ARTIFACT
EVALUATED
USENIX
AVAILABLE

FABRICKED: Misconfiguring Infinity Fabric to Break AMD SEV-SNP

Benedict Schlüter* Christoph Wech* Shweta Shinde
ETH Zurich

Abstract

Confidential computing is gaining popularity with real world cloud deployments. We present FABRICKED, a new attack that manipulates fabric routing to compromise AMD SEV-SNP. FABRICKED shows that a software adversary, by redirecting interconnect transactions, can mislead the secure co-processor to falsely initialize the system. Our primitive obtains arbitrary read and write within the victim address space, thereby violating confidentiality and integrity guarantees. FABRICKED also forges attestation reports.

1 Introduction

With growing reliance on large-scale and on-demand resources for computing, it has become commonplace to shift sensitive computation and data from on-prem infrastructure to the cloud. This choice comes with added trust assumptions on the cloud service provider's software and management (e.g., platform components, bios firmware, host hypervisor). To address this threat, confidential cloud computing has emerged as a promising solution, where the cloud tenant can offload computation to remote cloud resources without having to trust the cloud service provider. Hardware-based trusted execution environments are the cornerstone of achieving this goal while being able to scale to resource-intensive workloads. Specifically, hardware extensions from all major vendors allow the creation of confidential virtual machines, or CVMs for short. While Intel TDX and AMD SEV-SNP are available in production CPUs and offered on commercial cloud services, Arm CCA and RISC-V CoVE are specified and Arm CCA will soon be deployed in production [3, 13, 15, 41, 48, 55, 60].

These hardware extensions ensure that no software entity other than the owner CVM can access its code and data, including privileged host hypervisor and platform firmware.

Modern processors increasingly adopt chiplet-based architectures to improve yield. Unlike monolithic SoCs that integrate cores, caches, and memory controllers on a single die, leading to poor yield at advanced nodes, chiplets partition functionality across smaller dies, assembled using high-speed interconnects [25, 36]. AMD *Infinity Fabric* and Intel Mesh architecture provide such connectivity between the chiplets while ensuring coherence, high bandwidth, and low latency [20, 38, 40, 53]. Figure 1 shows a simplified overview of AMD Infinity Fabric in an SoC. Within Infinity Fabric, the Data Fabric is responsible for memory and I/O coherency, address mapping, and routing between CPU cores, memory controllers, and peripheral devices [20]. Data Fabric must be configured during the platform initialization phase to enforce routing policies such as mapping data from x86 cores to specific address spaces, controlling device access, and managing aliasing. Since the platform vendor has knowledge of the specific motherboard topology and connected components, the platform firmware initializes the Data Fabric configuration. Specifically, during the boot process, UEFI firmware programs the Data Fabric by writing to hardware configuration registers to establish routing and access-control rules.

In the confidential computing threat model, neither the motherboard vendor not the cloud service provider, is trusted to configure the Data Fabric correctly. Both Intel and AMD ship production hardware support for confidential computing

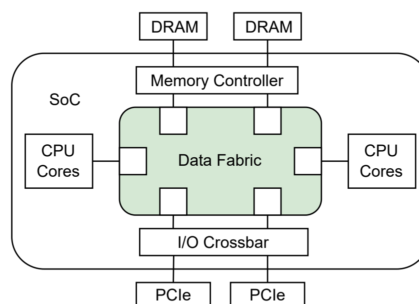

Figure 1: Simplified overview of an AMD EPYC platform. The Data Fabric connects chiplets (e.g., CPU cores, memory controller, I/O crossbar) inside the SoC, which then interfaces externally, e.g., with PCIe devices and DRAM.

* equal contribution

and explicitly state that the BIOS / UEFI components provided by the motherboard vendor are outside the trusted computing base. Intel provides platform-specific signed trusted software blobs to support Scalable SGX and TDX on various motherboards [31]. AMD takes a different approach: it allows the motherboard vendor to almost freely program certain Data Fabric registers, but checks the initial configuration via a secure processor, called PSP, before initializing SEV-SNP on the platform. Either way, both Intel and AMD check that the interconnect is correctly configured because any misconfiguration can potentially break the confidential computing guarantees of TDX and SEV-SNP. For example, if an attacker can misconfigure the Data Fabric to drop writes from PSP to DRAM or corrupt reads from DRAM, it can subvert the PSP security enforcement and break SEV-SNP.

In this paper, we examine the impact of misconfiguring the Data Fabric on AMD SEV-SNP. As a first step, we analyze AMD defenses to detect malicious Infinity Fabric configurations. Analysis on Intel quickly turns out to be infeasible, as the checking mechanism is proprietary and undocumented [31, 37]. We shift focus to AMD, where, fortunately, we find promising avenues for systematic analysis, thanks to documentation, open-source reference implementation, and feasibility of binary analysis [2, 5, 6, 8]. Specifically, we reverse engineer the untrusted modules of platform firmware binaries provided by the motherboard vendor (shown inside the red box in Figure 2) to identify if and how they configure the Data Fabric. Then we perform controlled experiments by modifying these binaries to study the effect of malicious Data Fabric configuration. In particular, our goal is to assess if AMD PSP detects this behavior. As the first result, we show that malicious platform firmware can corrupt the Data Fabric MMIO routing rules and can change them during runtime.

While the above finding may sound promising, we cannot use it directly to break SEV-SNP. This is because neither the PSP nor the x86 cores perform any security-critical MMIO read or writes during the SEV-SNP lifecycle. Thus, even if an adversary can corrupt MMIO routing rules, it has no avenue to use it for exploitation. However, during our experiments, we discover that the Data Fabric routing deviates from the expected behavior per the documentation. Instead of routing PSP DRAM memory requests according to DRAM routing rules, the Data Fabric first checks if the request matches any MMIO routing registers, and if so, the Data Fabric routes the traffic based on the MMIO rules. With this second result, we can maliciously redirect PSP intended DRAM accesses, of which there are many, to MMIO memory.

We concretely demonstrate the impact of our findings in form of FABRICKED attack on AMD SEV-SNP. We identify that the reverse map table (RMP), which is a data structure that protects SEV-specific metadata on the DRAM, starts with zeroed-out memory. During SEV-SNP initialization, there are critical writes from the PSP to the DRAM to setup the RMP. By misrouting these writes, we corrupt the initial RMP such that it fails to stop malicious accesses from the untrusted hypervisor to the CVM memory on the DRAM. Concretely, we showcase attestation forgery and debug-mode CVM execution, which goes undetected by all mechanisms of SEV-SNP. We outline two mitigations based on our root-cause analysis.

We make the following novel contributions:

1. We present the first study on the security implications of Infinity Fabric routing on AMD SEV-SNP.

2. We identify that the untrusted platform software, UEFI, can corrupt Data Fabric MMIO routing rules and these take precedence over uncorrupted DRAM routing rules.

3. FABRICKED uses these findings to misroute writes to corrupt a critical SEV-SNP in-memory data structure. Our exploits can forge attestation reports and enable debug during victim VM execution.

Responsible Disclosure. We responsibly disclosed our findings to AMD in August 2025. AMD acknowledged the vulnerability, thanked us for the disclosure, and plans to issue CVE-2025-54510. Our code is public at https://fabricked-attack.github.io.

2 Background

We explain concepts and terminology used in FABRICKED.

2.1 Confidential Computing on AMD

AMD confidential computing solution isolates VMs from the untrusted hypervisor. The most recent iteration of this technology is AMD Secure Encrypted Virtual-Machine Secure Nested Paging, SEV-SNP in short. SEV-SNP enables the creation of Confidential Virtual Machines (CVMs), where the hardware encrypts CVM memory such that the hypervisor cannot observe or tamper with the CVM. SEV-SNP ensures the integrity of CVM memory by introducing a Reverse Map (RMP) table. The RMP-Table is a structure in main memory holding the security attributes of each 4KiB page. When any entity on the system accesses main memory, a hardware unit consults the RMP and checks whether the access is allowed.

2.2 Platform Security Processor

The PSP is a dedicated co-processor whose firmware is exclusively controlled by AMD and acts as the root of trust on modern AMD platforms. It is an Arm Cortex A5 processor integrated into the chip die. It is known as the Platform Security Processor (PSP), Application Security Processor (ASP), or just Security Processor (SP). The SEV-SNP documentation refers to it as PSP; therefore, we follow the same convention [12]. The Arm Cortex A5 core has an internal RAM for its operating system and applications. The PSP uses the

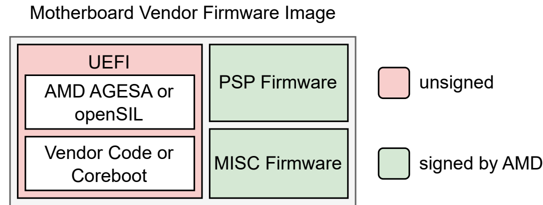

Figure 2: Simplified firmware image composition. In the AMD SEV-SNP threat model, the UEFI is untrusted and not verified by any instance.

AXI bus protocol to map x86 DRAM into the PSP's address space [23]. PSP interacts with the x86 cores to perform management tasks: verify and enforce security policies required by SEV-SNP, execute all security-sensitive SEV-SNP lifecycle operations (RMP initialization, CVM creation and destruction). The PSP also exposes APIs, through a set of MMIO mailbox registers, such that the untrusted hypervisor can communicate requests to the PSP (e.g., request the attestation report). PSP firmware, shown in Figure 2, is developed and signed by AMD before being shipped as part of the motherboard vendor firmware image alongside UEFI.

2.3 Unified Extensible Firmware Interface

The Unified Extensible Firmware Interface (UEFI) is designed to provide a versatile and modular framework for platform initialization, replacing the aging BIOS. Originally designed for Itanium-based systems, it has since become the de facto standard in modern x86 computer hardware [69, 70, 80]. UEFI can be seen as a file format similar to Executable and Linkable Format (ELF). Once the UEFI finishes executing, it passes control to the operating system boot loader (e.g., GRUB). UEFI implementations for AMD platforms consist of two parts, one part supplied by AMD and the other one by the respective motherboard manufacturer [2]. Figure 2 shows a simplified composition of a firmware image.

