Mior: Breaking AMD SEV-SNP with Device-as-Memory

Chris Wech
chris.wech@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Philipp Giersfeld
philipp.giersfeld@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

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

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

Abstract

Confidential computing aims to protect tenants in cloud computing settings against malicious privileged software controlled by the cloud provider. AMD SEV-SNP is one approach to enable confidential computing, by providing guest virtual machines with confidentiality and integrity of data in use. AMD uses an on-SoC hardware component called *Platform Security Processor* (PSP) to implement parts of SEV-SNP.

We present a new attack Mior, to show that a malicious software adversary with hypervisor privileges (e.g., cloud provider) can misconfigure the platform to change the memory view of the PSP. Specifically, Mior coerces the PSP into reading arbitrary data in our control, and redirects PSP writes to our attacker-controlled destination. In other words, Mior creates an attacker-controlled memory for the PSP. We refer to this primitive as *Device-as-Memory*. To showcase the impact of Mior, we use the Device-as-Memory primitive to corrupt PSP writes during the SEV-SNP lifecycle (e.g., initialization, attestation), resulting in violation of all security guarantees afforded by SEV-SNP.

CCS Concepts

• **Security and privacy → Hardware security implementation**; **Hardware attacks and countermeasures**.

Keywords

SEV-SNP, confidential computing, trusted execution environments, virtualization, computer architecture

ACM Reference Format:
Chris Wech, Philipp Giersfeld, Benedict Schlüter, and Shweta Shinde. 2026. Mior: Breaking AMD SEV-SNP with Device-as-Memory. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3830454.3846723

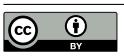
1 Introduction

Confidential computing creates an abstraction called *Confidential Virtual Machines (CVMs)*, such that privileged software components controlled by the cloud provider cannot access any CVM code or data during its execution [1, 9, 21]. AMD offers confidential computing on its EPYC line of SoCs, in the form of Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP). A dedicated hardware security co-processor, called the Platform Security Processor (PSP), bootstraps the SEV-SNP environment and ensures runtime guarantees such as memory isolation.

The PSP is an Arm core that is placed on the EPYC SoC alongside the high-performance x86 cores and several other components (e.g., DRAM controller, cache coherency controllers). AMD's interconnect, called Infinity Fabric, acts as the central nervous system of the SoC [8]. It consists of a sub-component known as the Data Fabric that routes all memory transactions, which can be memory-mapped I/O (MMIO) or DRAM [2]. The Data Fabric is highly configurable to allow component-specific routing (e.g., when an address access originates from a core, it gets routed to DRAM; but the same address access from the PSP can be routed to MMIO). Similarly, when external accelerators are attached to the SoC (e.g., GPUs, NICs, and TPUs), the Data Fabric can configure their MMIO accesses as well.

For the security of SEV-SNP, it is critical that the Data Fabric routes PSP accesses correctly to DRAM. For example, when the PSP sets up data structures that are used for access-control to CVM memory (e.g., Reverse Map Table or RMP), these writes should be routed correctly to DRAM. If an attacker can manipulate the Data Fabric routing to divert the PSP writes to an MMIO region instead of DRAM, the data structures in DRAM will remain uninitialized, thereby holding default unsafe values set up by the untrusted hypervisor. This defeats the access-control checks, which in turn will break SEV-SNP isolation. The onus of ensuring the correctness of the Data Fabric configuration and routing is on the PSP.

In this paper, we make four main observations. First, we find that during SEV-SNP initialization, the PSP first checks for a locked Data Fabric such that the adversary can no longer change it. Then the PSP reads the Data Fabric configuration and computes a memory map that captures which access will go to DRAM vs MMIO based on the addresses. The PSP consults this memory map before making any DRAM accesses itself to detect any runtime misrouting that deviates from the initial Data Fabric setup. Second, we find that the PSP does not compute the memory map correctly, which allows misrouting to go undetected. Third, we find that the hypervisor can configure the routing of individual components before the PSP locks the Data Fabric to construct the memory map. Thus, a malicious hypervisor can intentionally configure Data Fabric components with divergent MMIO routing configurations to corrupt the memory map that the



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846723

PSP will construct. Then, during the actual routing on the Data Fabric the malicious configuration by the hypervisor will lead to MMIO access. In other words, the attacker can bypass the locking and the checks to change PSP DRAM accesses to MMIO.

A natural use of this capability is to target the PSP writes that set up the data structure called the Reverse Map Table (RMP). By misrouting these PSP writes, say by simply dropping them, we can potentially break SEV-SNP. When we perform this dropping attack, our exploit fails, and the PSP refuses to complete SEV-SNP initialization. Our fourth finding shows that the PSP reads back the RMP after the setup (i.e., writes) to check if the writes were done correctly. This *read-back check* thwarts our exploit. To circumvent this check, we have to trick the PSP into believing the write succeeded. This means, when the PSP reads back the values, Data Fabric should return the values PSP wrote. Note that these values need not come from the DRAM for the *read-back check* to pass the check. As long as we can capture the writes from the PSP and supply the values back to the PSP when it reads them, we will be able to trick the PSP into believing that the RMP was initialized correctly. In reality, however, these writes never reached DRAM, thus leaving the RMP uninitialized. As a consequence, the CPU then uses the unsafe default values when it attempts to use the RMP to enforce CVM isolation.

Putting together our findings, we build a primitive we call *Device-as-Memory*, wherein all the PSP reads/writes are backed by an attacker-controlled memory instead of the actual DRAM. Our insight is to route the PSP reads/writes to an attacker-controlled device. This way, the device can both remember the values that the PSP wrote and provide them when the PSP does a read to check the values. Specifically, we set up an attacker-controlled device and configure the Data Fabric such that the PSP reads/writes go to the MMIO region of this device.

In summary, we first corrupt the PSP memory map and then use Device-as-Memory to bypass the PSP read-back check. This way we corrupt the RMP without the PSP detecting it during SEV-SNP initialization. We use Mior to showcase end-to-end exploitation of SEV-SNP on the latest AMD Zen 5 EPYC 9135 SoC with up-to-date firmware. Our two case studies demonstrate how Mior can enable debug capabilities on victim CVMs and forge attestation with 100% accuracy. In a few milliseconds of PoC execution, both case studies allow arbitrary reads/writes of CVM memory without getting detected.

To address the root cause for Mior, the PSP should build the memory map while accounting for the divergent MMIO configurations that are allowed on the Data Fabric. Alternatively, AMD can disallow divergent Data Fabric configurations at least when SEV-SNP is being enabled on the platform. As both of these require firmware-level or hardware changes to proprietary components, we are unable to implement and verify the effectiveness or performance impact of these changes. Mior breaks SEV-SNP via a misconfigured interconnect. It is therefore the fourth instance of *Interconnect Corruption Attacks* (XCA), wherein an attacker compromises the security guarantees of TEEs using their interconnects [15, 32, 33]. We hope that this continued demonstration of weaknesses in AMD Infinity Fabric under the confidential computing threat model motivates a systematic defense.

Contributions. Mior makes the following contributions.

(1) We identify a flaw in the Data Fabric routing on the latest AMD EPYC SoCs that support confidential computing.
(2) We perform Data Fabric misrouting and show that the gaps in PSP memory map construction logic fail to detect it. We then introduce Device-as-Memory to further bypass PSP checks when it initializes the Reverse Map Table (RMP) in DRAM.
(3) Our end-to-end attack called Mior breaks SEV-SNP guarantees with 100% accuracy and gains arbitrary read/write over victim confidential VMs.

Responsible Disclosure. We reported our findings to AMD in February 2026. AMD thanked us for the disclosure, acknowledged the vulnerability, and asked for an embargo until November 10th 2026. AMD plans to issue CVE-2026-0462, mitigate the vulnerability, and roll out updates before the embargo.

2 Background

We introduce the required background needed for Mior. In particular, we describe the required details of the AMD EPYC Platform, such as the interconnect, the Data Fabric, and AMD confidential computing features, as well as the Peripheral Component Interconnect Express (PCIe) hierarchy.

2.1 AMD Confidential Computing

AMD offers confidential computing in the form of AMD *Secure Encrypted Virtualization* (SEV). SEV and *Secure Encrypted Virtualization-Encrypted State* (SEV-ES) allow the isolation of *Confidential Virtual Machines* (CVMs) from each other and the hypervisor by transparently encrypting the guest data in DRAM. The platform then ensures that only code running in the correct execution context, i.e., the guest, can correctly decrypt this memory. Crucially, neither SEV nor SEV-ES provide integrity protection for guest memory. This lack of integrity lets an attacker modify the ciphertext, either to introduce uncontrolled changes or to replay previously seen ciphertexts [43].

Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP), the successor of SEV and SEV-ES, addresses a shortcoming where a malicious hypervisor could perform remapping attacks [43]. Thus, SEV-SNP protects the integrity of guest memory from a malicious hypervisor with the help of the *Reverse Map Table* (RMP). The RMP is a large in-memory array that contains access control information for every physical page. During address translation, the RMP ensures that the current execution context has permission to access the memory. Similarly, the *Input–Output Memory Management Unit* (IOMMU) blocks all device accesses to memory that are not hypervisor-owned in the RMP. As the RMP contains entries for every physical page, some of these entries also protect the RMP itself by prohibiting the hypervisor from modifying it. These entries protecting the RMP itself are protected by a special RMP entry, the *self-protecting entry*. Consequently, being able to modify this entry allows modification of the entire RMP [1, 6, 16, 33, 34]. Since the RMP is security-critical, the PSP, which is the AMD root-of-trust, administers it.

