BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments

Jesse De Meulemeester*
jesse.demeulemeester@esat.kuleuven.be
COSIC, KU Leuven
Leuven, Belgium

Luca Wilke*
l.wilke@uni-luebeck.de
University of Lübeck
Lübeck, Germany

David Oswald
d.f.oswald@bham.ac.uk
University of Birmingham
Birmingham, United Kingdom

Thomas Eisenbarth
thomas.eisenbarth@uni-luebeck.de
University of Lübeck
Lübeck, Germany

Ingrid Verbauwhede
ingrid.verbauwhede@esat.kuleuven.be
COSIC, KU Leuven
Leuven, Belgium

Jo Van Bulck
jo.vanbulck@cs.kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

***Abstract*—The growing adoption of cloud computing raises pressing concerns about trust and data privacy. Trusted Execution Environments (TEEs) have been proposed as promising solutions that implement strong access control and transparent memory encryption within the CPU. While initial TEEs, like Intel SGX, were constrained to small isolated memory regions, the trend is now to protect full virtual machines, e.g., with AMD SEV-SNP, Intel TDX, and Arm CCA. In this paper, we challenge the trust assumptions underlying scaled-up memory encryption and show that an attacker with brief physical access to the embedded SPD chip can cause aliasing in the physical address space, circumventing CPU access control mechanisms.**

We devise a practical, low-cost setup to create aliases in DDR4 and DDR5 memory modules, breaking the newly introduced integrity guarantees of AMD SEV-SNP. This includes the ability to manipulate memory mappings and corrupt or replay ciphertext, culminating in a devastating end-to-end attack that compromises SEV-SNP's attestation feature. Furthermore, we investigate the issue for other TEEs, demonstrating fine-grained, noiseless write-pattern leakage for classic Intel SGX, while finding that Scalable SGX and TDX employ dedicated alias detection, preventing our attacks at present. In conclusion, our findings dismantle security guarantees in the SEV-SNP ecosystem, necessitating AMD firmware patches, and nuance DRAM trust assumptions for scalable TEE designs.

1. Introduction

Cloud computing has become an important paradigm that takes advantage of the economy of scale by sharing platform resources among mutually distrusting tenants. Traditionally, a privileged hypervisor software layer orchestrates and isolates different guest Virtual Machines (VMs). However, in this paradigm, cloud users must assume the hypervisor to be free from exploitable vulnerabilities, as well as trust the cloud service provider's administrators, staff with physical access, and local law enforcement. In response to these concerns, Trusted Execution Environments (TEEs) have been developed, including AMD's Secure Encrypted Virtualization (SEV) [19], Intel's Software Guard Extensions (SGX) [31], [57] and Trusted Domain Extensions (TDX) [34], and Arm's Confidential Compute Architecture (CCA) [6]. TEEs aim to facilitate private computations even in the presence of an untrusted hypervisor, guarding against both privileged software-level and hardware-level attacks.

To this end, TEEs implement strong, hardware-enforced access control mechanisms to protect data in use within the trusted CPU package while transparently encrypting all data before writing it to untrusted off-chip DRAM. Therefore, TEEs safeguard data confidentiality against advanced physical adversaries employing cold boot attacks [84] or DRAM interposers [49]. Initial TEE designs, like Intel SGX, prioritized additional strong cryptographic integrity and freshness protection to thwart data modification and replay attacks. However, ensuring freshness requires secure on-chip storage for the root of the integrity tree, which does not scale effectively with larger memory sizes. Consequently, these initial designs are limited to a relatively small memory region (*i.e.*, 128 or 256 MB) [26]. A clear industry shift, exemplified by Scalable SGX, TDX, SEV, and CCA, has since extended protection to full VMs, thereby scaling memory encryption to encompass the entire DRAM. However, this expansion may come at the cost of theoretically reducing the strength of cryptographic integrity guarantees against physical adversaries [31].

While it has been established that data encryption cannot conceal address access patterns [18], [49], and scaling up memory encryption may introduce powerful ciphertext side channels [50], [53] in code that is otherwise constant time, no practical integrity breaches have been demonstrated to date. One reason for this is that the cost of such advanced physical attacks is considered to be exceedingly high. For instance, the sole previous hardware attack that managed to extract access patterns from SGX's memory encryption engine required a prohibitive investment of $170,000 for

*These authors contributed equally to this work.

a DDR4 DRAM interposer [49]. It is worth noting that this interposer requires continuous physical access and may not even be fast enough to manipulate or replay data, let alone target more recent DDR5 technologies. Thus, in this paper, we analyze the remaining DRAM trust assumptions, considering the following fundamental questions:

Can the memory subsystem, especially DRAM modules, be manipulated to break integrity protections in scalable, new-generation TEE designs? Are these attacks viable for low-cost adversaries with minimal or no physical access?