AMD develops and provides AGESA software, which stands for AMD Generic Encapsulated Software Architecture, and contains code to perform platform-specific low-level initialization (e.g., initialize DRAM, configure memory routing). AGESA is shipped in binary blobs that export global functions to motherboard vendors for integration in their board-specific UEFI implementations. In early 2023, AMD announced to replace AGESA in the long term with openSIL [2]. It has the same functionality as AGESA but is fully open source. openSIL details how the CPU initializes modern AMD chipsets. While openSIL greatly improves the understanding of the AMD platform, some key aspects are omitted within the code (e.g., DRAM initialization). AGESA/openSIL are part of UEFI.

The second half, i.e., code from the motherboard manufacturers, executes more generic initialization tasks (e.g., registers system management mode handlers) and calls AGESA functions. Open source projects, such as coreboot, aim to replace the code provided by the motherboard vendors as part of open UEFI efforts [1].

2.4 Infinity Fabric

AMD introduced the Zen architecture in 2017 [67], which was the first to use the chiplet design [20]. Instead of fabricating everything on a single die, AMD uses multiple dies and connects them to enhance the yield [53]. The Infinity Fabric is a proprietary on-chip interconnect from AMD that connects CPU core dies with on-platform devices and memory controllers. It consists of a Control Fabric (CF) and Data Fabric (DF). Data Fabric is responsible for coherent data transport between different components [20], it is not an active computing component but is integrated into the system. During platform boot, the Data Fabric must be configured and adjusted to reflect the current system configuration and connected devices. For instance, different PCIe devices request different amounts of MMIO regions, and the platform must be able to accommodate these configurations. The details of the configuration process are not publicly documented. The platform initializes the Data Fabric during boot such that it routes memory requests to either DRAM or platform devices with exposed memory regions. The Data Fabric distinguishes between MMIO and DRAM when routing memory, as these have two different destination IP blocks. The distinction is vital since wrongly routing memory requests would induce security-relevant effects for SEV-SNP.

3 FABRICKED Overview

We explain how we maliciously reconfigure the Data Fabric.

Threat Model. We operate in the AMD SEV-SNP threat model [3]. Only hardware, microcode, PSP firmware, and the CVM image are considered trusted. AMD documentation explicitly states that other firmware, such as the UEFI controlled by the motherboard manufacturer/hypervisor, is untrusted. We assume a software-only attacker who controls the hypervisor and the UEFI. We do not assume physical access to the platform. Lastly, we assume that the attacker cannot single-step the CVM.

3.1 Boot Process

Figure 3 summarizes the boot flow. When the platform boots, the PSP is the first unit to start before any x86 cores. The PSP performs the necessary platform initialization such that x86 cores can communicate with DRAM. Once the PSP finishes memory controller and routing initialization, it starts one of the x86 cores to execute the untrusted UEFI firmware

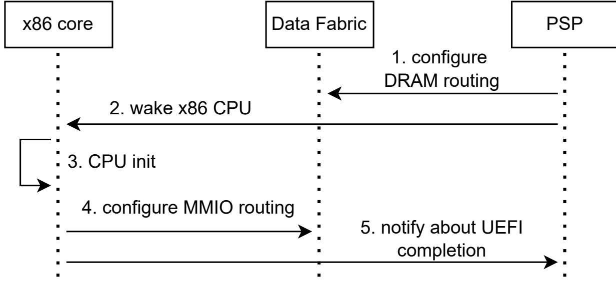

Figure 3: Data Fabric initialization at platform boot.

code provided by the motherboard manufacturer [11]. Subsequently, the x86 core initializes the remaining CPU cores and configures the remaining registers of the Data Fabric. Notably, the x86 core initializes the Data Fabric MMIO routing, while the PSP initializes DRAM routing as we experimentally confirm in Section 6. Once the UEFI finishes the basic initialization, it informs the PSP about completion and hands over control to the bootloader.

3.2 Attack

A malicious UEFI can program routing registers in the Data Fabric. Misconfigured DRAM routing definitely breaks SEV-SNP because it allows redirecting and discarding arbitrary DRAM writes not only by the x86 cores but also by the PSP. Our experiments show that a malicious UEFI cannot tamper with the DRAM routing rules because the registers are write-protected. However, we make two important observations. First, as we will explain in Section 4, a malicious UEFI can tamper with the MMIO routing registers. Modifying the MMIO routing registers alone should not be a problem in itself, as SEV-SNP does not use MMIO memory for any security-relevant tasks in its lifecycle. Second, and more importantly, MMIO routing configuration impacts DRAM routing. We will explain this in detail in Section 6. Figure 4 shows the benign memory routing and maliciously altered routing. By virtue of our two observations combined, we corrupt MMIO routing to redirect PSP writes intended for DRAM to I/O. The PSP performs critical write operations, especially when initializing SEV-SNP by setting up the RMP. As we will explain in Section 7, by targeting these writes for malicious routing, we can corrupt the RMP, which then impacts attestation and debug configurations of the CVMs. In this way, we show that a malicious UEFI can misconfigure Data Fabric to break SEV-SNP.

4 UEFI Experiments

This section describes the experiments we conduct to analyze the Data Fabric initialization capabilities of the UEFI.

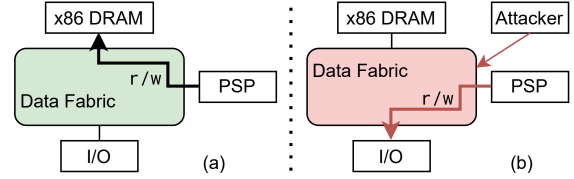

Figure 4: (a) benign access from PSP to x86 DRAM. (b) maliciously configured Data Fabric redirects access.

4.1 UEFI Execution Stages

The UEFI firmware hosts assembly code in modules separated into three main phases: (1) SEC, (2) Pre EFI Initialization (PEI), (3) Driver Execution Environment (DXE). The SEC and PEI modules perform basic initialization of the platform state (e.g., initialize the memory) [70]. After the SEC and PEI phases, the DXE phase initializes non-essential platform devices (e.g., SATA, NVME, BMC). Via static reverse engineering, we verify the presence of Data Fabric configuration functions in the PEI and DXE phases (Appendix B).

4.2 Firmware Modification

As a first step, we verify that the UEFI is not signed, and we can boot with a modified UEFI. Specifically, we want to confirm that the SEV-SNP feature activates after we flash a modified firmware image on our motherboard.

```
1 outl(0x80, 0xBA);
2 for(uint64 i; i<400000; i++){;}
3 outl(0x80, 0xB0);
4 for(uint64 i; i<400000; i++){;}
```

Listing 1: Custom UEFI module

Listing 1 shows the code we compile into a UEFI module. We use an old version of UEFITool [62] to inspect and inject the modules, since newer versions do not allow injecting new modules. We attempt to insert a PEI module into the firmware volume. However, the insertion changes the location of existing modules in the firmware volume, resulting in the system not booting and erroring with post code `0x05`. Since debugging UEFI errors without a proper hardware debugger is extremely time-consuming and error-prone, we circumvent the issue by replacing an existing module. Specifically, by examining the UI Section strings, we identify modules that are non-critical to the boot process and can be easily replaced. We replace the `AmiTpm20PlatformPei` module, as our platform does not use any TPM functionality. Flashing the modified UEFI onto the SPI chip works without errors. During platform boot, we do not observe any visible side effects that prevent the system from booting. Most importantly, SEV-SNP activates when the system boots our custom firmware.

To verify the execution of our module, we target the `DR Debug` display of our motherboard. `DR Debug` is an LED display that shows the last two hex digits of the current POST

code output. We use the `out` instruction on x86 with port `0x80` as shown in Listing 1. The `DR Debug` device displays the least significant byte of the POST code, which is the least significant byte we send to port `0x80`. To ensure the byte is visible and not overwritten with a different POST code shortly after, we insert a busy loop after each write, ensuring the byte is observable by the human eye. When flashing the firmware with our custom module and booting the system, we can indeed see the custom POST code sequence `0xBA 0xB0`, which experimentally confirms that the CPU executes our code.

4.3 Register Write Protection

Writes from the x86 host operating system to Data Fabric registers that configure DRAM or MMIO routing fail silently. The move instruction executes successfully, but the register does not contain the written value. Based on the openSIL code and our reverse engineering (Appendix B), we see UEFI writes to Data Fabric registers. We suspect that the registers are locked at some point during the boot process, prohibiting modification through the operating system. We aim to identify the locking procedure within our closed-source UEFI.

First, we verify that the UEFI Data Fabric writes update the Data Fabric register. We insert a simple PEI module to read and write to a Data Fabric MMIO routing register, immediately read it back, and test if the value changes. We output the POST code sequence `0xBA; val1 == val2; 0xB0` when performing the overwrite, so `0xBA; 0x00; 0xB0` indicates the overwrite failed, and a non-zero value means the overwrite succeeded. We observe that the MMIO register writes are, in fact, visible for a short period. However, when we read them back during operating system execution, our changes are not visible. We assume that this is the case because the register is overwritten by later modules in the PEI or DXE phase, or by the PSP. According to the openSIL codebase, it seems more likely that a later module overwrites the registers; we explore this possibility first.

We use automated reverse engineering techniques to identify code locations at which these registers may subsequently be written. UEFI executes respective modules in each phase in a non-predetermined order, which has to satisfy the dependencies between modules. We identify reads, and more crucially, writes in both the PEI and DXE sections using signature matching (See Appendix B.2 for details). We then use carefully crafted `Dependency Expressions`, part of UEFI firmware modules, to ensure our overwrite module is executed after every other module that performs writes. After booting the system with this configuration, we are able to read our changes from the operating system context. This confirms that we can overwrite the Data Fabric registers during the DXE phase, which shows that the registers are still writable. We therefore hypothesize that an operation or state change within the UEFI must be responsible for the Data Fabric lock.

ID	String
`0x5`	`"Psp.PspMboxBiosQueryCaps"`
`0x6`	`"Psp.C2PMbox.ExitBootServices"`
`0x18`	`"Psp.C2PMbox.LockDFReg"`
`0x34`	`"Psp.C2PMbox.SignValidateHmacDataPreSmm"`
`0x52`	`"Psp.C2PMbox.DeferredPsbFuse"`
`0x53`	Unnamed, referred to as `"Command 0x53"`
`0x5d`	`"Psp.PspMboxBiosCmdSetConfig"`
`0x61`	`"Psp.C2PMbox.ResetTpmEstablishment"`
`0x62`	`"Psp.C2PMbox.DisableDrtmCapability"`
`0x6c`	`"Psp.C2PMbox.SmmLock"`
`0x74`	`"Psp.C2PMbox.SendCoSignPublicKey"`

Table 1: PSP Command IDs present in `AmdPspDxeV2Bhr` with accompanying debug strings, and if they are exposed.

