one race condition. While the PSP is flushing the entire cache, an x86 core could insert TLB entries that allow writes to the RMP memory range simply by reading or writing the RMP memory. As the TMR is still in place, such access would be only in the caches and will not propagate to physical DRAM. After flushing the cache, the PSP would SEV-SNP globally, flush the TLBs, and disable the TMRs in a last step. Since flushing the caches and TLBs does not happen atomically, this opens up a race condition for creating stale TLB entries that could overwrite the RMP after initialization. If the PSP first flushes the caches and then the TLBs, an x86 core could use a stale TLB entry to create another dirty cache entry pointing to the RMP. If the PSP does it the other way around, the x86 core could use the window to create another stale TLB entry, as the RMP checks of the x86 core use the stale cache entries for their checks.

Firmware Defense. Our proof-of-concept attacks heavily rely on a writable RMP at runtime. However, there exists no benign use case where an RMP entry protecting the RMP is in a state other than firmware. Thus, the PSP could check the RMP entries protecting the RMP on every x86 API call. While this mitigation can be circumvented since it is vulnerable to TOCTOU attacks, it will likely make the exploitation harder. The processor caches RMP permissions in the TLB. A core with stale TLB entries can overwrite the RMP entry despite the RMP in memory indicating that write access is disallowed. To prevent this, the PSP must flush the TLB of all cores, but this is racy as flushing the TLBs and reading the RMP entry is not atomic. Further, flushing the TLB globally after every RMP entry read would cause severe overhead. While not stopping our attack, it makes exploitation more challenging. Periodic RMP checks were suggested by Google Researchers in a preliminary assessment of the SEV-SNP [16].

10 Related Work

We discuss attacks on AMD SEV variants (SEV, SEV-ES, SEV-SNP).

Attacks against SEV and SEV-ES. Several attacks exploit AMD's initial SEV design either with page remapping attacks or by directly modifying or observing the VMSA of the guest CVM [20, 29]. Many of the SEV page remapping attacks also apply to SEV-ES [29, 37]. Notably, Morbitzer et al. achieve arbitrary code injection without relying on I/O operations [37]. Crossline exploits weak ASID bindings to exploit SEV and SEV-ES [24]. Radev exploit untrusted interfaces to fully compromise SEV and SEV-ES et al. [32]. Hetzelt analyze device interfaces of CVMs and use them to exploit implementation vulnerabilities et al. [21]. Cipherleaks exploits the deterministic encryption of the VMSA to leak information about executing programs in the VM [25]. While the proof-of-concept was done on SEV-ES due to the unavailability of SEV-SNP machines at the time of writing, AMD notes that SEV-SNP is also affected. PwrLeaks abuses power side channels to infer instruction information about CVMs [36]. Li et al. uses TLB Poisoning to break SEV-ES [26]. Buhren uses a power glitch attack against the PSP to extract the root keys, rendering key-based attestation useless et al. [9]. All of these attacks were on the predecessors of SEV-SNP, which were considered insecure in the confidential computing threat model. Since RMPocalypse exploits the RMP to break SEV-SNP, our attack does not apply to SEV and SEV-ES.

Attacks against SEV-SNP. A pre-release security report by Google Project Zero uncovered multiple severe flaws in SEV-SNP architecture [16]. CacheWarp exploits an unlocked MSR, allowing the untrusted hypervisor to execute the `INVD` instruction [41]. It builds primitives to rollback caches and compromise userspace applications. Heckler and WeSee exploit malicious interrupt injection in CVMs [33, 34]. Heckler uses the `Int0x80` interrupt to corrupt the state of userspace applications. WeSee uses the `#VC` interrupt to attack the Linux kernel running in the guest CVM to gain arbitrary code execution. CounterSEVeillance uses performance counters to observe information inside the CVM (e.g., branches) and recover secret key values [14]. BadRAM exploits memory aliasing of the DRAM chips [12]. DRAM aliasing allows BADRAM to bypass RMP

protections and circumvent SEV-SNP security guarantees. Furthermore, there have been multiple ciphertext side-channel and cache side channel attacks on SEV-SNP [11, 15, 23, 35, 39, 40]. RMPocalypse is the first work that examines the RMP lifecycle and shows critical security gaps.

11 Conclusion

We present a simple yet elegant and powerful attack called RMPocalypse, that exploits the lack of proper RMP protection during initialization on AMD SEV-SNP. We hope that our analysis of AMD SEV-SNP from a Data Fabric viewpoint helps other researchers to further assess its security from a different perspective. Finally, we thank AMD for open-sourcing the SEV firmware code and encourage them to continue in that direction.