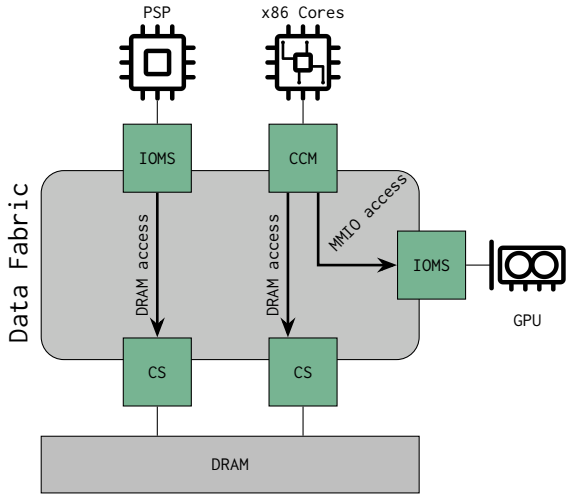


Figure 1: The Data Fabric routes data accesses. DRAM accesses terminate at Coherent Slave (CS) blocks and MMIO at I/O Master/Slave (IOMS) blocks. This also applies to accesses by the PSP.

2.2 PSP

The *Platform Security Processor* (PSP) serves as the AMD on-chip security co-processor and bootstraps trust for the entire platform. Among others, it is responsible for platform initialization (e.g., DRAM initialization) and SEV-SNP initialization.

PSP for SEV-SNP. For SEV-SNP CVMs, the hypervisor is responsible for CVM management tasks (e.g., CVM scheduling). More security-sensitive tasks, such as moving CVM pages or attestation, are under the control of the PSP. Despite being prohibited from accessing it, the hypervisor still needs a method to move CVM data within the physical address space. To this end, the PSP exposes the SEV-SNP API to cooperate with the hypervisor [6]. For instance, to allow the hypervisor to move CVM data, the PSP implements the `SNP_PAGE_MOVE` command, which carefully checks the hypervisor-provided parameters and then relocates CVM data. Furthermore, the PSP provides debug functionality. If enabled by the CVM owner, `SNP_DBG_ENCRYPT` and `SNP_DBG_DECRYPT` allow a hypervisor to read and write guest memory. For the SEV-SNP API and other tasks, the PSP requires access to DRAM. For example, for SEV-SNP to function securely, it also has to set up in-DRAM data structures (e.g., RMP). Since the PSP attaches to the Data Fabric as a device, the routing of its accesses depends on the Data Fabric configuration.

2.3 Data Fabric

Modern SoCs consist of many components. For AMD Zen SoCs, these components are additionally spread across multiple dies, called *chiplets*, on the same CPU substrate. For these SoCs to still function, a fast interconnect between components is required. Part of the AMD interconnect is the so-called *Data Fabric* (DF). The Data Fabric is the component that provides a high-bandwidth interconnect for data accesses on the AMD platform. We explain the aspects of the Data Fabric required for this work.

Data Fabric Blocks. At a high level, the Data Fabric consists of different blocks that route requests to one another. Requests enter the Data Fabric through *ingress* blocks, and exit through *egress* blocks. When a component on the SoC, e.g., a core, requires data from some remote component, it sends a request to its ingress point. For instance, when a CPU core issues a memory load, the caches are checked first; if they do not return a hit, the data must be fetched from DRAM. The core, therefore, sends a request to its ingress point to the Data Fabric, called a *Core Coherent Master* (CCM). In other words, the CCM is the gateway for the CPU core into the Data Fabric. Based on an internal routing configuration, this block will determine which destination block to target for this request. Depending on the target type, MMIO or DRAM, a different destination block will be targeted.

CS and IOMS. If the request is for DRAM, the destination block will be one of the *Coherent Slave* (CS) blocks. The CS block will then directly access DRAM to retrieve the data. A request to an MMIO region of memory would instead be routed to an *I/O Master/Slave* (IOMS) block. IOMS blocks serve as both ingress and egress points. For example, it serves as an egress point when a CPU core issues a memory load to an MMIO address of a specific device. The CCM then routes the access to the corresponding IOMS of the targeted device. On the other hand, it serves as an ingress point when a device attached to an IOMS issues a request that targets DRAM or a device connected to a different IOMS. This applies to both external devices connected via PCIe (e.g., GPUs) and integrated devices (e.g., PSP or USB controller). Figure 1 illustrates this routing behavior. The number of IOMS and CS blocks on an SoC varies. Current AMD SoCs have up to 8 IOMS and 12 CS blocks [2].

2.4 Data Fabric Routing

As established in Section 2.3, the Data Fabric ingress points route accesses to other components on the Data Fabric. The Data Fabric routes accesses based on the routing configuration. This routing configuration is governed by configuration registers that the BIOS and the PSP initialize during boot [11]. Specifically, the PSP handles the setup of DRAM routing, while the BIOS configures MMIO routing. For this, the BIOS first enumerates all devices to identify their capabilities and memory requirements (see also Section 2.5) to set up a suitable MMIO routing configuration. Using the enumerated device data, the BIOS provisions the MMIO routing registers so that any request hitting a specific MMIO range is forwarded to the corresponding IOMS. The correct IOMS block is the IOMS that connects the device, which is configured with the MMIO address.

To resolve routing destinations, AMD employs an address region-based routing mechanism. A single routing configuration consists of a set of four registers, mainly a base and limit address, and a target instance ID. The base and limit values encode a memory range that is then routed to the component with the target instance ID. These target IDs are architecturally hardcoded; for example, IDs `0x20` through `0x27` designate IOMS blocks, while `0x10` through `0x14` designate CCM blocks.

Each Data Fabric ingress point contains 20 MMIO routing register sets [2]. Consequently, every IOMS possesses 20 discrete MMIO configurations, each encoding a base and limit address with a corresponding target ID. Additionally, these registers encode information

such as whether the range is read-enabled or write-enabled, as well as potential extensions of the address space for MMIO accesses. Assuming the BIOS applies a uniform configuration across all ingress nodes, the platform supports defining up to 20 distinct MMIO address regions.

2.5 PCIe Devices

We establish the required background on *Peripheral Component Interconnect Express* (PCIe) necessary for the subsequent experiments. Devices, such as network cards, Graphics Processing Units (GPUs), and Tensor Processing Units (TPUs), can interface with the host via the standardized PCIe communication interface [29]. PCIe organizes these components hierarchically, starting from a root node called the *Root Complex* (RC). Below the root complex are its *Root Ports* (RPs), which can connect directly to devices. Devices serve as leaves of the PCIe hierarchy.

To interact with the PCIe devices, software running on the host needs a method to discover and configure them. For this purpose, each PCIe entity exposes the PCIe configuration space. The PCIe configuration space is a 4 KiB register space that contains information about the entity and allows configuration. In addition to the configuration space, the host can interact with devices via MMIO. MMIO can expose control registers or device memory directly to the host. The device stores its MMIO base addresses and ranges within its PCIe configuration space in *Base Address Registers* (BARs), which are configured by the host. The host must also configure the platform in a way that the BAR regions of all devices are accessible via the Data Fabric, i.e., all BARs are part of an MMIO configuration. This means that the MMIO configuration of the IOMS that connects to the device must also contain the BAR of that device.

3 Mior Overview

We provide a high-level overview of Mior.

Threat Model. We operate in the AMD SEV-SNP threat model [1]. Only the AMD-supplied hardware (SoC), AMD-supplied microcode, PSP firmware, and CVM code and data are trusted. The hypervisor and the BIOS are untrusted and assumed to be under attacker control. Additionally, the malicious hypervisor can use commodity PCIe devices under its control (e.g., GPUs or NICs). We do not assume single-stepping capability.

PSP Memory Map. Recall from Section 2.2 that the PSP must access DRAM for certain operations, such as updating the RMP or moving CVM pages. To defend against attacks, the PSP must carefully validate these DRAM accesses. For instance, a malicious hypervisor could provide a physical address that maps to MMIO rather than DRAM, and then instruct the PSP to perform a write operation at that location. If the PSP fails to detect this subversion, the Data Fabric routes the write to the IOMS for MMIO. The IOMS then forwards the access to a device BAR or silently drops it if no device BAR exists at that address. Consequently, the targeted DRAM address remains unmodified, which can lead to unprotected data (e.g., if the PSP intended to update the RMP to mark a page as guest-private). To prevent this, the PSP builds an internal memory map detailing which physical address ranges are not DRAM and consults it prior to every DRAM access. Thus, even if the hypervisor maps MMIO memory as DRAM and tries to trick the PSP, the PSP check detects this before performing DRAM reads/writes.

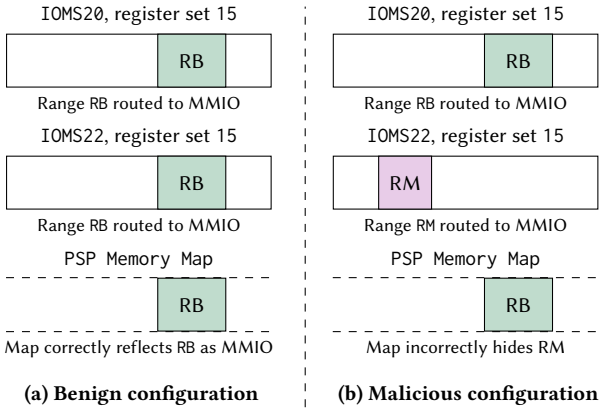


Figure 2: Example of how a careful misconfiguration can remain hidden from the PSP memory map. RB is a benign MMIO memory range, spanning from `0xd3000` to `0xd3fff`. If routing configuration 15 at all ingress points routes accesses to RB to an IOMS (case (a)), the resulting memory map will correctly qualify RB as being MMIO. Contrast this with case (b). By carefully choosing the bounds of a malicious region, RM, and programming this as configuration 15 on an IOMS, the attacker can hide it from the PSP memory map. Concretely, by spanning RM from `0x1000` to `0x1023`, the resulting ORs of the base and limit values are equal to those for RB. Thus, completely hiding RM in the PSP memory map.