4.4 Finding the Data Fabric Lock

A comment within openSIL, in `xUSL/DF/DfX/SilFabricRegistersDfX.h:1193`, reveals that the PSP locks the Data Fabric registers by setting a bit, named `CfgRegInstAccRegLock`, in a Data Fabric register [11].

Thus, our focus shifts to search for mailbox interfaces between the UEFI code and the PSP. As the openSIL codebase does not contain a direct interface between the UEFI and PSP we shift our focus to coreboot. The coreboot project source code contains such an interface and also documents certain commands, including their meaning [1]. The code contains a macro definition, `MBOX_BIOS_CMD_BOOT_DONE`, which expands to the literal `0x06`. As an initial step, we search and identify the function handling the mailbox commands with the PSP in our closed-source firmware. We identify the `AmdPspDxeV2Bhr` module as a potential implementation of such functionality in our UEFI implementation by the contained debug strings. Using manual analysis of this module, we find a function potentially responsible for PSP mailbox operations.

By analyzing the callsites of this function within this module, we confirm that the UEFI uses command `0x06` as an argument to this function. The containing function also has the logging string `"Psp.C2PMbox.ExitBootServices"`, further confirming the use of this function. In analyzing other callers of the mailbox function, we also find an invocation with the constant `0x1b`, accompanied by the debug print `"Psp.C2PMbox.LockDFReg"`. Neither of these functions is invoked within `AmdPspDxeV2Bhr`, but rather exposed to the remaining DXE modules using protocols and events. Further, we analyze the binary for other security-relevant calls that the UEFI may perform with the PSP. Table 1 shows a list of all identified function calls with the linked debug string.

Next, we verify if the PSP command `0x1b` locks the Data Fabric as the debug string indicates. We modify the UEFI to simply omit the call to lock the registers, to see if we can

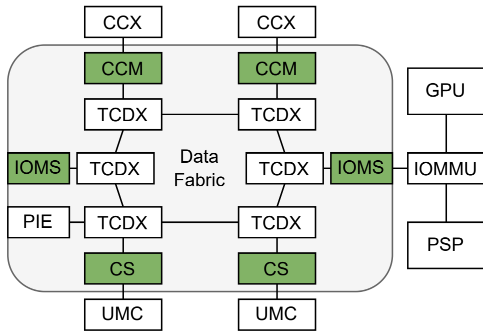

Figure 5: Overview of the Data Fabric and attached components. Green components have active routing registers.

continue to boot to the OS. Trying to boot with those changes results in the system automatically restarting just before invoking GRUB. We hypothesize that the `BOOT_DONE` call depends on the Data Fabric lock call being executed prior. When we omit both mailbox calls `0x1B` and `0x06`, our system boots into the OS with an unlocked Data Fabric. While our experiments confirm we can boot with an unlocked Data Fabric, its security implications are not yet clear. In the next two sections, we explore how an attacker can use an unlocked Data Fabric to circumvent AMD SEV-SNP security mechanisms.

5 AMD Platform Details

We introduce Data Fabric routing components, modern memory routing, and the Data Fabric configuration mechanism. Unless stated otherwise, we use AMD terminology.

5.1 Platform Units

The Data Fabric is the central crossbar in AMD Zen architecture, connecting major components. Figure 5 shows a simplified overview of each component and its connection to the Data Fabric [20]. We distinguish between active computing components sitting outside of the Data Fabric and routing and management components within.

5.1.1 Non Data Fabric

Non Data Fabric units are not parts of the Data Fabric configuration mechanism but are directly or indirectly connected to one of their units.

IOMMU: The I/O Memory Management Unit handles incoming memory transactions targeting the Data Fabric. The IOMMU can translate a virtual address to a physical address or directly pass through transactions. One IOMMU may connect multiple devices, as shown on the right in Figure 5.

CCX: Each Core Complex Die (CCX) hosts multiple x86 CPU cores and L1/L2/L3 caches. Since the CCX units are not within the Data Fabric, they interface with a CCM unit (explained soon), which acts as a gateway into the Data Fabric.

UMC: Unified Memory Controller (UMC) units interface to physical DRAM. Rather than the on-chip component, it configures the physical timings and DRAM chip properties.

5.1.2 Data Fabric

Data Fabric components sit within the Data Fabric and can be configured through Data Fabric registers.

TCDX: The Transport Command and Data Crossbar (TCDX) is the routing hardware unit. The Data Fabric routes memory requests through the TCDX blocks to the respective components. TCDX units are passive and do not have an individually controllable configuration on the Data Fabric.

CCM: Core Coherent Master units are the gateway for CCX units into the Data Fabric. CCX units host the x86 CPU cores and L1/L2/L3 caches. A CCM connects CCX units to the Data Fabric and routes the memory requests on behalf of the CCX. Thus, when configuring the Data Fabric routing for x86 cores, CCM instances contain the configuration details.

IOMS: I/O Master Slave units combine I/O Master and I/O Slave units into one. Master units are generally the origin of memory requests, whereas slave units are the recipients of those memory requests. Since IOMS units are the final gateway for PCIe devices into the Data Fabric, they can be the origin of a memory request (e.g, DMA) or the destination (e.g., PCIe configuration access). Memory requests enter the Data Fabric through IOMS or CCM units.

CS: Coherent Slave units are the endpoint for Data Fabric memory requests. CS units connect to Unified Memory Controllers (UMCs), which in turn interface with DRAM. Optionally, CS units can also connect to CXL devices, but CXL is unsupported with SEV-SNP.

PIE: The PIE unit receives memory requests targeting the Data Fabric configuration space. It performs the access checks and applies changes to Data Fabric registers. AMD does not document the internals of this unit, but routing registers and documentation reveal its existence. Appendix A has more details on the identification of this unit.

5.2 Memory Routing

x86 memory accesses distinguish between two memory types: DRAM and MMIO. DRAM memory accesses are routed to physical DRAM, whereas MMIO is routed to PCIe devices or other components that expose MMIO regions. x86 uses DRAM as general-purpose storage memory, whereas MMIO memory usually has devices or logic behind it that react in certain ways to the memory read or write. However, in the end, both memory types are just accesses in a contiguous physical memory space. So, how does a processor know where to route

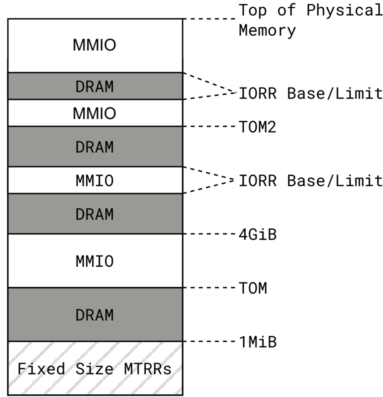

Figure 6: Memory routing configuration from an x86 core. Each of the registers is core exclusive.

the memory? To answer the question, we have to make two distinctions between x86 and Data Fabric memory routing.

x86 Memory View. x86 has an internal register priority which determines whether it requests MMIO or DRAM from the Data Fabric. The Preliminary Programming References (PPR), formerly also known as the Bios and Kernel Development Guide, outlines the memory hierarchy [5, 8]. If not otherwise specified, x86 issues memory requests to DRAM as the default type. Multiple MSRs shown in Figure 6 override the DRAM destination to MMIO or force it back from MMIO to DRAM. Namely, Top of Memory (TOM) and TOM2 MSRs have the lowest priority. TOM specifies a range starting below 4GiB, ranging to 4GiB, that resorts to MMIO. TOM2 specifies the upper physical address above which all accesses are MMIO. Followed by the I/O Range Registers (IORR), which define a range in which all memory accesses are either MMIO or DRAM, depending on a flag. The IORRs can be used twofold: either to shadow TOM registers and force DRAM access, or to force MMIO access over the default DRAM type. Next in the hierarchy are Fixed Size Memory Type Range Registers (MTRRs), which define the access destination (MMIO/DRAM) for memory below 1 MiB.

Data Fabric Memory View. The Data Fabric receives a memory request from its connected components (e.g., IOMMU / CCX) and routes them according to their requested type. It has 20 configuration register ranges for DRAM and 16 for MMIO. Each range consists of a base, a limit, a control, and an interleaving or extension register [11]. If a memory request entering the Data Fabric through one of the master units (IOMS or CCM) has a matching type, the Data Fabric routes it to the destination specified in the register associated with the base and limit. Figure 7 shows routing examples for MMIO and DRAM requests once they reach the Data Fabric.

DRAM and MMIO ranges may overlap with each other, but must be exclusive within their type. For instance, MMIO regions on the Data Fabric may overlap the DRAM routing

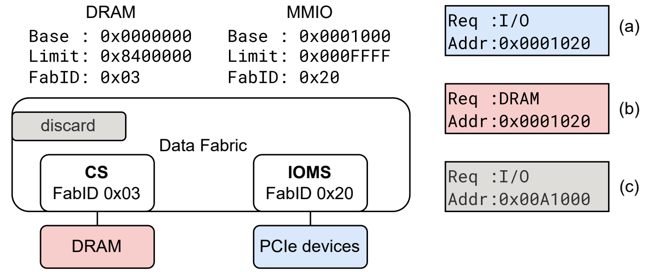

Figure 7: Simplified Data Fabric routing configuration. Request (a) request I/O and matches an MMIO register range. Request (b) requests DRAM and matches a DRAM routing range. Request (c) requests I/O but matches no MMIO routing ranges; writes are thus discarded and reads return `0xFF`.

configuration, but MMIO regions cannot overlap each other. The Data Fabric gets a request from one of the master units (CCM / IOMS) and routes the memory request based on the MMIO or DRAM registers, depending on the requester. Since there are multiple possible CS or IOMS units, the Data Fabric stores a destination fabric ID with each memory range.

5.3 Data Fabric Configuration

We reverse engineer where Data Fabric configuration registers are stored and which platform device manages its security. During operating system runtime, the platform exposes the Data Fabric configuration registers as a set of 8 PCIe functions. The PCIe devices have the IDs: bus `0x00`, device `0x18`, and a function number between `0x0` and `0x7` [8]. To read or write any Data Fabric configuration registers, the UEFI or operating system uses the PCIe configuration space. The Data Fabric routes the configuration space memory request to the PIE unit, which is responsible for access controls (see Appendix A for details).