References

[1] Amazon. accessed 2025-09-10. AWS Nitro Enclaves - Create additional isolation to further protect highly sensitive data within EC2 instances. https://aws.amazon.com/ec2/nitro/nitro-enclaves/.

[2] AMD. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity protection and more. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[3] AMD. 2020. High Performance Computing (HPC) Tuning Guide for AMD EPYC™ 7002 Series Processors. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/amd-epyc-7002-tg-hpc-56827.pdf.

[4] AMD. 2021. Probe filter directory management (US12141066B2). https://patents.google.com/patent/US12141066B2.

[5] AMD. 2023. 58015: AMD EPYC 9004 Series Architecture Overview. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/white-papers/58015-epyc-9004-tg-architecture-overview.pdf.

[6] AMD. 2023. AMD-ASPFW. https://github.com/benschlueter/AMD-ASPFW/blob/3ca6650dd35d878b3fcbe5c7f58b145eed042bbf/.

[7] AMD. 2023. AMD64 Architecture Programmer's Manual Volumes 1–5, Rev. 4.08 (40332). https://docs.amd.com/v/u/en-US/40332-PUB_4.08.

[8] AMD. 2025. SEV Secure Nested Paging Firmware ABI Specification, Rev 1.58. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf.

[9] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2875–2889. https://doi.org/10.1145/3460120.3484779

[10] Thomas Burd, Noah Beck, Sean White, Milam Paraschou, Nathan Kalyanasundharam, Gregg Donley, Alan Smith, Larry Hewitt, and Samuel Naffziger. 2019. "Zeppelin": An SoC for Multichip Architectures. *IEEE Journal of Solid-State Circuits* 54, 1 (2019), 133–143. https://doi.org/10.1109/JSSC.2018.2873584

[11] Li-Chung Chiang and Shih-Wei Li. 2025. Reload+Reload: Exploiting Cache and Memory Contention Side Channel on AMD SEV. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, New York, NY, USA, 1014–1027. https://doi.org/10.1145/3676641.3716017

[12] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. 2025. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In *2025 IEEE Symposium on Security and Privacy (SP)*. 4117–4135. https://doi.org/10.1109/SP61157.2025.00104

[13] Christopher Domas. 2015. The Memory Sinkhole. In *Blackhat USA*. Blackhat.

[14] 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: NDSS 2025*.

[15] Lukas Giner, {Sudheendra Raghav} Neela, and Daniel Gruss. 2025. Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP. In *DIMVA* (22 ed.). 22nd International Conference on Detection of Intrusions and Malware, and Vulnerability Assess, DIMVA 2025 ; Conference date: 09-07-2025 Through 11-07-2025.

[16] Google. 2022. AMD Secure Processor for Confidential Computing. https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/AMD_GPZ-Technical_Report_FINAL_05_2022.pdf.

[17] Google. 2025. Driving enterprise transformation with new compute innovations and offerings. https://cloud.google.com/blog/products/compute/delivering-new-compute-innovations-and-offerings.

[18] Google. 2025. Zen and the Art of Microcode Hacking. https://bughunters.google.com/blog/5424842357473280/zen-and-the-art-of-microcode-hacking.

[19] Google. accessed 2025-09-10. Confidential Computing | Google Cloud. https://cloud.google.com/confidential-computing.

[20] Felicitas Hetzelt and Robert Buhren. 2017. Security Analysis of Encrypted Virtual Machines. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Xi'an, China) *(VEE '17)*. Association for Computing Machinery, New York, NY, USA, 129–142. https://doi.org/10.1145/3050748.3050763

[21] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. 2021. VIA: Analyzing Device Interfaces of Protected Virtual Machines. In *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. ACM, 273–284. https://doi.org/10.1145/3485832.3488011

[22] David Kaplan. 2017. PROTECTING VM REGISTER STATE WITH SEV-ES.

[23] 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. https://doi.org/10.1109/SP46214.2022.9833768

[24] 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) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2937–2950. https://doi.org/10.1145/3460120.3485253

[25] 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. https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan

[26] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *Proceedings of the 37th Annual Computer Security Applications Conference* (Virtual Event, USA) *(ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 609–619. https://doi.org/10.1145/3485832.3485876

[27] Meta. accessed 2025-09-10. Building Private Processing for AI tools on WhatsApp. https://engineering.fb.com/2025/04/29/security/whatsapp-private-processing-ai-tools/.