3.1 Insecure Configuration Read

We identify a fundamental flaw in how the PSP obtains the MMIO configuration that it uses to build its internal memory map. Recall that there are several IOMS blocks that have their own MMIO configurations. An IOMS block's MMIO configuration encodes which physical addresses are MMIO and where MMIO accesses are routed, i.e., the correct IOMS. The Data Fabric then applies this routing configuration to accesses entering the Data Fabric via these IOMS.

To get an accurate memory map that tracks which address ranges are MMIO, ideally, the PSP must collect all MMIO ranges from each IOMS block. To achieve this, the PSP must read MMIO routing registers (e.g., base and limit registers) across the Data Fabric. The Data Fabric contains 20 such register sets for each ingress point (e.g., from an x86 core to the CCM, or from a device to the IOMS). Each set consists of a (base, limit) pair to delimit the address range routed to MMIO.

We experimentally find that, instead of querying each component individually, the PSP issues a *broadcast read* to fetch all routing configurations simultaneously. This *broadcast read* computes a logical OR across all component configurations and returns the aggregated result to the PSP. Figure 2(a) shows the effect of this operation. Crucially, this logical `OR` can completely mask certain routing configurations. For example, as shown in Figure 2(b), suppose some IOMS blocks are configured with a base of `0x1000` while some other IOMS block is configured with a base of `0xd3000`. Because

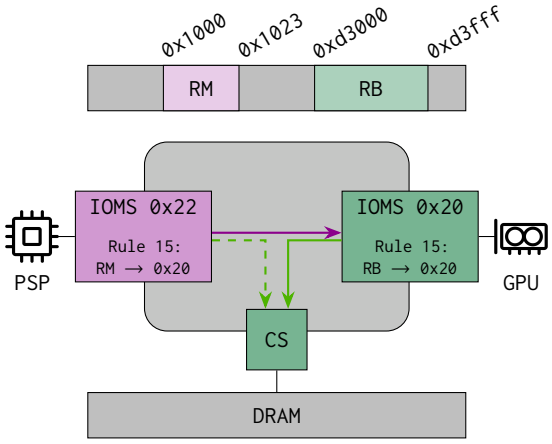


Figure 3: Illustration of how setting up routing configuration 15 differently on different routing blocks affects the platform. Standard system software chooses the original range RB from `0xd3000` to `0xd3fff` and installs it as configuration 15 on all IOMS blocks. An attacker chooses the malicious range RM from `0x1000` to `0x1023` and overwrites configuration 15 on IOMS `0x22`. The bounds for RM are chosen so they are hidden when ORed with the bounds of RB, thereby hiding this range from the PSP's memory map. Accesses into RM from IOMS `0x20` still correctly reach DRAM, but the platform routes accesses from IOMS `0x22` to IOMS `0x20`.

the bits of `0x1000` overlap completely with `0xd3000`, the broadcast read yields `0xd3000`. If the PSP builds a memory map based on this broadcast read, it will treat all addresses below `0xd3000` as DRAM, including the `0x1000` range, which is actually MMIO. Thus, if an attacker can change the MMIO configuration of an IOMS block, it will corrupt the PSP's memory map, which can render the checks useless. Furthermore, the attacker can specifically target the IOMS block that the PSP connects to. As a result, as shown in Figure 3, when the PSP attempts to access `0x1000`, the Data Fabric applies the MMIO config of the IOMS block to which the PSP connects, i.e., the one that the attacker has corrupted. Put together, the memory map fails to detect the misconfigurations, and the misconfiguration of the PSP's IOMS enables rerouting from DRAM to MMIO. Lastly, we find that the malicious hypervisor can manipulate the MMIO configuration before the PSP builds the memory map. By combining this capability with the flawed *broadcast read* logic, an attacker can inject malicious MMIO configurations that are absent in the PSP memory map. Specifically, it can inject a malicious MMIO configuration on the IOMS that connects the PSP to the Data Fabric. Consequently, the tampered configuration remains entirely invisible to the PSP. Nothing in the PSP memory map hints at the tampering or the malicious routing configuration.

3.2 Building the Mior Primitive

To exploit this malicious routing capability to break SEV-SNP, we introduce the Device-as-Memory primitive. While malicious routing lets us redirect PSP DRAM accesses to an MMIO region, it does not inherently allow us to tamper with PSP reads/writes. Thus, we must ensure these accesses terminate at an attacker-controlled device. The Device-as-Memory primitive redirects PSP accesses intended for DRAM to attacker-controlled PCIe MMIO pages. By backing these pages with a device under the attacker's control (such as a GPU), the attacker gains bidirectional control: they can observe the data the PSP writes and control the values the PSP reads. As a concrete use-case of Device-as-Memory, we target the PSP's RMP initialization logic, which is essential to protect CVM memory and, more crucially, the RMP itself. Specifically, the PSP is supposed to write RMP entries to the DRAM. By using Device-as-Memory, we coerce the PSP to perform the RMP write to MMIO space. This leaves the *actual* RMP uninitialized. Without initializing the RMP, integrity protections collapse: the malicious hypervisor can freely modify CVM data and metadata. Ultimately, we exploit this lack of integrity to enable debugging on production CVMs and forge attestation evidence to fully subvert SEV-SNP security guarantees.

4 Fabric Routing Exploration

AMD SEV-SNP supports integrity protection of guest memory using the RMP. This protection only applies to physical addresses. The platform must enforce separation between different memory regions, such as DRAM and MMIO. Otherwise, confusion between DRAM and MMIO pages can lead to attacks against SEV-SNP. An attacker could direct accesses destined for DRAM to MMIO. Then, an attacker can abuse this to observe and manipulate plaintext data via the underlying device of that MMIO address [16]. To mitigate against such attacks, AMD excludes MMIO for SEV-SNP usage. The PSP grounds its decision on which memory ranges to exclude on a memory map. This memory map, built by the PSP, describes which regions in the physical address space are DRAM and which are MMIO. Before executing operations that require the physical address to reside in DRAM, the PSP verifies its accesses against this memory map. For example, Listing 1 shows that the PSP validates whether a guest context page resides in DRAM. Consequently, to thwart potential attacks, the PSP must build the memory map correctly.

```
status = validate_memory_map(&gpDram->mem_map,
    gc->gctx_paddr,
    gc->gctx_paddr + PAGE_SIZE_4K - 1ULL);
if (status != SEV_STATUS_SUCCESS)
{
    status = SEV_STATUS_INVALID_CONFIG;
    goto end;
}
```

Listing 1: Usage of the memory map by the PSP [4].

4.1 Recovering the Memory Map

Given its role in the security of SEV-SNP, we investigate how the PSP builds the memory map. Reviewing the publicly released source code, we find many usages of the memory map (see Listing 1) [4]. Unfortunately, the publicly available code does not contain the part that actually builds the memory map. This code resides in the PSP operating system, which is closed-source. To infer details of how the PSP constructs the memory map, we instead attempt to recover it indirectly. The PSP does not expose functionality to obtain the memory map directly; however, it initializes the RMP based on this map. Concretely, during initialization, the SEV firmware uses

the PSP memory map and initializes the RMP so that all address ranges corresponding to MMIO memory are set to `0x4`. This value corresponds to the `HYPERVISOR_FIXED` RMP state. Therefore, we can scan the RMP in DRAM to partially recover the memory map of the PSP. Implementing this, we note that the map marks certain additional ranges as `HYPERVISOR_FIXED`, but all ranges marked as MMIO are contained within it. While we can recover the PSP memory map, we cannot infer how the PSP builds it.

4.2 How the Memory Map is Built

The memory map distinguishes between MMIO and DRAM; thus, we investigate how untrusted software configures those on the platform. Our reverse-engineering efforts identify two main configurable entities. (1) CPU core configuration, which controls whether an access is for MMIO or DRAM and (2) Data Fabric routing configuration that governs whether the Data Fabric routes accesses to DRAM or MMIO. BIOS sets up the MMIO core configuration during boot. It configures per-core Model-Specific Registers (MSRs) to specify which memory requests from the core are MMIO or DRAM accesses. Our analysis reveals that SEV firmware explicitly checks these MSRs during RMP initialization while also checking the memory map. Based on this, we hypothesize that the memory map does not reflect MSR core configurations. If it did, the PSP would have no reason to check both separately.

This leaves us with another option, the Data Fabric routing configuration. To investigate this further, we refer to `openSIL`, an open-source library that implements functionality for early platform initialization on modern AMD CPUs. Reviewing the `openSIL` source code, we find definitions for configuring MMIO ranges (see Listing 2). Concretely, as described in Section 2, we find that 20 base and limit registers define MMIO ranges on the Data Fabric. Once a memory request—designated as MMIO by the core configuration—reaches the Data Fabric, it is evaluated against these 20 ranges. If the request falls within one of them, the Data Fabric applies the corresponding routing rules to determine the final destination for the memory access [2, 33].

4.3 Modifying the Memory Map

In the next step, we modify the Data Fabric MMIO configuration and compare it against the memory map to see if it reflects our changes.