Exploring a new research direction, we focus our attention on the memory subsystem's initialization process, which is conducted at boot time by BIOS system software in conjunction with DRAM module configuration data, both of which are explicitly distrusted from the perspective of CPU-based TEEs. Specifically, we introduce a novel, platform-agnostic technique to double the apparent size of DDR4 and DRR5 memory modules by unlocking and manipulating the onboard Serial Presence Detect (SPD) chip, which provides a standardized method for reporting physical memory properties to the BIOS. We dub such manipulated memory modules *BadRAM*. Notably, our practical, low-cost SPD manipulation setup requires only brief, one-time physical access and can be built for approximately $10. Moreover, we find that certain DRAM vendors incidentally leave SPD unlocked, potentially enabling software-only attacks without any need for physical access.

In our attacks, we double the apparent size of the Dual Inline Memory Module (DIMM) installed in the system to trick the CPU's memory controller into using additional "ghost" addressing bits. These addressing bits will be unused within the virtually enlarged DIMM, creating an interesting *aliasing* effect where two different physical addresses now refer to the same DRAM location. We develop a practical reverse-engineering method to locate these aliases and show that they can be exploited to bypass access control restrictions, including those implemented by TEEs. Most impactful, in the case of AMD SEV-SNP, we show that BadRAM attackers can tamper with or replay ciphertexts and even manipulate the crucial reverse map table data structure, thereby re-introducing potent page-remapping attacks [60], [61], [62] that initially prompted the development of SEV-SNP. Building upon these primitives, we construct a comprehensive, end-to-end attack that allows replaying the cryptographic launch digest used in SEV-SNP's attestation process. We experimentally demonstrate that this capability permits the launching of arbitrarily modified VM images without altering their attestation report, consequently undermining all trust in the SEV-SNP ecosystem.

Next, we analyze the effect of our BadRAM aliasing primitive on the security of other popular TEEs beyond AMD SEV-SNP. We find that "classic" Intel SGX incorporates suitable cryptographic integrity protections that effectively thwart ciphertext replay or corruption attacks but still allow BadRAM adversaries to discern precise, noiseless write access patterns at a fraction of the cost of prior work [49]. Conversely, Scalable SGX and TDX include a trusted code module that explicitly checks the physical memory space for aliases during boot time, preventing our attacks at present. Following our responsible disclosure, AMD plans to introduce a similar countermeasure through a firmware update for SEV-SNP.

Contributions. Our main contributions are as follows:

- We present a novel physical memory aliasing primitive based on malicious SPD data that bypasses TEE-imposed access-control restrictions at low cost and with one-time physical access.
- We show how malicious SPD configurations break AMD SEV-SNP's memory integrity feature, allowing to remap pages and corrupt or replay ciphertexts.
- We present an end-to-end attack on SEV-SNP's attestation, allowing arbitrary changes to the VM.
- We demonstrate low-cost, fine-grained, noiseless write-pattern leakage for classic Intel SGX.
- We discuss existing countermeasures as well as improvements to harden TEEs against BadRAM.

Responsible Disclosure. We disclosed the SPD aliasing attacks with proof-of-concepts to break SEV-SNP to AMD on February 26, 2024. AMD acknowledged our findings, which they are tracking under CVE-2024-21944 and AMD-SB-3015, and requested an embargo until December 10, 2024. Notably, AMD's official CVSS assessment (`AV:L/AC:H/PR:H/UI:N/S:C/C:N/I:H/A:N`) acknowledges that BadRAM attacks can be mounted by local, software-only attackers without physical access (e.g., via SSH). For the issue in classic Intel SGX, we did not deem disclosure necessary at this point, as the underlying problem of write-pattern leakage has been demonstrated before. To mitigate our findings, AMD will interactively check the DRAM configuration using "AMD Secure Boot loader firmware". Appendix A contains AMD's verbatim response.

Open Science. To ensure the reproducibility of our results, and to enable future science on memory-aliasing attacks and defenses, we open-source our practical SPD tools and evaluation scenarios at https://github.com/badramattack/badram.

2. Background

2.1. AMD Secure Encrypted Virtualization

AMD's Secure Encrypted Virtualization (SEV) [19] is a TEE that protects virtual machines against privileged software attackers, such as a malicious hypervisor. The intended use case is to run VMs in the cloud without needing to trust the cloud service provider. SEV uses a combination of access rights and memory encryption to protect the VM's data. Before writing data to DRAM, it is encrypted with AES XOR-Encrypt-XOR (AES-XEX) using a tweak value derived from the physical address [19], [66], [80]. The encryption keys are managed by the AMD Secure Processor (SP), which forms the hardware root of trust

for the system. The SP offers an API to the hypervisor to manage encrypted VMs. To protect the VM's plaintext data inside the system-on-chip, e.g., while it is in the cache, the data is tagged with a VM-specific identifier to restrict access to the corresponding VM. The updated version SEV Encrypted State (SEV-ES) [47] added encryption to the previously unprotected VM register file. The latest version, SEV Secure Nested Paging (SEV-SNP) [3], [4], mainly adds integrity protection to both the VM's memory content and its memory layout, preventing an attacker from writing to an encrypted VM's memory from software and restricting their ability to remap the VM's secure memory pages. Both mechanisms are implemented via an additional, hardware-managed data structure called the Reverse Map Table.