[28] Microsoft. accessed 2025-09-10. Azure confidential Cloud - Protect Data In Use | Microsoft Azure. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[29] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security* (Porto, Portugal) *(EuroSec'18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3193111.3193112

[30] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. 2021. SEVerity: Code Injection Attacks against Encrypted Virtual Machines. In *2021 IEEE Security and Privacy Workshops (SPW)*. 444–455. https://doi.org/10.1109/SPW53761.2021.00063

[31] Enrique Nissim and Krzysztof Okupski. 2024. AMD Sinkclose: Universal Ring-2 Privilege Escalation. Presented at DEF CON 32. https://www.ioactive.com/event/def-con-talk-amd-sinkclose-universal-ring-2-privilege-escalation/.

[32] Martin Radev and Mathias Morbitzer. 2021. Exploiting Interfaces of Secure Encrypted Virtual Machines. In *Reversing and Offensive-Oriented Trends Symposium* (Vienna, Austria) *(ROOTS'20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3433667.3433668

[33] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. Heckler: Breaking Confidential VMs with Malicious Interrupts. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3459–3476. https://www.usenix.org/conference/usenixsecurity24/presentation/schlüter

[34] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. 2024. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 4220–4238. https://doi.org/10.1109/SP54263.2024.00262

[35] Benedict Schlüter, Christoph Wech, and Shweta Shinde. 2025. Heracles: Chosen Plaintext Attack on AMD SEV-SNP. In *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3719027.3765209

[36] 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: 20th International Conference, DIMVA 2023, Hamburg, Germany, July 12–14, 2023, Proceedings* (Hamburg, Germany). Springer-Verlag, Berlin, Heidelberg, 46–66. https://doi.org/10.1007/978-3-031-35504-2_3

[37] 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 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1483–1496. https://doi.org/10.1109/SP40000.2020.00080

[38] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. 2023. SEV-Step A Single-Stepping Framework for AMD-SEV. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 1 (Dec. 2023), 180–206. https://doi.org/10.46586/tches.v2024.i1.180-206

[39] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. 2024. HyperTheft: Thieving Model Weights from TEE-Shielded Neural Networks via Ciphertext Side Channels. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4346–4360. https://doi.org/10.1145/3658644.3690317

[40] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. 2025. CipherSteal: Stealing Input Data from TEE-Shielded Neural Networks with Ciphertext Side Channels . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 79–79. https://doi.org/10.1109/SP61157.2025.00079

[41] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. CacheWarp: Software-based Fault Injection using Selective State Reset. , 1135–1151 pages. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-ruiyi

A Attestation

There are two means for a guest to attest to the running image the hypervisor creates.

(1) ID Block attestation through `SNP_LAUNCH_FINISH`
(2) Runtime attestation through `SNP_GUEST_REQUEST`

A.1 ID Block Attestation

In addition to the image to boot, the guest supplies the hypervisor with an `ID_Block`. The `ID_Block` contains the expected 48-byte measurement of the guest, as well other other guest parameters (e.g., the guest policy). When the hypervisor is about to activate an SEV-SNP guest, it can optionally set the `ID_BLOCK_EN` as an argument to `SNP_LAUNCH_FINISH`. Doing so results in the PSP expecting an additional physical address as an argument, pointing to the `ID_Block`. Upon execution of `SNP_LAUNCH_FINISH`, the PSP checks that the computed hash matches the user-supplied hash in the `ID_Block`.

An attacker uses RMPocalypse before calling `SNP_LAUNCH_FINISH`, to overwrite the measurement hash stored in the *Context* page to match the expected hash within the `ID_Block`.

A.2 Runtime Attestation

A guest can dynamically request an attestation report from the PSP during runtime. The flow follows the standard hypercall procedure. The guest sets the hypercall information to `VMGEXIT_GUEST_REQUEST` and the request to the PSP to `MSG_REPORT_REQ`. The hypercall transfers control to the untrusted hypervisor. The hypervisor cannot determine which subcall the guest wants the PSP to perform, as the payload is encrypted with a key only accessible to the guest and the PSP. Thus, every `VMGEXIT_GUEST_REQUEST` may request the attestation report. The hypervisor forwards the encrypted request to the PSP for handling by calling `SNP_GUEST_REQUEST`. Once the PSP finishes the execution, the untrusted hypervisor copies the result back into guest shared memory and notifies the guest about completion. The guest decrypts the payload and checks if the attestation report matches the expected result.

A malicious hypervisor may use RMPocalypse any time before calling `SNP_GUEST_REQUEST` on the PSP, to overwrite the *measurement* value in the *Context* page.