Synchronous Routing Changes. When reading the MMIO configuration, we notice that all ingress points (e.g., all IOMS and CCM) contain the same 20 MMIO range configurations. For example, the first MMIO configuration on all ingress points ranges from `0xf0000000` to `0xf6ffffff` and designates instance ID `0x20` as the target. Naturally, this instance ID belongs to an IOMS block since it targets MMIO. Thus, in our first experiment, we change a single routing configuration across all ingress points to maintain this uniformity. Specifically, we modify the first MMIO configuration to cover the physical address range from `0x80000000` to `0x86ffffff`. Then, we lock the Data Fabric and instruct the PSP to initialize SEV-SNP, thereby also initializing the RMP. We again use the RMP to infer the memory map, which now also maps our modified MMIO range as `HYPERVISOR_FIXED`, thereby confirming that the MMIO configuration directly affects the memory map.

```
#define MMIOBASEADDRESS_0_FUNC 0x0
#define MMIOBASEADDRESS_0_REG  0xD80
#define MMIOBASEADDRESS_0_PCI_ADDR (MAKE_SBDFO (0, 0,
      0x18, MMIOBASEADDRESS_0_FUNC,
     MMIOBASEADDRESS_0_REG))

#define MMIOLIMITADDRESS_0_FUNC 0x0
#define MMIOLIMITADDRESS_0_REG  0xD84
#define MMIOLIMITADDRESS_0_PCI_ADDR (MAKE_SBDFO (0,
     0, 0x18, MMIOLIMITADDRESS_0_FUNC,
     MMIOLIMITADDRESS_0_REG))

#define MMIOADDRESSCONTROL_0_FUNC 0x0
#define MMIOADDRESSCONTROL_0_REG  0xD88
#define MMIOADDRESSCONTROL_0_PCI_ADDR (MAKE_SBDFO (0,
      0, 0x18, MMIOADDRESSCONTROL_0_FUNC,
     MMIOADDRESSCONTROL_0_REG))
```

Listing 2: Definitions of MMIO config registers in `openSIL` [5].

Asynchronous Configuration. We point out that we can change the routing configuration for individual ingress points. This leaves the Data Fabric with an asynchronous configuration as different ingress points map different memory regions differently, leading to disparate views of the physical address space. This difference could have severe security implications for SEV-SNP if not correctly accounted for. We could configure the Data Fabric such that one core accesses MMIO at address `0x1000`, while another core accesses DRAM when sourcing the same address. In our next experiment, we dissect how the PSP handles asynchronous routing registers in the memory map. For this experiment, we reconfigure only the routing registers of one IOMS block rather than all available ingress points. We configure it such that the set of bits in the IOMS routing is a subset of the bits in the remaining registers. Subsequently, we source the RMP again to see how the PSP handles this case. We see that the two RMP configurations are identical. This means that the PSP does not individually check the different MMIO routing configurations and excludes them, but instead, the PSP seems to ignore our configuration.

While searching for an explanation for this behavior in the source code, we find a feature in AMD hardware called *broadcast read*. Listing 3 details the comment. A *broadcast read* of a routing register reads the OR-sum of all ingress values for one register. Based on the comment, we assume that the PSP does not obtain the individual ingress routing configuration but sources the global routing for all ingress points using a *broadcast read*. The data we obtained aligns with our hypothesis. We conclude that the PSP has an inaccurate representation of the actual memory routing in its memory map.

4.4 Abusing Memory Map Deviation

Since the SEV firmware grounds its security checks on the memory map, an incorrect map may have devastating consequences for SEV-SNP security. To show the impact of an incorrect memory map, we conduct an experiment where we misconfigure the routing of the IOMS. More precisely, we can use carefully chosen addresses to hide certain routing registers from the memory map (cf., Figure 2).

For instance, if we have a benign configuration of `0x150000` as base and `0x160000` as limit, we can configure IOMS `0x22` with `0x50000` as base and `0x60000` as limit. When the PSP sources the memory map, the resulting entry will show `0x150000` as base and

```
typedef union {
  struct {
    uint32_t Reserved_0:1;
  ///< PSP must set this bit to 1 to lock this
  ///< register after setting up address maps,
  ///< routing tables, flow control, and all the
  ///< fabric per-instance registers. When this
  ///< bit is set to 1, Data Fabric
  ///< configuration access operates in
  ///< broadcast mode. This register can be
  ///< updated only after a system wide reset;
  ///< reads return OR-reduction of all
  ///< instances of the addressed register. For
  ///< information on per-instance access after
  ///< this register is locked, see
  ///< DF::FabricIndirectConfigAccessAddress.
    uint32_t CfgRegInstAccRegLock : 1;
    uint32_t :14;
    uint32_t Reserved_16:8;
    uint32_t :8;
  } Field;
  uint32_t  Value;
} FABRIC_CONFIG_ACCESS_CONTROL_REGISTER;
```

Listing 3: Comment in openSIL [5] indicating the existence of a *broadcast read* functionality.

`0x160000` as limit, completely hiding the other configuration. Thus, we created a routing configuration that is invisible to the PSP.

4.5 Validating PSP Misrouting

Given these findings, we next validate that our MMIO misconfiguration influences the PSP, even though the PSP remains oblivious to it. For this purpose, we insert a malicious routing that should affect PSP accesses intended for DRAM. Like other devices, the PSP itself connects to the Data Fabric through an IOMS. To inject a configuration that affects the PSP, we must first determine which IOMS it uses. The PSP exposes MMIO regions so the operating system can interact with it. On our system, these are in the range `0xd5200000` to `0xd5300000`. Reviewing the existing routing configurations, we find that the Data Fabric routes accesses in the range from `0xd4000000` to `0xdcffffff` to IOMS `0x22`. Therefore, we conclude that the PSP connects to IOMS `0x22`.

We therefore target IOMS `0x22` for our misconfiguration. Specifically, we inject a routing configuration that the *broadcast read* entirely masks. The subsequent SEV-SNP initialization succeeds without errors. We then use PSP `SNP_PAGE_MOVE` with a dummy CVM to move a guest page into the injected MMIO region. The operation succeeds, and we use the hypervisor to verify the intended target location in DRAM, confirming that its contents remain unchanged. Correspondingly, we repeat this experiment, using the injected MMIO region as the source address and applying a `SNP_DBG_DECRYPT` operation, such that the PSP writes the data in plaintext to DRAM. We observe that the resulting data consists of `0xff..ff`. Together, these experiments show that the PSP reads return an error value, whereas the Data Fabric silently discards writes.

This confirms that the PSP memory accesses adhere to our modified configuration, and also that it is unaware of our attack. Otherwise, it would signal an error when comparing its access with its memory map.

4.6 Misrouting RMP Initialization

We now attempt to use our insights to compromise SEV-SNP initialization. Specifically, we attempt to reroute PSP accesses during RMP bootstrapping. If the Data Fabric silently discards PSP writes to the RMP, SEV-SNP initialization will succeed while the RMP stays uninitialized, leaving SEV-SNP in a compromised state. Specifically, we inject a routing configuration that comprises parts of the RMP, specifically the self-protecting entry. Recall from Section 2, if the self-protecting entry is missing, the whole RMP can be modified and therefore disabled.

Unfortunately, performing this attack results in an error during `SNP_INIT`. Concretely, the PSP returns error `0x26`, which corresponds to `RMP_INIT_FAILED`. Comparing this with the SEV firmware source code, we find no explanation for this behavior. Thus, we repeat this experiment; this time, we apply our malicious configuration to all ingress points. If our attack succeeded in manipulating the memory map via the *broadcast read* but failed for another reason, we expect a different error. We observe that this time, the PSP detects the misconfiguration and signals error `0x03` (`INVALID_CONFIG`). This confirms that we successfully mask the malicious configuration but fail initialization for another reason.

5 Device-as-Memory

Our experiments in Section 4.5 demonstrate that our MMIO routing modifications successfully influence PSP DRAM accesses. However, attempts to tamper with the RMP initialization using this technique fail. We analyze the SEV firmware source code to find an explanation for this behavior, but without success. Instead, we hypothesize that the PSP performs a previously unreported read-back verification of the values it writes to the RMP. In our experiments, we confirm that the Data Fabric silently discards PSP accesses routed to unassigned MMIO addresses (i.e., addresses not belonging to any device). Consequently, any subsequent read-back check would fail due to the discarded write, explaining the observed failure. Such a check could have been added as an additional defense against previous RMP initialization attacks. See our discussion in Section 8 for more details. To investigate this hypothesis and, more generally, to obtain a stronger primitive, we configure the platform to map the PSP destination address to an attacker-controlled device that exposes its device memory via MMIO. This setup allows the hypothesized read-back check to succeed, enabling an attacker to manipulate both PSP writes and reads. We refer to this new primitive as *Device-as-Memory*.

For this primitive to be viable, we require a device that provides the attacker with precise control and exposes sufficient device memory via MMIO. Potential candidates include internal components (e.g., NICs or USB controllers) as well as external peripherals (e.g., GPUs or TPUs). Because internal devices are generally more difficult to control and offer less versatility, we turn our attention to external devices. To evaluate the feasibility of using a PCIe device, we opted for an NVIDIA GPU; these are widely available in cloud environments and expose large amounts of device memory via MMIO to meet our read/write requirements. Because capitalizing on this setup requires us to manipulate the underlying PCIe routing, we must first understand how these devices are integrated and configured within the system.

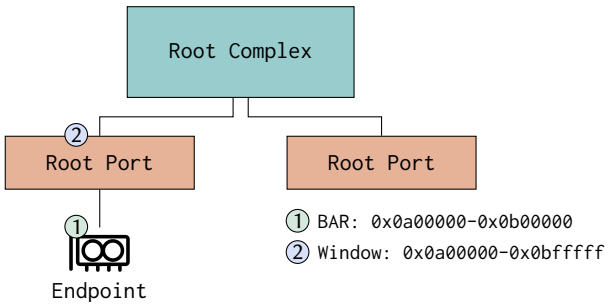


Figure 4: PCIe routing hierarchy. The root complex is the interface between the SoC and PCIe. The root complex can contain a sub-tree; thus, its routing range may cover the BAR of multiple devices. Devices are endpoints and expose one or multiple BARs for MMIO.