2.2. DRAM Organization

A DIMM consists of a number of SDRAM chips that are grouped together into ranks. Each of these dies contains grids of DRAM cells, consisting of a number of rows and columns, which are grouped together into banks. Each memory location within the DIMM is uniquely defined by its rank, bank group, bank, column, and row. Multiple DIMMs can be present in the system, organized into channels. Each channel can be accessed in parallel.

Accessing a certain memory location first requires activating the corresponding row. As only one row can be open at a time within a bank, switching between rows incurs a performance penalty. The translation of physical address to DRAM address bits, performed by the memory controller, is, therefore, not a one-to-one mapping. For instance, the channel, rank, and bank bits are typically obtained by XORing different physical address bits together [64], [76]. This spreads consecutive addresses over different channels, ranks, and banks, reducing the performance overhead.

2.3. Serial Presence Detect

The Serial Presence Detect (SPD) chip is a serial Electrically Erasable Programmable Read-Only Memory (EEPROM) part of a DIMM that contains the module's configuration data. This data includes, for instance, the physical properties of the DIMM (e.g., size, speed), as well as its metadata (e.g., serial number, manufacturing date). The SPD data encoding is standardized by JEDEC. On a high level, the EEPROM is divided into four blocks for DDR4 and 16 blocks for DDR5. In DDR4, the base configuration and end-user-programmable sections are stored in blocks 0 and 3, respectively, while in DDR5, they are stored in blocks 0–1 and 10–15.

Communication with the SPD chip takes place over the System Management Bus (SMBus) for DDR4 and the JEDEC Module Sideband Bus (SidebandBus) for DDR5. These are two-wire interfaces based on I^2C and I3C, respectively. Upon boot, the BIOS reads out the EEPROM of every connected DIMM to configure the system based on the reported parameters. This interface can also be exposed

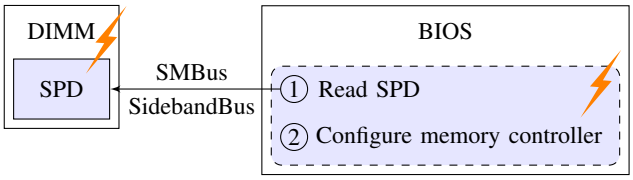

Figure 1. High level overview of the memory configuration steps performed by the BIOS. An incorrect DIMM topology can be either the result of a malicious SPD, or a malicious BIOS.

to software, allowing software to query the parameters of the connected DRAM modules.

Write Protection. To protect against accidental overwrites, each block can be optionally write protected. As per JEDEC specification, this write protection must be reversible, though doing so requires physical access to the DIMM. For DDR4, reverting the write protection requires connecting I^2C addressing pin SA0 to V_{HV} (7–10 V) and issuing the Clear all Write Protection (CWP) command [44]. For DDR5, the protection status is stored in registers MR12 and MR13, which can be modified by tying the HSA pin to ground [46].

JEDEC-compliant modules are required to protect blocks 0–1 for DDR4 and 0–7 for DDR5. Additionally, they must not set protection for the end-user-programmable blocks, which are, for instance, used to specify user-defined overclocking profiles through Intel XMP or AMD EXPO.

3. BadRAM Memory Aliasing Primitive

An adversary able to change the values of the SPD can trick the system into assuming different DIMM properties than those that are physically present. In this section, we use this idea to build up a primitive that creates physical memory aliases by making the DRAM appear larger than it actually is. In Sections 4 and 5, we explore the implications of this aliasing effect on TEE security.

To manipulate the reported DRAM size, the attacker needs to interfere with the memory initialization. This initialization is performed by the BIOS, which configures the memory controller based on the data reported by the SPD chip, as shown in Figure 1. As BIOSes are proprietary, we instead consider modifying parts of the SPD information to perform a data-driven attack against a benign BIOS.

3.1. Attacker Model

For our attacks, we assume an attacker with *(i)* root privileges on the target system and *(ii)* one-time physical access to a DIMM module installed in the system. Assumption *(i)*, *i.e.*, software root access, is the standard adversary model in the TEE context and in principle also includes arbitrary code execution in the BIOS. However, as BIOSes are notoriously inaccessible to end users, we introduce assumption *(ii)*, recognizing that TEEs generally assert a degree of protection against physical DRAM attacks. Intel SGX and TDX explicitly consider the DRAM subsystem

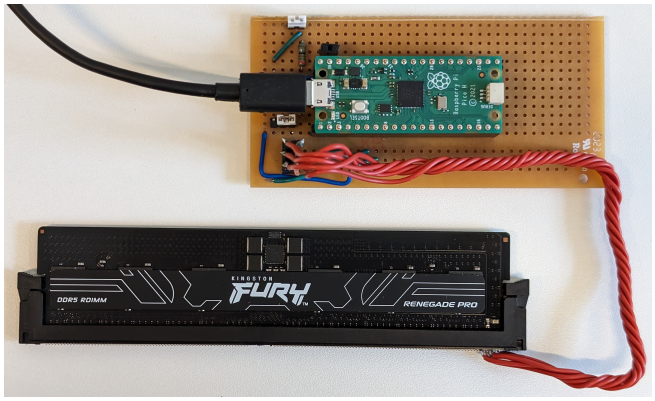

Figure 2. Raspberry Pi Pico setup to unlock and modify DDR4 and DDR5 SPDs.

untrusted [26], [36]. Similarly, AMD SEV-SNP considers certain DRAM attacks, such as cold boot attacks, as part of their threat model, while "on-line DRAM integrity attacks, such as attacking the DDR bus while the VM is actively running" are considered out of scope as they are deemed "very complex and require a significant level of local access and resources to perform" [3].