When using the PCIe offset to write to the Data Fabric, the PIE unit applies the change globally to all Data Fabric components if they support the configuration. For example, if one changes the DRAM routing, all units receive the update to then have the same configuration. To support the individual configuration of different units, AMD introduced indirect access to the Data Fabric configuration registers. AMD implements indirect access through two register pairs: Fabric Indirect Configuration Access Data (FICAD) and the Fabric Indirect Configuration Access Address (FICAA) register. A third register, the Fabric Configuration Access Control (FCAC), exposes access control settings to the hypervisor [8, 11]. The FICAA register specifies the Data Fabric register and the corresponding unit. Subsequently, the DFC unit forwards hypervisor reads and writes to the FICAD register to the respective Data Fabric configuration register of the unit specified in the FICAA register. Configuring the Data Fabric routing with FICAA and FICAD access allows for asynchronous routing.

For example, one CCM block may route a MMIO memory request to the IOMS block 0x21 while another CCM routes the same address to block 0x22.

Since AMD uses the Data Fabric configuration space address registers to route traffic to the unit, which in turn is responsible for configuring the Data Fabric routing, it may seem that there is a bootstrap problem. However, documents reveal that the Data Fabric registers have a pre-defined value on startup, such that the Data Fabric is in a state where it can properly route configuration requests.

6 Data Fabric Experiments

Recall that we concluded at the end of Section 4 that a malicious hypervisor can write to the unlocked Data Fabric. We conduct experiments to understand the capabilities that this primitive brings, with an emphasis on routing registers. We experiment with DRAM and MMIO routing registers of the Data Fabric.

6.1 DRAM Registers

As an initial step, we analyze the implications of modifying DRAM routing through the Data Fabric. According to openSIL and the PPR, Zen 5 has 20 register configuration quadruples. Trying to write any of those registers fails, no matter at which state we write it. Specifically, we try to overwrite all of the DRAM configuration registers during the PEI phase and OS runtime. Since at no point in time are the DRAM registers writable, perhaps another unit on the system initializes them. The DRAM registers are influenced by the UEFI configuration (e.g., interleaving setting); thus, it requires more complex logic to initialize them. We hypothesize that the PSP initializes the DRAM routing before the x86 cores start and locks the Data Fabric DRAM registers such that the x86 cores cannot change them. This assumption is further strengthened by the POST code description of the PSP, which includes many strings related to DRAM initialization [9]. Thus, we do not pursue this line of inquiry any further.

6.2 MMIO Registers

As a first experiment, we want to confirm if changing the Data Fabric MMIO registers at runtime influences MMIO routing.

As a goal, we try to redirect the x86 core access intended for a USB controller BAR MMIO region. We read at the base of the USB controller address range, `0xf0100004`. The Data Fabric function 0 registers `0xd80-0xd8c` map access between base address `0x0000f000` and limit address `0x0000f6ff` to IOMS block `0x20`. From this configuration, we deduce that the USB controller must be connected to IOMS `0x20`. When reading the memory at address `0xf0100004`, the USB controller returns `0x400840`. We change the configuration such that MMIO access falling within the base and limit range is

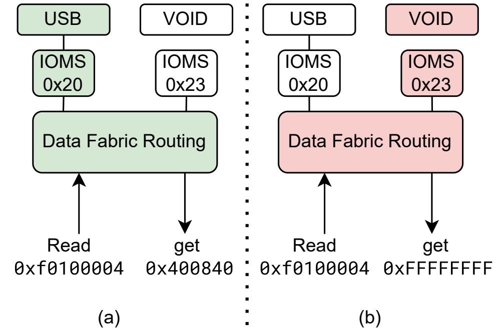

Figure 8: Data Fabric routing during the USB BAR experiment. (a) We normally read the USB BAR. (b) We change the Data Fabric to route the USB MMIO range to IOMS block `0x23` instead of block `0x20`.

never routed to IOMS block `0x20` but always to block `0x23`. We do not know which devices are behind block `0x23`, but the likelihood is high that they do not map the previous addresses since they are already taken. Experimentally, when we read the base of the USB controllers' address range after changing the routing, we always read `0xFFFFFFFF`, which is not the expected value. According to AMD documentation, `0xFFFFFFFF` is the default response if the routing logic does not find a device responding to the memory request. This confirms our hypothesis that if we write to the routing registers, they indeed influence Data Fabric MMIO routing, as depicted in Figure 8.

> **Observation 1:** Changes to the Data Fabric routing registers during runtime influence the platform routing.

In a second experiment, we try to shadow the memory requests of an x86 core. First, we write `0xDEADBEEF` from the x86 core at DRAM address 0x1000. After writing, we flush the caches with `wbinvd` and invalidate the `TLB`. Subsequently, we reconfigure MMIO register quadruple 15 to map the range 0x00000-0x10000 and read the same address we previously wrote. If the MMIO Data Fabric registers shadow the access DRAM, we would expect to receive `0xFF` as a response, otherwise `0xDEADBEEF`. When reading the memory after changing the Data Fabric register, we see the value `0xDEADBEEF`. This reveals that the Data Fabric MMIO routing registers do not influence the x86 core memory requests to x86 DRAM. Next, we try to force the x86 core to request MMIO routing instead of DRAM routing on the Data Fabric. This is not possible because AMD locks the x86 MMIO configuration MSRs once the SEV-SNP enable bit within the `SYSCFG` MSR is set [7].

> **Observation 2:** Data Fabric MMIO routing registers do not influence x86 DRAM memory requests originating from the x86 CPU cores.

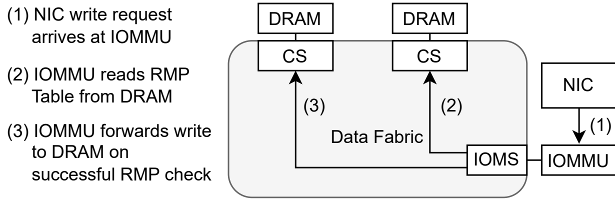

Figure 9: IOMMU RMP checks when a device requests DMA writes into memory.

In a third experiment, our goal was to route x86 MMIO traffic to DRAM by changing the destination fabric ID. openSIL documentation shows that the MMIO registers should only have an IOMS block as the destination fabric ID. However, writing a CS fabric ID as the destination fabric ID works, and we read the changed destination value back. But when we send an actual MMIO request that the Data Fabric routes to the CS unit (specifically DRAM), we observe a platform crash followed by a reboot. Recent Linux patches allow us to examine the reason for the previous platform reboot [27]. According to AMD patches, the reason for the crash is: `UNKNOWN` and `Data Fabric Sync Flood Event`, which we cannot debug further. We conclude that it is not possible to route MMIO traffic to DRAM using the Data Fabric registers. Further, the error codes do not have enough verbosity to further explore what happens on a platform level.

Observation 3: Traffic routed according to Data Fabric MMIO routing registers cannot have CS units (i.e., DRAM) as the destination.

Next to the x86 cores that interface through the CCM with the Data Fabric, there are also IOMS blocks with a master function on the fabric. IOMS blocks are the gateway for all kinds of external devices onto the Data Fabric routing (e.g., PSP, PCIe devices). PCIe devices generally do not have the notion of MSRs. Thus, they cannot specifically ask for MMIO or DRAM. IOMMU is situated between the PCIe devices and Data Fabric. It translates Direct Memory Access (DMA) requests from PCIe devices. In addition, the IOMMU reads the RMP-Table from DRAM and determines if an access is allowed. Figure 9 depicts the flow.

Given that the Data Fabric MMIO ranges already shadow the DRAM registers (Figure 7), we suspect that there may be an internal mechanism that routes PCIe non-explicit memory requests according to MMIO routing rules first, and then by DRAM routing rules. If true, Data Fabric to prioritize MMIO rules before DRAM for requests coming through IOMS. To test this, we change the Data Fabric MMIO routing register quadruple 15 to shadow the RMP-Table and route the memory accesses into a non-mapped region. Then, to test this configuration, we use the NIC to access DRAM. Since this process triggers the IOMMU RMP check, it needs to fetch the RMP-Table from the DRAM. Recall that reading from a non-mapped region returns just `0xFF` blocks, and writes are simply discarded. Going back to our NIC experiment with modified MMIO routing, we experience RMP page faults in the IOMMU. This confirms that the IOMS prioritizes MMIO rules and ends up reading `0xFF` instead of the RMP-Table entry from the DRAM. Notably, we were only experiencing RMP faults in the IOMMU and no RMP faults on the x86 cores. Thus, we conclude that the Data Fabric routes x86 RMP only based on DRAM routing rules. Nevertheless, we cannot exploit this behavior because we do not control the return value; worse, `0xFF` is interpreted as prohibited access by the RMP.

Observation 4: The Data Fabric routes IOMMU memory requests first based on Data Fabric MMIO routing rules before defaulting to DRAM routing.

IOMMU connects to the Data Fabric through IOMS. Although we cannot use our observation to bypass IOMMU, we can perhaps use other units, such as the PSP, but they would have to be connected to IOMS. Unfortunately, AMD does not document the exact location of the PSP on the platform. However, through code comments and debug strings, we know that the PSP is behind an IOMMU and thus most likely connected to an IOMS component. To further pinpoint the exact IOMS block of the PSP we use the PSP BAR registers and the Data Fabric MMIO routing registers. The PSP exposes a set of MMIO registers to the hypervisor for communication. These MMIO registers are part of a BAR region. The Data Fabric must forward writes targeting the BAR region to the PSP. Therefore, the Data Fabric must forward the MMIO requests to the IOMS with the PSP attached. So, to pinpoint the IOMS gateway of the PSP into the Data Fabric, we look at all the Data Fabric MMIO configuration registers and search for the one mapping the x86-PSP MMIO communication range `0x0xd1200000-0xd12fffff` and `0x0xd1300000-0xd1301fff`. We find register pair `0xdc0-0xdcc` maps the range `0xd0000000-0xd7ff0000` and routes the traffic to IOMS block `0x22`. We conclude IOMS block `0x22` connects the PSP with the Data Fabric.