PCIe Configurations. Recall (from Section 2.5) that the host must configure the PCIe hierarchy. This is commonly performed by the BIOS during platform boot. BIOS has to assign each device appropriate BAR regions based on the sizes the device requests. It queries each device for the carve-out sizes and alignment requirements required in the physical address space, assigns address ranges accordingly, and writes the address into the BARs to inform the device. For example, ① in Figure 4 illustrates such a BAR configuration. As PCIe is hierarchical, the parent node of a device also has to be configured to expose that range. In the example given in Figure 4, this would be the root port: The host has to configure it such that its *window* covers all addresses accessible for all devices downstream of that root port. See ② in Figure 4 for such a configuration. While these configurations are often set during platform boot, they can be changed. In our testing, we found that the platform with an NVIDIA GPU responds to such changes of BAR configuration and correctly behaves with the new base address configured.

5.1 P2P Exploration

PCIe not only allows the host to communicate with the device, but also provides ways for other devices to communicate directly with each other. Such device-to-device interaction is called Peer-to-Peer (P2P). We aim to force the PSP access to terminate at an attacker-controlled device. Effectively, this represents a P2P transaction between the device and the PSP. Specifically, to create a versatile primitive, we need to do this at arbitrary physical addresses. Thus, we first test P2P functionality on our platform, then verify that we can adjust the BAR address and still perform P2P, and finally enable P2P to a BAR at any address. For our experiments, we use an NVIDIA RTX 3080 as backing memory. This GPU contains a BAR that exposes 256 MiB of device memory for MMIO. Additionally, we use a PCIe Squirrel to issue read/write requests simulating the PSP [22]. The PCIe Squirrel is a small dedicated device that lets the host program it to send arbitrary PCIe transactions.

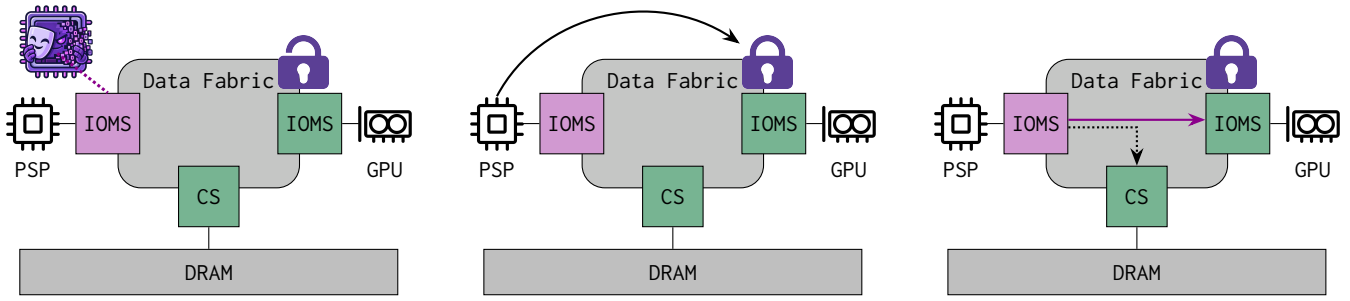
P2P. We begin by verifying that our platform supports P2P. Since our final exploit abuses a misconfiguration in the Data Fabric, we want to test P2P accesses that traverse the Data Fabric. To do this, we must ensure that the devices attach to different Data Fabric IOMS blocks; otherwise, the P2P transaction will be routed only within the PCIe hierarchy. We can easily guarantee this by attaching both devices to different physical PCIe slots, which causes them to connect to different PCIe root ports. We confirm this by verifying that their BAR physical address ranges map to distinct IOMS instances in the MMIO routing. Concretely, we use this to infer that the Squirrel and the GPU attach via `IOMS 0x22` and `IOMS 0x20`, respectively. We now verify P2P functionality by issuing a request from the PCIe Squirrel to the GPU BAR at `0x11fe0000000` under a benign configuration. Unlike the PSP, the PCIe Squirrel is subject to IOMMU checks. Thus, we use the hypervisor to add IOMMU entries to allow the PCIe Squirrel to access the GPU BAR. We confirm that both read and write requests from the Squirrel to the GPU BAR region succeed.

Relocating a BAR. Next, we test if we can change the physical addresses that our target device responds to and still access it. Recall that the BAR configuration register allows relocation of its BAR by writing a different address to it in the PCIe configuration space. We now change this address by adding 4 KiB to it, thus placing it at `0x11fe0001000`. We then reissue our memory requests and confirm that the GPU correctly responds to the new address range. Next, we investigate how far we can relocate the BAR while P2P still succeeds. We find that if we move the BAR by more than 288 MiB the P2P transaction fails, returning `0xff` for reads and silently discarding writes. Consequently, we investigate why this fails and how we can rectify it.

PCI Bridge Windows. We initially assumed this failure occurs because the new address falls outside the MMIO configuration targeting the appropriate IOMS. In other words, the MMIO configuration does not contain the address at all or routes it to a different IOMS. However, we find that even when we adjust the MMIO routing configuration to account for this, the access still fails. Since PCIe has a hierarchical structure, PCIe components that the request traverses must also be aware of the address change. In particular, PCIe components responsible for routing requests (e.g., root ports) contain a configuration in their PCIe configuration space that determines which addresses may be handled by devices located below them in the hierarchy. These configuration registers are referred to as *bridge window registers*. These windows, defined by a base and limit, instruct switches and root ports which accesses they should forward to a device. In Figure 4, ② is such a window configuration. Therefore, when we relocate the BAR with an address outside of the window of the GPU root port, we observe the accesses failing. To rectify the failure, we change bridge windows. Afterward, we can relocate the BAR region of the GPU to an arbitrary base address, as long as a Data Fabric routing rule also covers it.

5.2 Building Device-as-Memory

Finally, we have obtained all pieces to build Device-as-Memory: our end-to-end attack primitive that redirects PSP DRAM accesses to an attacker-controlled device. We begin by selecting a target physical address in DRAM; in our case, we choose the RMP base address and align it to the BAR size of the attacker-controlled device (`0x91900000`). As shown in Figure 5 a), we then modify the MMIO configuration based on our observations in Section 4.1. Specifically, we target register set 15 of IOMS `0x22`, which is the IOMS



(a) The attacker (symbolized by the Mior logo) misconfigures the MMIO configuration on the Data Fabric.

(b) The PSP locks the Data Fabric and builds the memory map. This map does not correctly reflect the malicious configuration.

(c) Oblivious to the PSP, critical accesses to (trusted) DRAM are redirected to Device-as-Memory

Figure 5: Mior attack overview: The attacker carefully misconfigures the Data Fabric routing in a way the PSP will not reflect in its memory map. It then instructs the PSP to lock the Data Fabric, after which the PSP can initialize SEV-SNP and build its memory map. Finally, memory map checks for PSP accesses to DRAM are bypassed, and security-critical accesses get misrouted to MMIO.

the PSP connects to. The benign configuration describes an MMIO range from `0x1dff9000000` to `0x1fffffffffff` that is routed to instance ID `0x27`. Note that BIOS configures this register set identically across all IOMS and CCM blocks. We introduce a malicious configuration that describes an MMIO range from `0x91900000` to `0x9223ffff` with a target ID of `0x20`, matching the GPU IOMS. Figure 5 b) shows how the PSP then locks the Data Fabric and builds its memory map. The *broadcast read* and logical OR operation do not perfectly mask our injection since the resulting MMIO region in the map is a smaller sub-area of the benign region (starting at `0x1dff9900000`). Thus, while not perfect, this area remains entirely disjoint from our injected region and therefore successfully bypasses access checks.

Next, Figure 5 c) shows that we adjust the PCIe configuration and instruct the PSP to perform its operation on DRAM. Specifically, we configure the bridge window of the GPU root port and BAR 1 of the GPU to start at `0x91900000`. We then instruct the PSP to write known data to `0x91900000` by invoking the `SNP_GUEST_STATUS` command, which writes the guest status data structure to a chosen physical address. Afterward, we revert the BAR relocation and bridge window modifications to read the BAR region via the hypervisor. We observe the expected guest state values, confirming that the Data Fabric successfully routes the PSP writes to the GPU. In other words, this confirms that Device-as-Memory provides us with full control over PSP accesses to DRAM while the PSP remains completely oblivious to the attack.

6 Case Studies & Implementation

To demonstrate the capabilities of Mior, we first show how an attacker can use Device-as-Memory to compromise the RMP. Subsequently, we present two case studies demonstrating the impact of a compromised RMP.

System Setup. We use the same system setup as during the validation of Device-as-Memory. We test our case studies on a Zen 5 AMD EPYC 9135 16-core processor running microcode version `0xb00215a`. While we have Zen 3 and Zen 4 CPUs, we do not have a setup (i.e., changeable BIOS) for them, and therefore could not perform the experiments on these generations. However, we confirmed that other generations also use the same Data Fabric routing configuration logic. Since our reverse engineering is heavily grounded in public Zen 4 documentation, we believe the PSP uses the flawed logic across other generations, making our attack applicable there as well. The SEV-SNP API version is `1.58.4`. All case studies run a modified Linux `6.19.0-rc7` commit `63804fed149a` host kernel. Our core attack, the rerouting and backing of RMP writes by the PSP, is implemented in 206 LoC in the Linux kernel. We use individual kernel modules to execute the guest (meta)data corruption for the following case studies.

Impact of Ciphertext Hiding. Ciphertext hiding is an optional feature that prohibits the hypervisor from even reading the encrypted guest memory. It does so by replacing the ciphertext values that would be returned with a default value, such as `0xff..ff` [6]. We experimentally checked if this feature can thwart Mior. To this end, we tested our case studies in two modes: with ciphertext hiding disabled and enabled. Our case studies succeed irrespective of whether the ciphertext hiding feature was on or off. This confirms the expected outcome, as our case studies attack the guest context page, which the PSP encrypts directly and therefore is accessed without the C-bit set [4]. Ciphertext hiding therefore does not stop the hypervisor from reading the guest context page, and therefore does not impact our case studies. In other words, this feature is orthogonal to the root cause of Mior and thus does not stop our attack. We show this experimentally in our case studies.