For our attacker model, we only require one-time physical access to manipulate the data on the DIMM's SPD chip. This could, for example, be performed by a malicious employee at a cloud service provider or through a supply-chain attack, without adding extra hardware to the system or leaving physical traces. We also note that we encountered off-the-shelf DRAM modules with disabled write protection, cf. Table 1, where the SPD could be potentially overwritten purely from software. Once the manipulated DIMM is installed, the attacker is no longer required to have physical access to the targeted system and can carry out the subsequent steps remotely from software.

3.2. Modifying SPD Contents

Experimental Setup. To physically interface with the SPD chip, we connected a standard Raspberry Pi Pico to the I^2C interface exposed on the DIMM, as shown in Figure 2. This offers a direct connection to the EEPROM, enabling read and write access, and allows to disable any potential write protection. As the SPD lacks authentication measures, its contents can be altered without detection by the system. This has been used before to adjust the DIMM's frequency [27], [48], or to create counterfeit memory modules [72]. Note that while DDR5 supports I3C, it remains backward compatible with I^2C and defaults to I^2C on power on [46]. The total setup cost to perform SPD modifications, which includes the Raspberry Pi Pico and DDR sockets, is approximately $10. Appendix B includes a full parts list.

DRAM Vendor Analysis. We analyzed several off-the-shelf DDR4 and DDR5 modules, including RDIMMs, UDIMMs, and SODIMMs, as summarized in Table 1. Performing a BadRAM attack requires modifications to block 0 for DDR4 and blocks 0 and 7 for DDR5 (as DDR5 stores the CRC in a separate block). The protection of these blocks is defined by WP0 for DDR4 and bits 0 and 7 of register MR12 for DDR5 (cf. Figure 3).

We found that most, though not all, memory modules lock the base configuration by default, as required by JEDEC, though they do not all set the remaining protection bits equally. We experimentally verified the ability to remove this protection with physical access as per the JEDEC specification. This operation can be performed entirely using the Raspberry Pi, with DDR4 only requiring an additional 7–10 V source, such as a boost converter or a 9 V battery. Notably, we found at least two off-the-shelf DDR4 DIMMs ($Corsair_1$ and $Corsair_2$) that leave the base configuration entirely unprotected, possibly exposing them to software-only BadRAM attacks. In specific cases, this may even lead to accidental corruption of the SPD, a known problem for some motherboards [55]. However, as these particular modules are UDIMMs, they are not compatible with our test server systems.

TABLE 1. WRITE PROTECTION AND ADDRESSING (HIGHLIGHTED) FOR VARIOUS DIMMS. FULL VERSION IN APPENDIX (TABLES 7 AND 8).

Manufacturer	Type	Write Protection				Addressing	
	DDR4	***WP0***	*WP1*	*WP2*	*WP3*	***Row***	*Col.*
$Corsair_1$	UDIMM	✗	✗	–	–	15	10
$Corsair_2$	UDIMM	✗	✗	✗	✗	16	10
$Crucial_1$	SODIMM	✓	✓	✓	✗	16	10
$Kingston_1$	RDIMM	✓	✓	✗	✗	16	10
$Kingston_2$	RDIMM	✓	✓	✗	✗	17	10
$Kingston_3$	RDIMM	✓	✓	✗	✗	17	10
$Kingston_4$	UDIMM	✓	✓	–	–	16	10
$Micron_1$	RDIMM	✓	✓	✓	✗	17	10
$Micron_2$	RDIMM	✓	✓	✓	✗	18	10
$Micron_3$	RDIMM	✓	✓	–	–	16	10
$Micron_4$	RDIMM	✓	✓	✓	✗	17	10
$Samsung_1$	UDIMM	✓	✓	✗	✗	16	10
$Samsung_2$	SODIMM	✓	✓	✗	✗	16	10
$SK\ hynix_1$	RDIMM	✓	✓	✗	✗	17	10
$SK\ hynix_2$	RDIMM	✓	✓	–	–	17	10
$SK\ hynix_3$	UDIMM	✓	✓	✗	✗	15	10
$SK\ hynix_4$	UDIMM	✓	✓	✗	✗	16	10
$SK\ hynix_5$	SODIMM	✓	✓	✗	✗	15	10
	DDR5	***MR12***	*MR13*			***Row***	*Col.*
$Kingston_5$	RDIMM	`0xff`	`0x3c`			16	10
$Kingston_6$	RDIMM	`0xff`	`0x00`			16	10
$Kingston_7$	RDIMM	`0xff`	`0x00`			16	10
$Samsung_3$	RDIMM	`0xff`	`0x01`			16	10
$SK\ hynix_6$	RDIMM	`0xff`	`0x01`			16	10
$SK\ hynix_7$	UDIMM	`0xff`	`0x01`			16	10