Next, we want to tamper with the PSP accesses that are routed through the Data Fabric (e.g., to the x86 DRAM). We revive our attempts of shadowing the DRAM with MMIO routing rules, as we did for the IOMMU above, but this time for PSP. Specifically, we want to check if we can target PSP read and writes for the x86 DRAM and misroute them. Again, we target the PSP accesses to the RMP, which is stored in the x86 DRAM. We know that when the x86 core requests to enable SEV-SNP, it performs an API call to the PSP, which in turn initializes the RMP-Table in a secure state. If our hypothesis holds, we could redirect the PSP writes to the RMP-Table during initialization and thus compromise the initial RMP configuration. We use the MMIO register quadruple 15 and

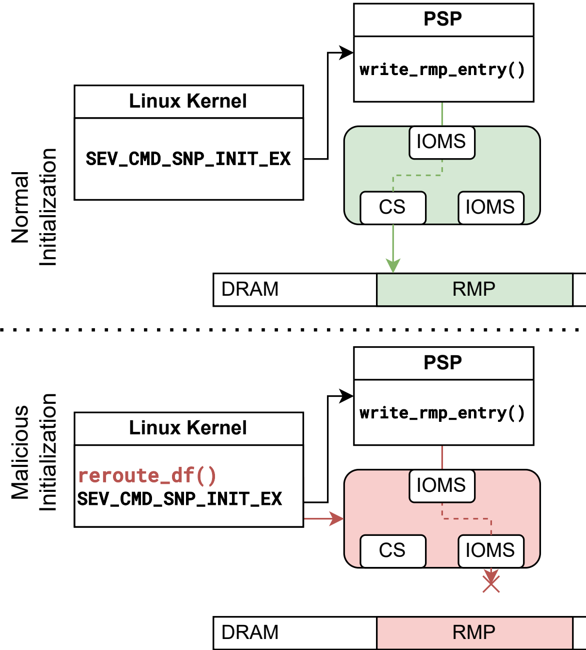

Figure 10: A malicious hypervisor reconfigures the Data Fabric at init-time and re-routes PSP memory requests.

shadow the RMP-Table range defined by two MSRs. Our experiments confirm that we can redirect PSP writes targeting the Data Fabric because the entire RMP stays zeroed (i.e., unchanged) despite PSP writes. In summary, observation 4 also applies to the PSP as it interacts with the Data Fabric through an IOMS unit.

Observation 5: The Data Fabric routes PSP memory requests first based on Data Fabric MMIO routing rules before defaulting to DRAM routing.

7 Case Studies & Implementation

This section presents how we use the previously introduced primitive to compromise SEV-SNP. We execute our experiments on a Zen 5 AMD EPYC 9135 16-core processor running SEV firmware version `1.55:44` with microcode revision `0x0b002116`. As a motherboard, we use an AsRock BERGAMOD8-2L2T with firmware version 10.01.00 and Linux 6.15.0-rc5 as the guest and host operating system. We present two case studies to undermine SEV-SNP.

Evaluation Beyond Zen 5. We evaluate our attack on the newest Zen 5 generation. We do not have access to a compatible motherboard that supports Zen 3 or Zen 4 to test our attack. Since the cloud providers restrict flashing the firmware on bare metal machines, and the potential of accidentally bricking the cloud machine with corrupted firmware, we refrain from renting a machine to test our exploits on cloud offerings.

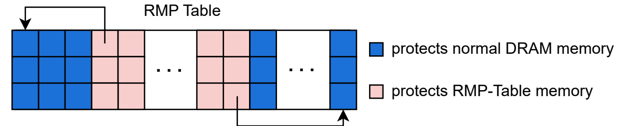

Figure 11: RMP-Table layout. FABRICKED enables an attacker to control the RMP entries protecting the RMP entry. This allows FABRICKED to make pages writable that are read-only for security reasons.

7.1 Compromising the RMP-Table

FABRICKED enables us to control the initial RMP-Table state after SEV-SNP initialization.

Skipping the Data Fabric Lock. We modify the UEFI of our motherboard to skip the PSP API calls `BootDone` and `LockDF`. Skipping the calls results in the Data Fabric being unlocked during runtime.

Shadowing the RMP Table. Before initialization, the RMP-Table is writable by the untrusted hypervisor. During RMP-Table initialization, we modify the Data Fabric configuration to redirect PSP writes. Figure 10 depicts the re-routing. Therefore, the writes to initialize the RMP-Table into a secure state never reach DRAM. This way, the SEV-SNP initialization succeeds while the hypervisor-supplied RMP state stays intact in the DRAM. The initial state of RMP is also critical, since the RMP-Table uses the RMP-Table for protection. Figure 11 depicts the RMP-Table and the entries in the RMP-Table that protect RMP-Table memory. Under benign execution, after SEV-SNP initialization, a hypervisor cannot directly write to memory hosting the RMP table, as x86 cores are subject to RMP checks, which prohibit direct writes. However, if we control the initial RMP configuration, we can set these self-protecting entries to allow writes to the RMP-Table memory. By gaining write access to the RMP table after the PSP initialized SEV-SNP, all SEV-SNP integrity protections are bypassed as the RMP ensures integrity by stopping certain memory writes.

Implementation. We modify the `AmdPspDxeV2Bhr` module by replacing the PSP API calls to `BootDone` and `LockDF` with `NOP`s. We further modify the Linux kernel with 223 LoC within the `sev-dev.c` file. The code changes the Data Fabric MMIO routing registers for IOMS unit `0x22` to shadow the RMP-Table. After initialization, we revert the Data Fabric changes and put the fabric into a stable state. Listing 4 shows the simplified attack code we run in the modified Linux kernel.

7.2 Arbitrary Memory Read and Write

We show how an attacker can transparently read and write arbitrary memory on a fully attested non-debug-enabled production CVM. This case study was originally proposed by a

```
df_indirect_write(/*[..]*/, rmp_base>>16);
df_indirect_write(/*[..]*/, rmp_limit>>16);
df_indirect_write(/*[..]*/, 0x22);
rc = __sev_do_cmd_locked(SEV_CMD_SNP_INIT_EX,
  /*[...]*/);
df_indirect_write(/*[..]*/, old_base);
df_indirect_write(/*[..]*/, old_limit);
ctrl.DstFabricID = old_dst;
df_indirect_write(/*[..]*/, old_dst);
```

Listing 2: Abbreviated modifications to `drivers/crypto/ccp/sev-dev.c`, misconfiguring the fabric.

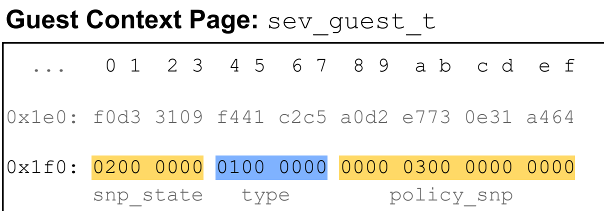

Figure 12: Approximate layout of the guest context, as per psp source code [6]. The `policy_snp` field needs to have its 19th bit set to high to enable debugging [12].

Google Project Zero SEV-SNP security review [30].

Debug Mode. If a CVM is in debug mode, a hypervisor can request the PSP to arbitrarily decrypt and encrypt CVM memory. While this feature is undesirable in production CVMs, AMD only enables the API if the hypervisor sets a specific debug-enable bit during guest creation. The debug-enable bit is part of the attestation report, and a guest thus detects if a hypervisor maliciously sets the bit. At runtime, the PSP accesses the debug-enable bit in the so-called guest context page. The guest context page is a 4KiB page that the PSP assigned to each CVM and holds security-critical information about a CVM (e.g., attestation hash, CVM ID). When the hypervisor wants to use the CVM memory decrypt/encrypt APIs of the PSP, it also supplies the physical address of the guest context page. The PSP checks the permission bits of the guest context page and either allows or disallows the API call. The RMP-Table prevents the hypervisor from writing to the guest context page at runtime. As an additional means of protection, the PSP inline encrypts the guest context page content. However, only the RMP-Table prevents the hypervisor from overwriting the encrypted guest context page, as there are no other integrity checks.

Malicious Debug Mode Enable. To maliciously enable the debug mode, we must flip the debug-enable bit after attestation. Since the guest only attests to the image in the beginning, any changes after attestation are impossible to detect. The only mechanism that hinders us from enabling the debug mode of a production CVM is the RMP-Table. The RMP-Table protects the guest context page from hypervisor overwrites. Since FABRICKED allows us to directly write to the RMP-Table, we can circumvent the write-protection by changing the RMP-Table entry of the guest context page. The guest context page is still encrypted, so we are unable to directly change the debug-enable-bit. However, we can change the ciphertext of the debug-enable-bit location, and have a 50% chance of the bit flipping. To find the offset within the guest context page where the debug-enable-bit is stored, we use static analysis of the open source PSP firmware. We compute the offset as the 16-byte encryption block starting from offset `0x1F0`. Figure 12 shows the position of the `policy_snp` enable struct field that embeds the debug-enable-bit. By enabling the debug mode after attestation, the hypervisor has the ability to read and write arbitrary CVM memory and thus compromises all SEV-SNP protection guarantees.

Implementation & Evaluation. We implement our attack in a kernel module in 226 LoC. Our code repeatedly changes the 16-byte encryption block at guest context page offset `0x1F0` and tries to execute the `SNP_DBG_DECRYPT` API call. If the call succeeds, we have successfully flipped the debug-enable bit; if not, we try a different ciphertext value and repeat until success. Since AES acts as a pseudorandom function, each iteration has a 50% chance of success. Our attack succeeds on average after 1.97 iterations for 1006 experiments, as we expect. The mean duration of the attack, all retries considered, is 1.59ms, and we record a maximum number of 21 attempts. Note that the victim CVM cannot detect the failed attempts. Thus, the attacker can keep trying till it succeeds, after which the VM is also unaware of the debug enablement.

7.3 Forging Attestation

We demonstrate how an attacker can supply the guest with arbitrary attestation values, similar to BadRAM [24].

Attestation. The guest requests attestation by making a hypercall to the hypervisor, which contains an encrypted payload with the attestation request. The hypervisor is required to forward the request to the PSP, who authenticates the guest request and serves it. The PSP returns an encrypted response to the hypervisor, who in turn forwards it to the guest for authentication. The guest observes the attestation value and determines the authenticity of the CVM image it is running.