6.1 Compromising RMP Initialization

We now show how Mior allows the hypervisor to corrupt the RMP. Recall that the RMP protects itself, guest metadata, as well as guest data, and that the PSP initializes its entries in DRAM. Thus, corrupting the RMP completely bypasses SEV-SNP access control.

To achieve this, we use Device-as-Memory to tamper with the PSP during initialization. Specifically, we can divert the initial writes

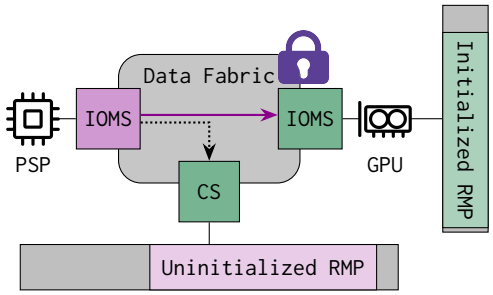


Figure 6: Illustration of the Device-as-Memory primitive for a concrete attack. The PSP is tricked into accessing GPU device memory, thinking it is DRAM. It can verify that the RMP is correctly initialized. The actual RMP located in DRAM, which is used for access control, is left uninitialized.

to the RMP to device memory as shown in Figure 6. On our machine, the RMP is located at the physical addresses `0x91900000` – `0x99efffff`, and its self-protecting entry is at `0x92226260`. Out of the 20 available rules, we use MMIO configuration rule 15. This configuration initially designates the region `0x1dff9000000` – `0x01fffffffff` as MMIO. We now change this configuration on the IOMS (`0x22`) of the PSP. In particular, we overwrite it with a configuration that routes the address range `0x91900000-0x9223ffff` to the IOMS of our GPU, and we also adjust the PCIe routing accordingly. Similar to our experiments in Section 5, the *broadcast OR* does not perfectly mask our configuration. This is fine since the resulting region is completely disjoint from our injected range and therefore not contained in any MMIO region in the PSP memory map. With Device-as-Memory correctly in place, the PSP completes the initialization without errors.

While our range does not cover the RMP entirely, it most importantly redirects the initialization of the self-protecting entry to our GPU. Without the self-protecting entry, the hypervisor can freely modify RMP entries not already covered by our malicious configuration. While the data remains encrypted in memory, we next show how we can use the essentially disabled RMP to compromise SEV-SNP.

6.2 Case Study I: Arbitrary Read & Write with Malicious Debug

Corruption of guest data can then be used to bootstrap a stronger attack. SEV-SNP relies not only on the RMP to protect guest integrity but, crucially, also to protect certain metadata required for SEV-SNP to work. One such metadata structure is the *guest context page*. The hypervisor provides this page, and the PSP uses it exclusively to store guest metadata. Control over the page is transferred to the PSP by calling `SNP_MCMD_ID_GCTX_CREATE`, and from then on the hypervisor should not be able to modify it. The PSP also enforces a strict mapping between guest context pages and SEV-SNP guests by ensuring only one guest context page can exist with the same ASID at the same time. One metadata item contained in the guest context page is the *guest policy*. This policy, among other things, lets the hypervisor arbitrarily encrypt or decrypt guest pages by setting the `DEBUG_ENABLED` bit, bit 19, to 1. If this bit is set, the PSP allows calls to `SNP_DBG_ENCRYPT` & `SNP_DBG_DECRYPT` for this particular guest.

We can use Mior to maliciously toggle this bit after disabling the RMP protection for the guest context page. This lets us write other pages in the RMP, and therefore change the RMP entry for the guest context page to allow the hypervisor to write to it. Unfortunately, while the write protection is now disabled, the page is still encrypted. The encryption uses AES-XEX, an encryption mode instantiated with the AES block cipher. We can therefore not fully control individual bits in the guest context page. Overwriting the ciphertext, however, scrambles the plaintext when decrypting. We can use this fact to repeatedly randomize the block which contains the debug-enabled bit, and should expect it to be toggled high in half the attempts.

This attack can be implemented efficiently by repeatedly writing to any address within the same block—16 bytes in the case of AES—and using the `DBG_` commands to check if the modification worked. Given the pseudo-random property of the encryption, we expect this to succeed in approximately $\frac{1}{2}$ the attempts; therefore, by performing the modification repeatedly, we can virtually guarantee success within just a few attempts. Since the current layout of the guest context page is unknown—the last available information being the open-source PSP firmware release—we instead use a similar trial-and-error approach to find the block within the guest context page containing the debug-enabled bit. We perform this search once and find the offset to be 31 blocks, or `0x1f0` bytes from the base of the context page. The attack requires the corrupted RMP, and therefore uses the same modifications in the Linux kernel. We use an additional kernel module with 381 LoC to perform the search for the correct ciphertext. To test, we use a modified guest kernel with version `6.10`, which writes a known value to a fixed guest physical address for the attack to leak.

We measure the performance of the attack in 1000 cases, both with ciphertext hiding enabled, and with it disabled. We find that the attack terminates after an average of 2.13 attempts with ciphertext hiding enabled, and 2.03 attempts when it is disabled. With ciphertext hiding enabled the highest number of attempts required was 11, while without it the attack terminated in 9 attempts at most. Given the theoretical distribution, such an outlier is expected, and the difference is therefore likely caused by statistical noise. The average time to completion is 1.76 milliseconds with and 1.69 milliseconds without ciphertext hiding. The attack succeeded in all 1000 attempts for both experiment runs. In addition, we can completely hide from the guest. The guest can request attestation at any time, which does contain the current guest policy. During the attack, however, we can simply stop the guest from executing, and restore the original policy value from a snapshot of the ciphertext after completion. Repeating the attack against the same CVM can then be done without any retries by snapshotting the ciphertext of the context page after observing a successful decryption.

6.3 Case Study II: Forging Attestation

SEV-SNP computes the measurement of the CVM image, required for the attestation report, once during the CVM initialization and

stores it in the guest context page. The guest can then request attestation reports at its own discretion. It does so by issuing a hypercall, instructing the hypervisor to request the attestation report. The hypercall contains the attestation request, which is cryptographically protected. The hypervisor cannot tamper with it, except to drop it. Crucially, regardless of any repeat request by the guest, the measurement for the CVM image will not be recomputed.

Our attack now works as follows: First, we boot a benign version of the CVM. We then take a snapshot of the CVM guest context page, which contains the ciphertext for the attestation measurement; this measurement will attest to the benign state of the CVM. While this measurement is encrypted, this encryption is again deterministic when located at the same physical location. The hypervisor is responsible for assigning physical pages, including the guest context page, and can therefore trivially ensure that subsequent CVMs have their guest context pages allocated to the same physical address, as long as the previous one has been cleaned up. After shutting down and cleaning up the benign VM, we then launch a new guest, sharing the same guest context, using a malicious image. This will, at first, create a measurement in the guest context page based on the malicious image. Using the Mior primitive, we then remove the integrity protections of the guest context page and replace the measurement with the previously snapshotted measurement of the benign CVM. Any attestation now requested by the CVM will attest that the image is benign, breaking attestation.

We implement this case study using the common modifications to the host kernel, and 13 additional lines in the host kernel module. Additionally, we use an attack kernel module comprising 213 LoC to perform the writes. We use an unmodified guest kernel with the `snpguest` tool to verify the overwrite of the guest measurement. We tested this case study again with ciphertext hiding enabled to confirm our findings from analyzing the PSP firmware: The guest context page is not encrypted in the same way as normal CVM memory, which is therefore not affected by ciphertext hiding being enabled. This case study therefore works the same regardless of ciphertext hiding. The active part of this attack proceeds quickly, as it requires only a few bytes written by the hypervisor. The attack does, however, require a *preparation* phase, which entails launching the guest in a benign state once and creating the snapshot of the guest context page. Afterwards, the attacker can modify the CVM to their liking, such as introducing persistent read/write primitives, all while the victim is served the benign launch measurement.

7 Discussion

We discuss the root cause, possible mitigations, and the applicability of similar vulnerabilities to other CVM architectures.

7.1 Root Cause

We identify a fundamental root cause enabling Mior.

Asynchronous Routing Configuration. The PSP proactively marks all memory routed as MMIO in the Data Fabric configuration as locked and unusable for SEV-SNP. Additionally, all memory the core marks as MMIO that may differ from Data Fabric MMIO definitions is excluded as well [4]. This ensures that a malicious hypervisor cannot use malleable MMIO pages for SEV-SNP operations. However, as we show in Mior, AMD did not account for architectural nuances when building the memory map. Specifically, we abuse the fact that the memory map can be asynchronous across different components. This means that different Data Fabric ingress points can have a different memory view. The PSP firmware does not consider this distinct memory view when building its memory map. Instead, our experiments and documentation show that it calculates the OR-sum of all Data Fabric unit routing registers and builds the security map on the result. This does not reflect the actual routing configuration and allows us to compromise SEV-SNP.

7.2 Mitigations

We discuss possible mitigations aimed at preventing Mior and Mior-style attacks in the future.

Making MMIO Configuration Privileged. In the current platform initialization model, the BIOS is responsible for PCIe-related initializations. While it is possible, as AMD has done for DRAM initialization, to move this responsibility towards the PSP, this may not actually be desirable. Modern x86 platforms are highly heterogeneous and require extensive configuration during initialization. Implementing this functionality exclusively in the PSP would therefore significantly increase the PSP TCB. It may be even more challenging to transfer responsibility for PCIe initialization to the PSP. PCIe devices may act non-conformantly or change during runtime (PCIe hotplug). Thus, handling all the MMIO configuration within the PSP may not be desirable to limit the TCB.