	7	6	5	4	3	2	1	0
MR12	WP7	WP6	WP5	WP4	WP3	WP2	WP1	WP0
MR13	WP15	WP14	WP13	WP12	WP11	WP10	WP9	WP8

Figure 3. Encoding of MR12 and MR13 for DDR5 [46]. BadRAM SPD modification attacks require bits 0 and 7 of MR12 to be clear.

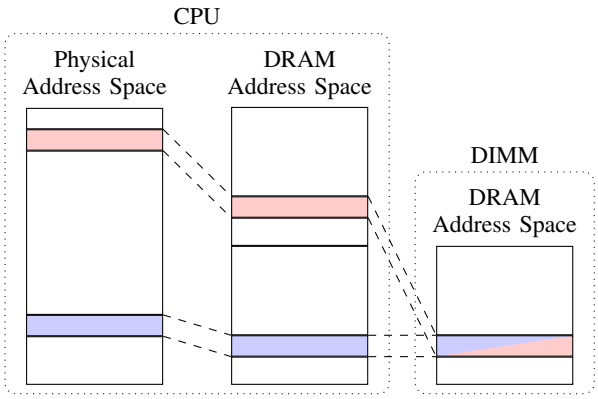

Figure 4. An incorrect configuration of the memory controller can result in unused address bits. Two addresses that only differ in the ghost bit alias to the same physical location inside the DRAM memory as this bit is ignored.

3.3. Creating Memory Aliases

With the ability to modify the SPD data, we can change the base configuration information of the DIMM. While not changing the underlying physical properties of the DIMM, this will change the properties as perceived by the host system. For instance, we can change the addressing that is used for the DIMM. The memory controller relies on this information to construct its physical-to-DRAM address mapping. This mapping depends, for instance, on the number of bits that are required to address each level of the DIMM's hierarchy [64], [76]. An incorrect configuration, as reported by the SPD, can create an inconsistent memory view between the CPU and DIMM. For instance, if the SPD reports more addressing bits than are actually used by the module, effectively increasing the size of the DIMM, the CPU will accordingly incorporate these bits into its mapping. However, since these "ghost" bits are not used by the DIMM, they are effectively ignored, creating aliases in the CPU's memory view.

Concretely, we consider modifying the SPD to report one additional address bit, not used by the DIMM. This effectively doubles the apparent size of the module. From the CPU's perspective, an additional address line will be driven, which is not connected to the DIMM and thus ignored by the addressing logic on the DIMM. This discrepancy in addressing between the CPU and DIMM results in two DRAM addresses, which only differ in the unused "ghost" bit, mapping to the same DRAM location, as shown in Figure 4. These aliases are invisible to the memory controller: from the CPU's perspective, these are two distinct addresses. As a result, they can bypass access control checks based on physical addresses, for instance, in the context of TEEs.

SPD Encoding. As mentioned before, the required information to correctly address the DIMM is stored in the SPD. For instance, byte 5 encodes the number of row and column bits in use by the DIMM, as shown in Figure 5. To introduce an additional DRAM address bit, we can modify

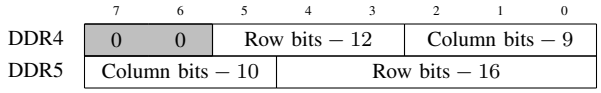

Figure 5. Encoding of byte 5 of the SPD's content, representing the SDRAM addressing [42], [45]. This byte is part of block 0 in the SPD.

TABLE 2. MAXIMAL DIMM CAPACITY SUSCEPTIBLE TO ROW-BASED BADRAM ATTACKS.

	Maximal DIMM Capacity	
Ranks	**DDR4**	**DDR5**
1	16 GB	32 GB
2	32 GB	64 GB
4	64 GB	128 GB
8	128 GB	256 GB

this byte to either increment the number of rows or columns. This necessarily also requires a modification to the DRAM density per die (byte 4, bits 0–3 for DDR4 and bits 0–4 for DDR5), which has to be updated to reflect the doubled capacity due to the additional addressing bit. Finally, the CRC bytes must be updated to match the modified content.

When booting a system containing a DIMM with a modified SPD, we found that some BIOSes may cache the SPD contents of the DIMM based on its serial number. This is, for instance, the case in coreboot, an open-source BIOS implementation [16, `src/include/spd_cache.h`]. The changes in the SPD may, therefore, not be applied if the module was already part of the system before. Modifying the serial number (bytes 325–328 for DDR4 or 517–520 for DDR5) simulates inserting a different module and thus forces the system to re-read the modified addressing information in the EEPROM. This behavior highlights that these memory mapping manipulations could also be performed by a malicious BIOS, which we discuss further in Section 6.1.