Forging attestation. The PSP computes the attestation value once on CVM activation. Like the debug-enable bit, the attestation value is part of the guest context page. The attestation value is located at offset `0x460` and is 48 bytes long. The PSP encrypts the guest context page with a static key, freshly generated on every host boot. To successfully forge an attestation report, a malicious hypervisor must first create a benign CVM and snapshot the attestation value. In a second step, the hypervisor boots an arbitrary image and lets the guest connect to it. Shortly before the guest requests attestation, the malicious hypervisor uses the FABRICKED primitive to overwrite

the guest context page with the expected attestation value. The PSP returns the expected attestation value, despite the image being maliciously modified by the hypervisor. Subsequently, attestation succeeds, and the guest wrongly assumes it executes in the image supplied to the hypervisor.

Implementation & Evaluation. We boot the benign image and snapshot the attestation report using a kernel module. Subsequently, we boot a malicious image that generates a wrong attestation report when we query it from the guest. We request the attestation report. We observe the report differs, and attestation recognizes the changed image, as expected in a benign case. Next, we use the same kernel module again with different parameters and replace the `measurement` value in the guest context page. We request another attestation report, and the report matches the one of the benign image despite us changing the image. The attack works with 100% probability.

8 Discussion

We discuss the root cause, mitigations, and applicability to other architectures and motherboards.

8.1 Root Cause Analysis

Pinpointing the root cause of FABRICKED is not immediately obvious, as two issues combined enable our attack.

Root Cause #1. PSP forgot to verify that the UEFI locked the Data Fabric, which we classify as a firmware vulnerability. However, even with a locked Data Fabric, a malicious UEFI can create overlapping MMIO mappings for memory to influence the PSP writes to the RMP. We investigated the PSP source code and observed the following: To prevent the use of unprotected MMIO resources, the PSP operating system builds an internal system memory map that distinguishes between DRAM and MMIO [6, 10]. In fact, it even checks that all security-relevant data structures, such as the RMP-Table, are indeed residing in DRAM. Further, to avoid overlaps between the Data Fabric DRAM and x86 core MMIO regions, the PSP ensures that the RMP and other critical data structures are never mapped in any Data Fabric or x86 MMIO regions. None of these are effective since we directly change the MMIO routing. Thus, a defense would need to lock both the MMIO Data Fabric configuration and the existing mapping checks. Importantly, the PSP must recompute the internal memory map after locking the Data Fabric.

Root Cause #2. PSP writes targeting x86 DRAM are routed through MMIO rules first. We investigated the PSP source code and report the following observations. According to publicly available source code for an older SEV firmware version `1.55:25`, the PSP maps RMP-Table memory with the flag `AxUSER_DRAM_BYPASS_IOMMU`. As the SEV firmware code does not include the header file defining the flag, we search in all publicly available AMD repositories and find it. The header file states that this flag requests `General DRAM` and has the `ReqIO` bit not set [10]. The experiments in Section 6 show that when the x86 cores request DRAM, the Data Fabric routes it based on the DRAM routing rules, and the MMIO routing configuration is ignored, even if there is an overlap with the MMIO registers. We would expect the same behavior from the PSP, such that all memory requests with the target DRAM are routed based on the DRAM routing configuration. The experiments in Section 6, show that this is not the case, and PSP memory accesses targeting DRAM are first routed based on the MMIO Data Fabric registers, and if there is no match according to the DRAM Data Fabric registers. This leaves us with three possible conclusions. Either it is a hardware bug, and the Data Fabric should route it properly based on the DRAM registers, or the behavior is intended, in which case the source code comments are misleading, or the DRAM bit is not properly propagated when the PSP memory requests get forwarded to the IOMS unit. As this involves proprietary hardware knowledge, we are unable to narrow it down further.

8.2 Mitigation

We propose two mitigations to address each of the root causes.

Writable Data Fabric Registers. The PSP must check on SEV-SNP init that the Data Fabric register lock is active. Alternatively, the PSP can set the lock itself before SNP init. After locking the Data Fabric, the PSP must rebuild its internal system memory map to reflect possible changes. With a locked Data Fabric and a correct system memory map within the PSP, FABRICKED becomes infeasible as an attacker cannot re-route memory requests anymore. The PSP already builds a memory map and excludes critical data structures from overlapping with MMIO [6, 10].

MMIO Routing. The PSP requests writes to DRAM, but the Data Fabric routes the request based on MMIO routing rules first. We can only hypothesize if some component simply does not propagate a bit or if the PSP is architecturally incapable of requesting DRAM from the Data Fabric. However, the mitigation must ensure that the hardware routes the PSP memory access based on DRAM rules, like it already does for x86 memory accesses. Whether implementing this mitigation requires a new CPU generation or is as simple as correctly propagating the request can only be answered with HDL/firmware-level access.

8.3 Applicability to Other Architectures

In Intel TDX threat model, the UEFI is also untrusted. However, instead of relying on a dedicated security co-processor, Intel relies on Authenticated Core Modules (ACMs) and microcode like execution (MCHECK) on the x86 cores to verify that the UEFI has configured the platform in a correct state [39]. Due to the closed-source nature of Intel systems and the unavailability of any information regarding memory

routing, we were unable to conduct any experiments. A security report by Google Project Zero looked into the threat of a malicious UEFI for TDX and did not identify any weaknesses [31]. The report highlights the lack of specification and source code as a problem when evaluating TDX security.

Arm CCA uses the EL3 firmware to bootstrap its platform. As there is no hardware available, we are unable to conclude on the viability of FABRICKED on Arm CCA. However, public documents indicate that the entire bootchain will be signed such that no malicious code can be run during early boot [14].

8.4 UEFI Modification

Motherboard vendors may prevent end-users from flashing custom UEFI firmware onto their motherboards. This is in accordance with the NIST SP 800-193 specification to make the firmware more resilient against multiple attack vectors [54]. The implementation of the NIST SP 800-193 specification often includes a trusted FPGA that ensures only motherboard vendor-signed firmware boots. Importantly, these protections are orthogonal to AMD confidential computing. Thus, SEV-SNP does not interact with the FPGA on the motherboard in any way. More importantly, motherboard vendors are outside of the SEV-SNP threat model and can also act maliciously. This includes cloud providers that often have their custom motherboards [29, 49]. By using Asrock's BERGAMOD8-2L2T motherboard, which allows us to flash custom UEFI firmware, researchers can operate in the SEV-SNP threat model to experiment with the capabilities that a cloud provider has, i.e., full control over the UEFI firmware.

9 Related Work

We discuss closely related work to FABRICKED.

SEV and SEV-ES Attacks. Attacks against the first version of SEV exploited architectural weaknesses. Hetzelt et al. exploit the weakness that SEV only encrypts the CVM memory and not the register state [34]. Morbitzer et al. use the hypervisor ability to remap pages to extract the full encrypted CVM memory [51]. Many other works also use the ability to remap pages to extract secrets [50–52, 72]. Multiple works use untrusted CVM interfaces to compromise SEV-ES security guarantees [35, 58], as well as using power interfaces to compromise SEV [19, 71]. Lastly, Li et al. introduces TLB poisoning as well as ASID confusion attacks against SEV and SEV-ES [44, 46]. Since FABRICKED redirects PSP writes, it may be possible to also apply it to SEV and SEV-ES variants.

SEV-SNP Attacks. SEV-SNP is the de facto secure standard, solving many architectural weaknesses of its predecessors. However, a long list of side-channels is still applicable to particular applications running in an SEV-SNP CVM [21, 28]. Cipherleaks exploits repeating ciphertext values in the VMSA block to leak application data [45]. Li et al. systematically study ciphertext side channels in Linux on SEV-SNP [43]. Similarly, multiple other works use ciphertext visibility to leak confidential data. Hypertheft and Ciphersteal apply it to neural networks to leak weights and user inputs [76, 77]. Relocate+Vote and Heracles use an SEV-SNP API to move pages in memory and circumvent AMD tweak value protection [66,75]. Cachewarp exploits the `invd` instruction to revert CVM state [78]. Schlüter et al. exploit the interrupt injection interface to compromise SEV-SNP [64,65]. BadRAM exploits the missing alias checking in SEV-SNP to create alias memory addresses. They use the aliases to circumvent the RMP protection and forge attestation [24]. SEV-Step is a single-stepping framework that enables multiple of the previously mentioned attacks [73]. RMPocalypse uses a catch-22 during SEV-SNP initialization to compromise its integrity [63].

Platform Reverse Engineering. SGX Explained discusses the startup sequence of Intel processors and reverses aspects thereof [22]. FABRICKED reverses how AMD initializes the platform and fabric in the context of SEV-SNP. Buhren et al. reverse engineered the AMD PSP boot flow and voltage supplies [19]. A blog post looks into the PSP firmware and memory layout [23]. Ritter et al. reverse engineer the performance characteristics of the port mapping algorithm on AMD Zen [59]. Entrysign and CustomProcessing unit reverse engineer the microcode functionality on AMD and Intel [18, 42]. Further, multiple works reverse engineer different components of the CPU with the goal of security analysis (e.g., TLB caches [26, 68], the CPU frontend [74], the Intel CPU interconnect [56]) and RMP caching behavior [16]. Several works examine the inner working of Nvidia GPUs [32, 33, 79].

UEFI Reverse Engineering & Open Source Implementations. UEFIs are traditionally closed source. The coreboot [1] and Libreboot [47] projects aim to provide open source and versatile alternatives. Coreboot in particular also contributes to the Ghidra project, making analyzing UEFI modules in Ghidra possible [4]. In a similar vein, efiXplorer is a plugin for the Ida Pro SRE toolchain, which can analyze entire UEFI volumes and infer the dependencies between them [17]. UEFITools is a program to analyze and potentially modify UEFI firmware volumes [61, 62]. The firmware blobs for AMD platform have some additional, non-standard structures and content. PSPTool can analyze, display, and extract these [57]. FABRICKED benefits from some of these tools.

10 Conclusion

We present FABRICKED, a novel attack that manipulates the interconnect routing to compromise AMD SEV-SNP security guarantees. We hope that our reverse engineering of the platform initialization and its interaction with AMD Infinity Fabric opens a new line of investigation. FABRICKED serves as a strong motivator to vendors such as AMD to continue their open-source efforts for improved security analysis.