Address Data Fabric Misrouting. Currently, the PSP data flow specifies in multiple instances which type of memory it wants to interact with. Our experiments show that the Data Fabric ignores these types of requests and first checks the MMIO routing rules. This is different from the CPU cores: x86 (μ-)code states the type of memory it interacts with using other configuration registers, such as the `IORR` MSRs. Our experiments show that the Data Fabric correctly respects the memory type, even in the presence of overlapping Data Fabric rules. The Data Fabric must behave likewise for the PSP and explicitly route memory requests to DRAM if requested. We note that such a mitigation may be difficult to deploy. Doing so may require changes to the AXI bus integration within the larger platform. We do not know whether AMD can update the platform so that MMIO routing does not take precedence, especially if the PSP explicitly requests DRAM. In particular, it is unclear whether this is possible with firmware updates alone or if it even requires hardware updates. As we do not have access to the HDL code from AMD, we can only hypothesize about the difficulty of implementing this mitigation.

Disallow Asynchronous Routing. A firmware-only mitigation is to force a synchronous configuration of individual Data Fabric blocks. Or in other words, prohibit asynchronous routing on the Data Fabric. Like the broadcast read, the Data Fabric implements a *broadcast write* functionality that sets the configuration registers on all Data Fabric blocks. The PSP could perform a synchronous read of all Data Fabric blocks and, in a second step, perform a synchronous write to all Data Fabric components for each register. This would ensure that all Data Fabric routing elements have the same memory view. If all Data Fabric routing elements have the same view, the PSP memory map reflects the actual routing configuration, and thus Mior would no longer be possible. We note that this may

limit functionality since there may be cases where AMD customers require asynchronous routing. We did not find this to be the case for our `ASRock Rack BERGAMOD8-2L2T` motherboard.

Reflect Asynchronous Routing in the Memory Map. Another approach to handling these asynchronous configurations across the different Data Fabric blocks is to modify the memory map. Instead of building the OR-sum of all Data Fabric units, the PSP builds the map, such that different configurations are correctly represented. This would, if implemented correctly, defend against these types of attacks against SEV-SNP by simply not accepting any memory in such a misrouted range. Implementing this correctly could also be error-prone due to the number of edge cases to consider. This is especially true for AMD's latest extension, Trusted I/O (SEV-TIO), which supports secure MMIO for accelerators [7].

Abandon DRAM as Secure Storage. A somewhat orthogonal solution may be to remove or reduce the PSP reliance on interacting with DRAM. For previous versions of SEV, for example SEV-ES, which does not have any integrity guarantees, the PSP would store guest metadata in a dedicated secure area of DRAM. Such areas are, however, still vulnerable to Mior-style attacks. Moving such data structures, for example, into the PSP secure RAM, could reduce susceptibility to malicious routing-type attacks. Given that the RMP, however, must be accessible from the core, at least for the trusted μ-code, implementing such an approach for SEV-SNP may also be impossible without fundamental architectural changes. It is also not fully clear whether the PSP in its current iteration has sufficient RAM available to move all of this functionality there.

7.3 Impact on Intel TDX and Arm CCA

While Mior focuses on AMD's architecture, secure memory initialization and asynchronous routing configuration may also face an issue for other TEE architectures. As with AMD, Intel treats the BIOS as untrusted in the confidential computing threat model. Instead of having a dedicated security co-processor validating the memory map, Intel uses *Authenticated Code Modules* (ACMs), BIOS code securely signed by Intel, and mcheck. These microcode-assisted modules are signed by Intel and validate the platform configuration. As Intel is closed source and Arm CCA is not yet commercially available, we did not study them. Nevertheless, Intel has internally found a TDX-compromising vulnerability within the memory routing configuration code [20]. We therefore conclude that this type of attack also applies to other TEE architectures. As Arm CCA and RISC-V CoVE are not yet fully specified and commercially available, we cannot definitively conclude on their susceptibility.

8 Related Work

We discuss Mior in relation to other work in the area of confidential computing and TEE research.

Legacy SEV and SEV-ES. The shortcomings of earlier iterations of AMD's confidential computing architecture were extensively studied. SEVered demonstrates how a malicious hypervisor can extract the full memory of an SEV-encrypted VM in plaintext [28]. Morbitzer et al. further generalized these methodologies in subsequent work on secret extraction [27]. Hetzelt and Buhren conducted a comprehensive security analysis highlighting that SEV lacks memory integrity protection, allowing malicious administrators to inject payloads and extract secrets [17]. Building upon this, SEVurity shows that this lack of integrity allows attackers to reuse existing ciphertexts to build a high-speed encryption oracle [43]. This enables arbitrary code injection even into SEV-ES secured VMs without relying on easily detectable I/O manipulation. Radev and Morbitzer analyzed vulnerabilities within the SEV management interfaces [30], while VIA focuses on attacks executed via device interfaces [18]. CrossLine exploits SEV and SEV-ES improper handling of Address Space Identifiers (ASIDs) for memory isolation [24]. Li et al. demonstrated TLB poisoning, showing that because the hypervisor controls Translation Lookaside Buffer (TLB) flushes, it can maliciously poison shared TLB entries to alter a victim process's execution flow [26]. Additionally, the PwrLeak side-channel attack proves that the power reporting interfaces available to the host can leak enough power-consumption data to compromise cryptographic secrets operating inside the SEV VM [41].

Vulnerabilities in SEV-SNP. SEV-SNP was a major architectural improvement over its predecessors, in particular due to its newly introduced memory integrity guarantees. Still, it was found to be vulnerable to a number of attacks. Following Google's foundational security review of the PSP and the broader SNP environment, researchers uncovered multiple avenues for exploitation [16]. One major class of attacks targets cryptographic leakage. Cipherleaks [25] demonstrated the first viable ciphertext side channel against SEV-SNP. Several further cryptographic side-channel attacks were subsequently identified in SEV-SNP that compromise guest confidentiality. These include Hypertheft and Ciphersteal [23, 45, 46]. Relocate+Vote and Heracles are further examples of cryptographic side channels against SEV-SNP [37, 44]. Shen et al. break SEV-SNP by compromising the SEV update mechanism using memory aliasing [38]. Another class of attacks involves the abuse of exposed architectural interfaces and state manipulation. Researchers showed that the guarantees of SEV-SNP can be thwarted by modifying unprotected MSRs to roll back cache lines and manipulate the stack with Cachewarp [47] and Stackwarp [48]. SEV-SNP has also been shown to be susceptible to interrupt injection attacks [35, 36]. Platform telemetry information, such as performance counters, can also be used to defeat SEV-SNP and exfiltrate secrets, as demonstrated by CounterSEVeillance [14]. System initialization and firmware integrity also present critical attack surfaces. BadAML [40] leverages unmeasured CVM BIOS firmware to execute malicious payloads. Finally, the architecture's lack of robust physical defenses enabled a series of hardware-centric attacks [10, 12, 13]. RMPocalypse abuses misaligned barriers during initialization to overwrite the RMP [34]. Like Mior, Fabricked and RMPocalypse also corrupt the RMP during initialization using their primitives [33, 34].

Interconnect Corruption Attacks. Mior is the fourth instance of attacks that break SEV-SNP by misconfiguring the interconnect. Fabricked abuses a missing lock of the Data Fabric routing registers to modify routing configuration at runtime and then redirects PSP RMP writes into the void [33]. In contrast, Mior shows a novel way to change the Data Fabric routing before locking by manipulating the PSP memory map, as we reverse-engineer (see Section 4). Breakfast, similarly, reroutes PSP memory accesses to escalate control plane privileges to the level of the PSP [15]. Staleus misconfigures

the interconnect to disable memory coherency between the PSP and DRAM, allowing it to overwrite guest metadata [32]. We hope that these four attacks bring further attention to interconnects as a critical component when reasoning about the security guarantees in confidential computing.

Does Mior Arise due to Poor Patching of Fabricked? During our testing, we found that AMD now locks the Data Fabric, thus mitigating Fabricked. Fabricked further documents that, although the PSP explicitly designates the target address as DRAM when accessing memory, the Data Fabric routing prioritizes the MMIO configuration over this designation. Our experiments corroborate that the prioritization behavior is present even after Fabricked was patched. However, when we attempted to misuse this behavior to drop RMP writes in Mior, some other checks detected it and stopped the malicious SEV initialization. To identify if these checks were introduced to mitigate Fabricked, we tested Mior on older SEV firmware (version 1.55.44, before the Fabricked [33] mitigation) and confirmed that our attack also succeeds there. Hence, we conclude that the flawed broadcast read logic we identified in Mior is an unrelated vulnerability to Fabricked and its mitigation. In our test against a previous firmware version, we also note that Mior succeeds even without re-routing the writes to an actual device. We therefore hypothesize that AMD added a read-back check that detects drops of RMP writes. They might have added this as a part of a new feature or as a mitigation for Fabricked, or to patch some other vulnerability. Irrespective of the reason why AMD added the check, in the context of Mior, it is simply an obstacle in exploiting the flawed broadcast read logic. As Mior introduces the Device-as-Memory primitive which allows us to fully control the PSP reads and writes (see Section 5), we can trivially bypass this obstacle.

Intel TDX. Similarly to SEV-SNP, researchers also analyzed the security of TDX [21]. Two security reports from Google uncovered multiple issues in TDX [3, 39]. Heckler abuses the interrupt injection interface to break TDX CVMs [36]. TDXdown and TDXploit abuse a flaw in Intel single-stepping protection to re-enable single-stepping and leak confidential data [31, 42]. TDXray combines known side channels on TDX to leak tokens from a TDX-shielded LLM application [19].

9 Conclusion

We present Mior, an attack to break confidential computing offered by AMD in the form of SEV-SNP. Mior can be performed by a software-only adversary, is deterministic, and does not require any profiling of victim workloads executing in the confidential VMs. The attack stems from incorrect checks in the PSP logic that are in place to protect the PSP from mistakenly accessing attacker-controlled MMIO regions. More broadly, Mior draws attention to AMD Infinity Fabric in the context of the confidential computing threat model. Our reverse engineering motivates the need to further analyze the assumptions pertaining to routing between AMD SoC components and systematically address the gaps.