In our experiments, we opted to increment the number of row address bits. These bits are typically mapped to the highest physical address bits [64], [76], making unintended aliases, which impact system stability, less likely. Additionally, the column bits are typically combined in linear functions to determine the channel, rank, and bank, making the search for aliases more complicated. Assuming common DIMM properties (*i.e.*, a 64-bit interface, an 8 kB row size, and 16 (32) banks for DDR4 (DDR5)), all single-rank DIMMs with a capacity up to 16 GB for DDR4 and 32 GB for DDR5 will have at least one free row address bit. This capacity increases with additional ranks; the maximal capacity for DIMMs susceptible to row-based BadRAM attacks for common rank configurations is given in Table 2. In practice, however, this restriction does not significantly limit the attack surface as servers typically have many DIMM slots and only one modified DIMM is required for our attacks.

Finding Memory Aliases. In contrast to the simple example from the previous paragraph, physical address bits need not map one-to-one to DRAM address bits. The ghost bit, therefore, does not necessarily correspond to a single physical

Algorithm 1 Search alias for physical address $\mathcal{A}$.

```
1: for each page-aligned physical address B ≠ A do
2:     m1, m2 ← 64 random bytes;
3:     flush(B); write_mem(A, m1); flush(A)
4:     g1 ← read_mem(B); flush(B)
5:     write_mem(A, m2); flush(A)
6:     g2 ← read_mem(B)
7:     if m1 ⊕ m2 = g1 ⊕ g2 then
8:         return B
9:     end if
10: end for
```

address bit. Thus, finding two aliasing addresses may require scanning the entire physical memory space. However, this step needs to be done only once per memory configuration, as the mapping is deterministic. On a high level, we can search for the alias of address $\mathcal{A}$ by writing a marker value to it and scanning the remaining memory for another appearance of this marker. The process is slightly complicated by memory scrambling [84], where the memory controller XORs a randomized scramble pattern to the payload data before writing it to DRAM to even out the electrical load on the memory bus. As the scramble pattern is based on the physical address, a different pattern will be applied when reading from address $\mathcal{A}$ and its alias, hiding that they are, in fact, containing the same marker value.

To find the aliased address for $\mathcal{A}$, we use the approach shown in Algorithm 1. The flush operations are required because there is no cache coherency for aliased physical addresses. By comparing the XOR values on line 7, we ensure that the effect of memory scrambling cancels out. Note that for this one-time search, we temporarily disable memory encryption features like AMD SME [5, §7.10] or Intel TME-MK [33], as they encrypt the memory contents using AES with an address-based tweak. This is no longer a linear operation, and thus cannot be canceled out by an XOR-based comparison. For both SME and TME-MK, the encryption status can be configured with page granularity at runtime or for the whole system via BIOS settings. Some mainboards also allow memory scrambling to be disabled in the BIOS, simplifying the alias scanning to just looking for a second appearance of the marker value.

Evaluation. We evaluated our memory aliasing primitive on three different systems, which we refer to as AMD_1, Intel_1, and Intel_2 (cf. Table 3). On all systems, we modified the SPD to report one additional row and thus twice the actual memory size. To ensure stable system operation, we must prevent the kernel and all applications from using the introduced ghost memory regions to avoid accidental overwrites. We achieve this via the Linux kernel command line parameter `memmap=nn$ss`, which marks the memory region ss to $ss+nn$ as reserved, preventing the system from using it [54]. On all systems, booting with the upper half of the memory blocked by `memmap` resulted in a largely stable system. Our alias search implementation consists of a user space application implementing the core logic in Algorithm 1, assisted by a small Linux kernel module that enables direct access to arbitrary physical addresses.

TABLE 3. OVERVIEW OF EVALUATION SYSTEMS USED IN THIS PAPER.

System	TEE	Mainboard	CPU	DIMM(s)	DRAM
AMD_1	SEV-SNP	ASRock ROMED8-2T	EPYC 7313P	$1\times\text{Micron}_1$	DDR4
Intel_1	Classic SGX	Intel NUC7i3BNH	i3-7100U	$1\times\text{Crucial}_1$	DDR4
Intel_2	Scalable SGX	Supermicro X12DPi-NT6	Xeon 6330	$16\times\text{Micron}_3$	DDR4
Intel_3	TDX	ProLiant DL320 Gen11	Xeon 5515+	$8\times\text{Kingston}_7$	DDR5

The alias search revealed that on the two Intel systems, the ghost row address bit corresponded to a single physical address bit. On Intel_1, this bit corresponded to the most significant physical address bit, whereas for Intel_2—a dual-socket system—it corresponded to the second most significant bit, with the most significant one specifying the socket. On AMD_1, on the other hand, the memory was fractured into multiple chunks, with each chunk having a separate aliasing function. Furthermore, we observed that the exact layout was influenced by the memory regions blocked with `memmap`. Nonetheless, the system was stable enough to successfully carry out the attacks on SEV-SNP that are described in Section 4. We suspect that the address space fracturing could be related to "memory hoisting" [2], [41], which allows applying offset-based modifications to the way physical addresses map to DRAM.