Ethical Considerations

We informed AMD about the vulnerability on the 3rd of August 2025. AMD acknowledged the finding on the 14th of August 2025. We discussed the vulnerability with AMD and agreed to a responsible disclosure process, accompanied by an embargo.

Stakeholders

The key stakeholders involved in this matter are AMD, cloud service providers, motherboard vendors, customers of those providers, and end users.

Impacts

Impacts: We follow the following ethical principles: reduce harm, make sure that the entity that can issue the patch is informed as soon as possible, help them to mitigate the vulnerability if needed, and ensure that the patches are released and deployed before the vulnerability is made public. We further follow a coordinated disclosure process that embodies these principles. We aim to reduce the following harms: data exposure, service disruption, and research misuse.

Responsible disclosure within the confidential computing threat model remains challenging, as the cloud providers, who may be the attackers, must be informed before the clients, potential victims, to deploy patches.

Mitigations

As per our independent assessment, as the manufacturer of the affected CPUs, we deem that AMD is in the right position to coordinate communication and mitigation efforts with the remaining stakeholders. Typically, they take on the responsibility of implementing the mitigations (which are possible in the case of our paper, as they informed us) and coordinate the release of the mitigation such that the impacted machines can be patched before the embargo is lifted. Since in our case, AMD has agreed to take the responsibility, we decided not to engage with the remaining stakeholders to ensure clear communication through one point of contact. Accordingly, we have not disclosed the vulnerability to any additional parties, including cloud vendors or customers, beyond our formal communication with AMD. To mitigate potential harms caused by our research, we did an initial responsible disclosure with AMD as our primary point of contact. They have agreed to lead the coordinated disclosure, and we were operating under an embargo agreement.

Decision

We decide to continue our research, as our insights will help to secure other platforms and contribute to more secure systems in general.

Open Science

Our artifacts include the binary we flash onto our Asrock BERGAMOD8-2L2T motherboard. The creation steps for the binary are detailed in the paper (see Section 4). The binary bootstraps the EPYC system but does not lock the Data Fabric registers. We include the Linux kernel that performs the attack described in Section 7.1 to compromise the RMP during initialization. Lastly, we include the kernel modules to perform the debug enable and attestation forgery attack. The artifact is available at https://zenodo.org/records/17830021.

References

[1] coreboot: Fast, secure and flexible Open Source firmware. https://coreboot.org/. Accessed: 2025-08-11.

[2] Raj Kapoor. Empowering the Industry with Open System Firmware (OSF), 2023. Accessed: 2025-08-05.

[3] Advanced Micro Devices, Inc. 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, 2020.

[4] al3xtjames. [GSoC] Ghidra firmware utilities, wrap-up. https://blogs.coreboot.org/blog/category/uefi/.

[5] AMD. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 10h-1Fh Processors Rev 3.12 (ID 42300), 2015.

[6] AMD. AMD-ASPFW. https://github.com/amd/AMD-ASPFW, 2023.

[7] AMD. AMD64 Architecture Programmer's Manual Volumes 1–5, Rev. 4.08, 2024.

[8] AMD. Processor Programming Reference (PPR) for AMD Family 1Ah Model 02h, Revision C1 Processors (57238 Rev 0.24), 2024.

[9] AMD. Firmwares/turin/ablpostcode.h. https://github.com/openSIL/amd_firmwares/blob/619e402fdcac91da293de28386f4508da6140326/Firmwares/Turin/AblPostCode.h, 2025.

[10] AMD. Firmwares/turin/bl_syscall.h. https://github.com/openSIL/amd_firmwares/blob/619e402fdcac91da293de28386f4508da6140326/Firmwares/Turin/bl_syscall.h, 2025.

[11] AMD. openSIL: Open-Source Silicon Initialization Library. https://github.com/openSIL/openSIL, 2025. Accessed: 2025-08-27.

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

[13] Arm. Powering Microsoft's Azure Cobalt 200 with Arm Neoverse CSS V3: The next generation of Arm-based compute for the AI era. https://newsroom.arm.com/blog/microsoft-azure-cobalt-200-arm-neoverse-css-v3.

[14] Arm. Arm CCA Security Model 1.0 (DEN0096). https://developer.arm.com/documentation/DEN0096/latest/, 2021.

[15] ARM. Arm Confidential Compute Architecture (ARM-CCA). https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture, accessed 2025-8-2.

[16] Alexis Bagia, Vincent Quentin Ulitzsch, Daniël Trujillo, Mengyuan Li, Mengjia Yan, and Jean-Pierre Seifert. A Close Look at RMP Entry Caching and Its Security Implications in SEV-SNP. In *ACM HASP*, 2025.

[17] Binarly efiXplorer Team. efiXplorer: Hunting UEFI Firmware NVRAM Vulnerabilities. *binarly.io Blog*. https://www.binarly.io/blog/efixplorer-hunting-uefi-firmware-nvram-vulnerabilities. Accessed on 2025-08-11.

[18] Pietro Borrello, Catherine Easdon, Martin Schwarzl, Roland Czerny, and Michael Schwarz. CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode. In *IEEE SPW*, 2023.

[19] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *ACM CCS*, 2021.

[20] Thomas Burd, Noah Beck, Sean White, Milam Paraschou, Nathan Kalyanasundharam, Gregg Donley, Alan Smith, Larry Hewitt, and Samuel Naffziger. "Zeppelin": An SoC for Multichip Architectures. *IEEE Journal of Solid-State Circuits*, 2019.

[21] Li-Chung Chiang and Shih-Wei Li. Reload+Reload: Exploiting Cache and Memory Contention Side Channel on AMD SEV. In *ACM ASPLOS*, 2025.

[22] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[23] DayZeroSec. Reversing the AMD Secure Processor (PSP). Part 1: Design and Overview. https://dayzerosec.com/blog/2023/04/17/reversing-the-amd-secure-processor-psp.html, 2023.

[24] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In *IEEE S&P*, 2025.

[25] Electronic Design. Chiplets: The future of semiconductor design, 2023.

[26] Philipp Ertmer, Robert Dumitru, and Yuval Yarom. Reverse-Engineering the Address Translation Caches. In *DIMVA*, 2025.

[27] Yazen Ghannam. x86/CPU/AMD: Print the reason for the last reset. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ab8131028710d009ab93d6bffd2a2749ade909b0, 2025.

[28] Lukas Giner, Sudheendra Raghav Neela, and Daniel Gruss. Cohere+Reload: Re-enabling High-Resolution Cache Attacks on AMD SEV-SNP. In *DIMVA*, 2025.

[29] Google. Titan in depth: Security in plaintext, 2017.

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

[31] Google. Intel Trust Domain Extensions (TDX) Security Review. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf, 2023.

[32] Zhongshu Gu, Enriquillo Valdez, Salman Ahmed, Julian James Stephen, Michael Le, Hani Jamjoom, Shixuan Zhao, and Zhiqiang Lin. NVIDIA GPU Confidential Computing Demystified, 2025.

[33] Yanan Guo, Zhenkai Zhang, and Jun Yang. GPU Memory Exploitation for Fun and Profit. In *USENIX Security*, 2024.

[34] Felicitas Hetzelt and Robert Buhren. Security Analysis of Encrypted Virtual Machines. In *ACM SIGPLAN/SIGOPS*, 2017.

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

[36] IDTechEx. Chiplets: Revolutionizing Semiconductor Design and Manufacturing, 2023.

[37] Intel. XuCode: An Innovative Technology for Implementing Complex Instruction Flows – ID 758404. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/xucode-implementing-complex-instruction-flows.html, 2021.

[38] Intel. Skylake Mesh Architecture. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html#inpage-nav-1-2, 2022.

[39] Intel. Intel Platform Security (784473). https://cdrdv2-public.intel.com/784473/784473_Intel%20Platform%20Security.pdf, 2023.

[40] Intel. Architecting for Flexibility and Value with Next Gen Intel® Xeon® Processor. https://hc2023.hotchips.org/assets/program/conference/day1/Platforms/HC2023.Intel.Gianos.v7.pdf, 2025. Accessed: 2025-08-11.

[41] Intel. Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, accessed 2025-8-2.

[42] Josh Eads and Kristoffer Janke and Eduardo Vela Nava and Tavis Ormandy and Matteo Rizzo. EntrySign: Exploiting AMD Zen CPU Microcode Signature Verification. Blog post and technical disclosure, 2025.

[43] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *IEEE S&P*, 2022.

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

[45] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security*, 2021.

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

[47] Libreboot Conributors. Libreboot – Free and Open Source BIOS/UEFI boot firmware. https://libreboot.org/, 2025. Accessed: 2025-08-26.

[48] Microsoft. Announcing Cobalt 200: Azure's next cloud-native CPU. https://techcommunity.microsoft.com/blog/azureinfrastructureblog/announcing-cobalt-200-azure's-next-cloud-native-cpu/4469807.

[49] Microsoft. Protecting Azure Infrastructure from silicon to systems, 2025.

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

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

[52] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. SEVerity: Code Injection Attacks against Encrypted Virtual Machines. In *IEEE S&PW*, 2021.

[53] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families: Industrial Product. In *ACM/IEEE ISCA*, 2021.

[54] Nist. Platform Firmware Resiliency Guidelines. https://csrc.nist.gov/pubs/sp/800/193/final, 2025. Accessed: 2025-12-07.

[55] Nvidia. Inside the NVIDIA Rubin Platform: Six New Chips, One AI Supercomputer. https://developer.nvidia.com/blog/inside-the-nvidia-rubin-platform-six-new-chips-one-ai-supercomputer/, 2026.

[56] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *USENIX Security*, 2021.

[57] PSPReverse. PSPTool: Display, extract, and manipulate PSP firmware inside UEFI images. https://github.com/PSPReverse/PSPTool, 2025. Accessed: 2025-08-26.

[58] Martin Radev and Mathias Morbitzer. Exploiting Interfaces of Secure Encrypted Virtual Machines. ACM ROOTS, 2021.

[59] Fabian Ritter and Sebastian Hack. Explainable Port Mapping Inference with Sparse Performance Counters for AMD's Zen Architectures. In *ACM ASPLOS*, 2024.

[60] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. CoVE: Towards Confidential Computing on RISC-V Platforms. In *ACM CF*, 2023.

[61] Nikolaj Schlej. UEFITool 0.21.5: UEFI firmware image viewer and editor. https://github.com/LongSoft/UEFITool/releases/tag/0.21.5, 2015. Version capable of making edits.