References

[1] Advanced Micro Devices, Inc. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity protection and more. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. Accessed 2026-04-29.

[2] Advanced Micro Devices, Inc. 2020. *Processor Programming Reference (PPR) for AMD Family 1Ah Model 70h, Revision A0 Processors (PUB) (57930).* Technical Report. Accessed 2026-04-29.

[3] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. 2023. *Intel Trust Domain Extensions (TDX) Security Review.* Technical Report. Google. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf Accessed 2026-04-29.

[4] AMD. 2023. AMD-ASPFW. https://github.com/amd/AMD-ASPFW. Accessed 2026-04-29.

[5] AMD. 2025. openSIL: Open-Source Silicon Initialization Library. https://github.com/openSIL/openSIL. Accessed 2026-04-29.

[6] AMD. 2025. SEV Secure Nested Paging Firmware ABI Specification (56860, Rev 1.58).

[7] AMD. 2025. SEV-TIO Firmware Interface Specification (58271).

[8] AMD. 2026. AMD "Zen" Core Architecture. https://www.amd.com/en/technologies/zen-core.html. Accessed 2026-04-30.

[9] ARM. 2026. Arm Confidential Compute Architecture ARM-CCA. https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture. Accessed 2026-04-30.

[10] Jalen Chuang, Alex Seto, Nicolas Berrios, Stephan van Schaik, Christina Garman, and Daniel Genkin. 2026. TEE.fail: Breaking Trusted Execution Environments via DDR5 Memory Bus Interposition. In *IEEE S&P '26.*

[11] coreboot Contributors. 2019. *AMD Family 17h in coreboot.* coreboot. Accessed 2026-04-29.

[12] Jesse De Meulemeester, David Oswald, Ingrid Verbauwhede, and Jo Van Bulck. 2026. Battering RAM: Low-Cost Interposer Attacks on Confidential Computing via Dynamic Memory Aliasing. In *IEEE S&P '26.*

[13] 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 *IEEE S&P '25.*

[14] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. 2025. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In *ISOC NDSS '25.*

[15] Philipp Giersfeld, Benedict Schlüter, and Shweta Shinde. 2026. BreakFAST: Confused Deputy Attack on Infinity Fabric to Break AMD SEV-SNP. In *IEEE S&P '26.*

[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. Accessed 2026-04-29.

[17] Felicitas Hetzelt and Robert Buhren. 2017. Security Analysis of Encrypted Virtual Machines. In *ACM VEE '17.*

[18] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. 2021. VIA: Analyzing Device Interfaces of Protected Virtual Machines. In *ACM ACSAC '21.*

[19] Tristan Hornetz, Hosein Yavarzadeh, Albert Cheu, Adria Gascon, Lukas Gerlach, Daniel Moghimi, Phillipp Schoppmann, Michael Schwarz, and Ruiyi Zhang. 2026. TDXRay: Microarchitectural Side-Channel Analysis of Intel TDX for Real-World Workloads. In *IEEE S&P '26.*

[20] Intel. 2024. Trusted Execution Configuration Register Access. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/trusted-execution-configuration-register-access.html. Accessed 2026-04-29.

[21] Intel. 2025. Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html. Accessed 2026-04-30.

[22] LambdaConcept. 2022. LambdaConcept Screamer Documentation. https://docs.lambdaconcept.com/screamer/index.html. Accessed 2026-04-29.

[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 *IEEE S&P '22.*

[24] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2021. CrossLine: Breaking "Security-by-Crash" Based Memory Isolation in AMD SEV. In *ACM CCS '21.*

[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 *USENIX SEC '21.*

[26] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *ACM ACSAC '21.*

[27] Mathias Morbitzer, Manuel Huber, and Julian Horsch. 2019. Extracting Secrets from Encrypted Virtual Machines. In *ACM CODASPY '19.*

[28] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *ACM EuroSec '18.*

[29] PCI-SIG. 2019. PCI Express Base Specification Revision 5.0, Version 1.0 . https://pcisig.com/PCIExpress/Specs/Base/_5.0_1.0. Accessed 2026-04-30.

[30] Martin Radev and Mathias Morbitzer. 2021. Exploiting Interfaces of Secure Encrypted Virtual Machines. In *ACM ROOTS '20.*

[31] Fabian Rauscher, Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss. 2025. TDXploit: novel techniques for single-stepping and cache attacks on intel TDX. In *USENIX SEC '25.*

[32] Benedict Schlüter and Shweta Shinde. 2026. STALEUS: Breaking AMD SEV-SNP via Memory Incoherence. In *USENIX SEC '26*.
[33] Benedict Schlüter, Christoph Wech, and Shweta Shinde. 2026. FABRICKED: Misconfiguring Infinity Fabric to Break AMD SEV-SNP. In *USENIX SEC '26*.
[34] Benedict Schlüter and Shweta Shinde. 2025. RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP. In *ACM CCS '25*.
[35] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. 2024. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *IEEE S&P '24*.
[36] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. Heckler: Breaking Confidential VMs with Malicious Interrupts. In *USENIX SEC '24*.
[37] Benedict Schlüter, Christoph Wech, and Shweta Shinde. 2025. Heracles: Chosen Plaintext Attack on AMD SEV-SNP. In *ACM CCS '25*.
[38] Muyan Shen, Hongzhan Ma, Ketong Shang, Ruofei Qu, Yu Qin, and Dengguo Feng. 2026. Jailbreaking the AMD Secure Processor: Enabling Live Analysis of SEV-SNP's Undocumented Security Boundaries. In *(USENIX SEC '26)*.
[39] Kirk Swidowski, Daniel Moghimi, Josh Eads, Erdem Aktas, and Jia Ma. 2026. *Security Assessment of Intel TDX with support for Live Migration (Intel TDX 1.5 Full Report)*. Technical Report. Google. https://services.google.com/fh/files/misc/intel_tdx_1.5-full_report.pdf Accessed 2026-04-29.
[40] Satoru Takekoshi, Manami Mori, Takaaki Fukai, and Takahiro Shinagawa. 2025. BadAML: Exploiting Legacy Firmware Interfaces to Compromise Confidential Virtual Machines. In *ACM CCS '25*.
[41] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2023. PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV. In *DIMVA '23*.
[42] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. 2024. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In *ACM CCS '24*.
[43] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *IEEE S&P '20*.
[44] Yuqin Yan, Wei Huang, Ilya Grishchenko, Gururaj Saileshwar, Aastha Mehta, and David Lie. 2025. Relocate-Vote: Using Sparsity Information to Exploit Ciphertext Side-Channels. In *USENIX SEC '25*.
[45] 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 *ACM CCS '24*.
[46] 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 *IEEE S&P '25*.
[47] 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. In *USENIX SEC '24*.
[48] Ruiyi Zhang, Tristan Hornetz, Daniel Weber, Fabian Thomas, and Michael Schwarz. 2026. StackWarp: Breaking AMD SEV-SNP Integrity via Deterministic Stack-Pointer Manipulation through the CPU's Stack Engine. In *USENIX SEC '26*.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions, which helped improve the quality of this paper. Thanks to Reviewer A for encouraging us to demonstrate that Mior breaks AMD SEV-SNP even when ciphertext hiding is enabled. Benedict Schlüter is supported by the Google PhD Fellowship in Privacy, Safety, and Security. The authors used Anthropic Claude, OpenAI Codex, and Google Gemini for spelling and grammar checks. All edits were made and checked by the authors.

Open Science

We make the artifacts required to reproduce our results public.

Similar to Fabricked, this attack requires modification of the motherboard BIOS, which follows the same process as Fabricked. We make the BIOS for our motherboard, an `ASRock Rack BERGAMOD8-2L2T` motherboard, available as a binary blob, as well as instructions on how to recreate this for other motherboards. The result of these modifications is a BIOS that does not call upon the PSP to lock the Data Fabric configuration registers, leaving the fabric unlocked for modifications. We furthermore provide patches for our Linux host kernel, which perform the misconfiguration, as well as implementing the required functionality to then lock the Data Fabric. Finally, we provide Linux kernel modules and documentation that perform the attacks outlined by the case studies given. These artifacts will be available at https://github.com/xca-attacks/mior after the end of the embargo on November 10$^{\text{th}}$, 2026.

Ethical Considerations

Our paper details an attack that may impact the security of users of this technology. We therefore disclosed the attack to AMD on February 12$^{\text{th}}$, 2026 and agreed upon an embargo date of November 10$^{\text{th}}$, 2026. In the following, we detail our ethical considerations regarding the release of this work.

Stakeholders. We identify the key stakeholders affected by this attack as AMD, cloud service providers, motherboard vendors, customers, and end users.

Potential Harm. The potential harm arising from the disclosure of this attack is that sensitive data may be exfiltrated by an adversary. We acknowledge that disclosure of such attacks within the confidential computing threat model remains challenging, as the vendor must inform cloud service providers first, who are, however, assumed to be untrusted under this threat model.

Mitigations. To mitigate potential negative effects, we have taken the following steps. We identify AMD as the vendor of the technology under attack and consider AMD, as the owner of the technology, to be best positioned to coordinate further disclosure with affected stakeholders, as well as to oversee the rollout of mitigations. AMD acknowledged the attack, and we agreed on a coordinated disclosure date of November 10$^{\text{th}}$, 2026. We consider AMD to be responsible for any further communication and have therefore refrained from disclosing this attack to any other stakeholder prior to the end of the embargo. AMD also confirmed that a mitigation for this attack would be issued in an update before the end of the embargo.

Decision. We have weighed the potential harm of disclosing this attack against the benefit of making such security research public. After careful consideration—in particular given that the vulnerability will be mitigated prior to publication—we decided to proceed with this research, as we believe the insights it provides will contribute to improved security of SEV-SNP specifically, and confidential computing architectures in general.