4. Breaking AMD SEV-SNP

In this section, we show how the BadRAM primitive can be used to break SEV-SNP's newly introduced central memory integrity claim: "[...] if a VM is able to read a private (encrypted) page of memory, it must always read the value it last wrote" [3]. To this end, SEV-SNP imposes additional restrictions on the untrusted hypervisor to protect the integrity of the VM's memory layout and prohibit writing to its encrypted pages. In this section, we first explain how these features are implemented and then show how the protection can be broken using the BadRAM primitive. Finally, we demonstrate an end-to-end attack that breaks SEV-SNP's attestation, allowing an attacker to make arbitrary changes to a protected VM without changing its attestation report, breaking *all trust* in SEV-SNP.

With virtualization, there are two sets of page tables: the regular page tables used by the unenlightened OS inside the VM, and the Nested Page Tables (NPT) managed by the hypervisor. The addresses used by the VM are called Guest Virtual and Guest Physical Addresses (GPAs). The NPT is used to translate guest physical addresses to actual Host Physical Addresses (HPAs). With SEV, the hypervisor is in control of the NPT, allowing it to remap a GPA to a different HPA or to map two GPAs to the same HPA.

SEV-SNP's integrity features are implemented via the newly introduced Reverse Map Table (RMP). The RMP is a linear table that contains one entry for each HPA page that should be assignable to SEV-SNP VMs. Each RMP entry records various attributes. The most important ones are whether the page is used by an SEV-SNP VM

and, if so, the GPA at which the page is supposed to be mapped within the VM. Following AMD's nomenclature, we will call pages used by SEV-SNP VMs *guest-owned* and all other pages *hypervisor-owned*. The RMP has to be allocated before architecturally enabling SEV-SNP, by specifying its physically contiguous memory range via the `RMP_BASE` and `RMP_END` MSRs [5, §15.26.4]. Afterward, the hypervisor can no longer write to this memory region. Instead, it has to use a newly introduced set of instructions that grant it restricted access to the RMP. Thus, in contrast to the NPT, the information in the RMP is trustworthy. In the following, we show how we can use the BadRAM primitive to break both the memory layout integrity and the memory content integrity introduced by the RMP.

4.1. Breaking Memory Layout Integrity

Prior to SEV-SNP, there was no mechanism for a VM to detect changes in the GPA to HPA mapping performed by the hypervisor-controlled NPT. Morbitzer et al. exploited this for their SEVered attacks [60], [61], [62], which use the ability to swap the memory mapping of two pages in conjunction with a service running inside the VM, to decrypt arbitrary VM memory, inject hypervisor chosen plaintext and execute arbitrary code in the VM.

To prevent these kinds of attacks, SEV-SNP consults the RMP upon each page table walk caused by the VM to verify the integrity of the GPA to HPA mapping. The RMP assigns each guest-owned page a valid bit and an expected GPA. When the hypervisor first assigns a page to the SEV VM via the `rmpupdate` instruction or uses one of the other instructions to update the page's status, the valid bit is reset to 0. When a VM accesses a page with the valid bit set to 0, the VM is informed via a `#VC` exception, allowing it to validate the page if it deems the potential GPA change benign, e.g., the first time it accesses the page. For the validation, it needs to use the *pvalidate* instruction, which stores the current GPA of the page in the RMP and sets the valid bit to 1. When a VM accesses a page with the valid bit set to 1, where the expected GPA stored in the RMP does not match the GPA in the NPT, the hardware aborts the access and generates a nested page fault exception.

Using the BadRAM primitive, the hypervisor can circumvent the write protection of the RMP itself and, thus, make arbitrary changes to the stored GPA values without clearing the valid bit. Crucially, we find that the RMP's content is not encrypted. Thus, the hypervisor can directly write to the RMP and does not have to resort to replaying previously captured ciphertexts. As a result, the hypervisor can trivially swap the GPA-to-HPA mapping of two pages, by swapping both the GPA-to-HPA mapping in the NPT and the expected GPA in the corresponding RMP entry, as shown in Figure 6. This re-enables SEVered attacks [60], [61], [62], which, in conjunction with a service running in the VM, allow both decryption and encryption of SEV-SNP VM memory as well as code execution. Note that, as each GPA is inherently associated with exactly one RMP entry at a time, the hypervisor still cannot map two GPAs to the same

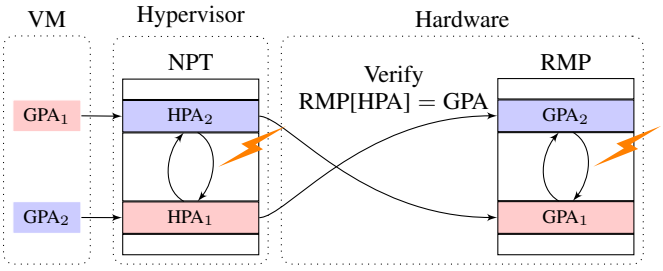

Figure 6. Mapping manipulation required for remapping attacks on SEV-SNP. The hardware checks that the HPA, as translated by the NPT, matches the expected GPA specified in the RMP. This enables the hardware to detect malicious mapping changes through the hypervisor controlled NPT. Thus, the attacker needs to change both mappings. The RMP manipulation requires our BadRAM primitive. The figure shows the entries *after* the swap has been performed.