[62] Nikolaj Schlej. UEFITool: UEFI firmware image viewer and editor. https://github.com/LongSoft/UEFITool, 2022. Latest release version A72, released 2025-06-16.

[63] Benedict Schlüter and Shweta Shinde. RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP. In *ACM CCS*, 2025.

[64] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *IEEE S&P*, 2024.

[65] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. Heckler: Breaking Confidential VMs with Malicious Interrupts. In *USENIX Security*, 2024.

[66] Benedict Schlüter, Christoph Wech, and Shweta Shinde. Heracles: Chosen Plaintext Attack on AMD SEV-SNP. In *ACM CCS*, 2025.

[67] Teja Singh, Alex Schaefer, Sundar Rangarajan, Deepesh John, Carson Henrion, Russell Schreiber, Miguel Rodriguez, Stephen Kosonocky, Samuel Naffziger, and Amy Novak. Zen: An Energy-Efficient High-Performance x86 Core. *IEEE Journal of Solid-State Circuits*, 2018.

[68] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In *USENIX Security*, 2022.

[69] UEFI Forum, Inc. UEFI Platform Initialization Specification. Release 1.9. Technical report, December 2024. Accessed: 2025-08-04.

[70] UEFI Forum, Inc. Unified Extensible Firmware Interface (UEFI) Specification. Release 2.11. https://uefi.org/specifications, November 2024. Accessed: 2025-08-04.

[71] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV. In *DIMVA*, 2023.

[72] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *IEEE S&P*, 2020.

[73] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step A Single-Stepping Framework for AMD-SEV. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023.

[74] Ke Xu, Ming Tang, Han Wang, and Sylvain Guilley. Reverse-Engineering and Exploiting the Frontend Bus of Intel Processor. *IEEE Transactions on Computers*, 2023.

[75] Yuqin Yan, Wei Huang, Ilya Grishchenko, Gururaj Saileshwar, Aastha Mehta, and David Lie. Relocate-Vote: Using Sparsity Information to Exploit Ciphertext Side-Channels. In *USENIX Security*, 2025.

[76] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. HyperTheft: Thieving Model Weights from TEE-Shielded Neural Networks via Ciphertext Side Channels. In *ACM CCS*, 2024.

[77] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. CipherSteal: Stealing Input Data from TEE-Shielded Neural Networks with Ciphertext Side Channels . In *IEEE SP*, 2025.

[78] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security*, 2024.

[79] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *ACM CCS*, 2023.

[80] Vincent Zimmer, Michael Rothman, and Suresh Marisetty. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface, Third Edition*. DeG Press, 3rd edition, 2017.

A Data Fabric Registers

We introduce the location of important Data Fabric registers.

Configuration Space Routing. Function `0x0` register `0xC20` contains the destination Data Fabric FabricID of the unit that receives PCIe configuration space writes, accessing Data Fabric registers. On our machine, this unit is `0x77`, which corresponds to the PIE unit. The PIE unit is not further documented, despite multiple mentions in the openSIL and PSP SEV firmware codebase [6, 11].

MMIO Register. Further, function `0x0` hosts the 16 MMIO routing register ranges of the Data Fabric. The MMIO base register at offset `0xD80` specifies the lower address bits for the MMIO region. The MMIO limit register at offset `0xD84` specifies the upper address bits for the MMIO region. The MMIO control register at offset `0xD88` controls read/write access and the destination FabricID for the memory region. Lastly, an MMIO extension register at offset `0xD8C` optionally specifies high-order bits for MMIO addresses that have bits higher than 48 set. The register quadruples are continuously replicated 16 times in the configuration space [11].

DRAM Register. The same register set is also present for DRAM routing. The DRAM base is located within function `0x7` offset `0xE00`, whereas the limit is at offset `0xE04`. An interleaving register at offset `0xE08` controls the interleaving setting of DRAM access and the destination CS unit. The last register at offset `0xE0C` specifies additional details for the interleaving control [11].

B UEFI Reverse Engineering

We statically analyze the UEFI binary of our motherboard to identify code that configures the Data Fabric. As the firmware package, we use firmware version `10.01.00` of an ASRock BERGAMOD8-2L2T.

B.1 Searching for Data Fabric Configuration

We resort to the openSIL code base to see how the UEFI may configure Data Fabric registers. openSIL defines 20 register quadruples for DRAM routing configuration and 16 registers for MMIO routing configuration. Further, openSIL accesses the registers using a function to perform sanity checks. Two functions within the openSIL codebase read and write Data Fabric routing registers, `DfXFabricRegisterAcc(Read | Write)`. Looking at the binary is challenging because the compiler may have inlined or optimized those functions.

```
1  uint32_t DfXFabricRegisterAccRead(uint32_t Socket,
2                                    uint32_t Function,
3                                    uint32_t Offset,
4                                    uint32_t Instance){
5    // [...]
6    assert(Socket < 2);
7    assert(Function < 8);
8    assert(Offset < 0x2000);
9    assert((Offset & 3) == 0);
10   assert(Instance <= FABRIC_REG_ACC_BC);
11   // [...]
12   PciAddr.Address.Device = Socket + 0x18;
13   if (Instance == FABRIC_REG_ACC_BC) {
14     // [...]
15     RegisterValue=xUSLPciRead32(PciAddr.AddressValue);
16   } else {
17     // [...]
18     xUSLPciWrite32(PciAddr.AddressValue,FICAA3.Value);
19     // [...]
20     RegisterValue=xUSLPciRead32(PciAddr.AddressValue);
21   }
22   return RegisterValue;
23 }
```

Listing 3: Minimized `DfXFabricRegisterAccRead`. Note the asserted preconditions, which are useful for matching this function. Adapted from `xUSL/DF/DfX/DfXFabricRegisterAcc.c` in openSIL [2]

Listing 3 shows a minimized version of the Data Fabric register read function in openSIL.

As the name implies, the UEFI may use these functions to read and write Data Fabric registers, as documented in openSIL [11]. We use UEFITool to unpack the different individual binaries from these phases and then analyze them with IdaPro. Each of these modules contains a part called the UI Section, which contains the module name. To start, we used this section of the individual PEI modules to identify

potentially interesting binaries. Since AMD supplies AGESA, the module names are likely prefixed with `Amd`.

```
1 int32_t sub_75cf0940(int32_t arg1 @ ecx,
2                     int32_t arg2,
3                     int32_t arg3,
4                     int32_t arg4) {
5   if (arg2 >= 8)
6       sub_75cf006b(0xdff50080);
7   if (arg3 >= 0x2000)
8       sub_75cf006b(0xdff50081);
9   if ((uint8_t)arg3 & 3)
10      sub_75cf006b(0xdff50082);
11  esi_3 = ((arg1 - 8) & 0x1f) << 0xf;
12  if (eax != 0xffffffff) {
13    if (eax > 0xff) {
14      sub_75cf006b(0xdff50091);
15    }
16    esi_6 = (esi_3 & 0xffffc08c) | 0x408c;
17    sub_75cf16a8(esi_6,
18      (eax << 5 | (arg2 & 7)) << 0xb
19        | ((arg3 & 0xffc) | 2) >> 1);
20    ecx = (esi_6 & 0xffffc0b8) | 0x40b8;
21  }
22  else
23      ecx = (arg2 & 7) << 0xc | (arg3 & 0xfff) | (
     esi_3 & 0xffff8000);
24  return sub_75cf1695(ecx);
25 }
```

Listing 4: Decompiler output of the same function found in `AmdFabricBhrPei`

Listing 4 shows a function that closely resembles the read and write function present in openSIL. We can, with some confidence, assume that these subroutines are *functionally* identical. One good indication of this is the similarity of assert/precondition verification. `sub_75cf006b` is likely to be a kind of `assert_fail` function, as it contains an infinite loop and some debugging output, and it is called when the complement of the asserted condition holds. The control flow of the function also closely resembles the reference, with the distinction of broadcast or individual accesses being made, and the accesses based on this. Finally, the bitwise operations also mirror the conceptual setting of values in the bit field. Compare, for example, the calculation of `PciAddr.Address.Device` defined in Listing 5 with Listing 4, line 13: The caluclation `-8 & 0x1f` is equivalent to `+ 0x18` for range $[0,7]$, which includes the valid range the `socket` argument. It is then shifted by 15 bytes which matches the position of the `Device` field in the bitwise representation of the `EXT_PCI_ADDR` from Listing 5.

```
1 typedef struct {
2   uint32_t Register:12;
3   uint32_t Function:3;
4   uint32_t Device:5;
5   uint32_t Bus:8;
6   uint32_t Segment:4;
7 } EXT_PCI_ADDR;
8 typedef union _PCI_ADDR {
9   uint32_t AddressValue;
10  EXT_PCI_ADDR Address;
11 } PCI_ADDR;
```

Listing 5: Bitfield definition for a PCI(e) address value. Adapted from `xUSL/Include/Pci.h` in openSIL [11]

These comparisons hold for all of the other bitwise operations, and similarly for the `-Write` function.

By matching the function callsites with logging strings present in the openSIL open source code, we are confident that we identified the Data Fabric register read and write functions within the binary. As the UEFI actively uses these functions to write to Data Fabric registers, it indicates that at least at this stage of the boot process, certain Data Fabric registers are writable from the x86 core.

B.2 Finding Data Fabric Operations

Another aspect of analyzing the platform initialization stage is determining where the data fabric registers are read and written. Unfortunately, differently from for example module specific registers, no special instruction is used to perform such actions. We therefore make use of *signature matching*, a feature common in many SRE toolchains. Signatures are representations of functions present in analyzed files, which can be used to identify the same function in other programs. These would typically be used for functions that are statically compiled into programs. In IDA specifically, this can be done by exporting named functions in a binary as a `sig` file and then matching against these in other files. We do so, first with the function identified in Section B.1, and later with another, similar function which is used by the DXE stage. This also highlights an issue with using this technique: Small differences in how the compiler chooses to emit the machine code for these functions may make it very difficult to make use of such maching techniques. In this particular instance the DXE phase only makes use of the broadcast write functionality, which seems to have prompted the compiler to remove the other codepath for optimization reasons. This highlights a limitation of relying exclusively on matching previously found functions. In this case, however, it succeeds in determining after which modules no other write occurs, at least to the registers are interested in.