HPA at the same time. However, such aliasing is not required for the cited attacks. If desired, adversaries can use single-stepping, e.g., SEV-Step [81], to precisely manipulate RMP entries at a maximal, instruction-level temporal resolution.

Proof-of-Concept. We implemented an elementary proof-of-concept on the AMD_1 system, using a single BadRAM memory module, showing that we can swap the mapping of two protected VM addresses. For the implementation, we modified the Linux kernel on the host system to provide an API to manipulate NPT entries. In addition, we use our previous kernel module for direct physical memory access to parse the RMP and modify it through a BadRAM alias.

4.2. Breaking Memory Content Integrity

Safeguarding the integrity of encrypted memory content in SEV-SNP is not enforced via cryptographic measures, but solely relies on the RMP. If SEV-SNP is enabled, the RMP is consulted for each memory write performed by the hypervisor [5, Table 15-39]. If the targeted page is guest-owned, the write attempt is blocked. Crucially, using the BadRAM primitive, we can circumvent this RMP protection by ensuring that the alias is hypervisor-owned. While the hypervisor can now write to guest-owned pages, their content is still encrypted with AES-XEX. As AES-XEX does not offer integrity, the guest cannot detect if a ciphertext has been manipulated. Instead, a modified ciphertext simply decrypts to a randomized value, i.e., it is not possible to make controlled semantic changes to the plaintext. Nonetheless, manipulating the ciphertext can be used as a capable fault primitive, e.g., against cryptographic schemes [11].

Since AES-XEX does not offer freshness, the hypervisor can also use the BadRAM primitive to replay previously captured ciphertexts. The tweak value used for the XEX mode depends on the HPA and boot-time randomness. Thus, ciphertexts can only be replayed to the same physical memory address they were read from. Otherwise, the mismatching tweak values lead to a randomized, garbage plaintext, similar to corrupting the ciphertext. As each SEV VM uses a

different encryption key, ciphertexts can also not be replayed across different SEV VMs.

Proof-of-Concept. We implemented an elementary proof-of-concept on the AMD_1 system, showing that we can randomize the content of a memory buffer inside the VM by modifying its aliased ciphertext from the hypervisor. To this end, we modified the Linux kernel on the host system to provide a convenient API for GPA-to-HPA translations. In addition, we use our previous kernel module for direct physical memory access to perform the modification through the BadRAM alias of the targeted address.

4.3. End-to-End attack on SNP's Attestation

In this section, we transition from integrity to confidentiality by demonstrating how the replay attack primitive discussed in the previous section can be leveraged to compromise AMD's crucial attestation feature, thereby undermining all trust in the SEV-SNP ecosystem.

SEV-SNP Attestation. Following common TEE design patterns, an SEV VM's lifecycle is split into two major phases:

1) In `GSTATE_LAUNCH`, the hypervisor creates the VM and prepares its memory content by using the corresponding launch API functions of the Secure Processor (SP), SEV's hardware root-of-trust. First, the hypervisor donates memory for the guest context data structure to the SP. The hypervisor has to mark the donated memory page as a firmware page in the RMP, preventing future writes. Next, the SP encrypts the donated memory using its memory encryption key and initializes the guest context. The guest context is the central data structure describing the SEV VM. Among other information, the guest context stores the launch digest, a cryptographic hash representing both the initial memory content and the initial memory layout of the VM.
2) Next, the initial memory content of the SEV VM is loaded by the hypervisor via the `SNP_LAUNCH_UPDATE` command offered by the SP. This command encrypts the memory with the VM's memory encryption key, without revealing the key to the hypervisor. On each invocation, the SP updates the launch digest inside the guest context accordingly. Eventually, the hypervisor uses the `SNP_LAUNCH_FINISH` command to transition the VM into the `GSTATE_RUNNING` state, which disables the launch commands API. The `GSTATE_RUNNING` state marks the VM as "runnable", allowing the hypervisor to start the VM via the `VMRUN` instruction.

There are two attestation mechanisms that can be used to verify the computed launch digest. First, the guest owner can prepare a so-called identity block (IdBlock), which, among other information, contains the expected launch digest. The IdBlock is an optional parameter that can be passed to the `SNP_LAUNCH_FINISH` command to make the SP check the launch digest before marking the VM as runnable. The second mechanism is more dynamic and allows the software in the VM to request an attestation report from the SP at runtime. The attestation report is signed by the SP and contains all information that is relevant for the guest owner to remotely verify the security of their running SEV-SNP VM. The launch digest is at the core of the attestation report, as it proves to the guest owner that only the expected code and data have been loaded into the VM and that the memory layout is as expected.

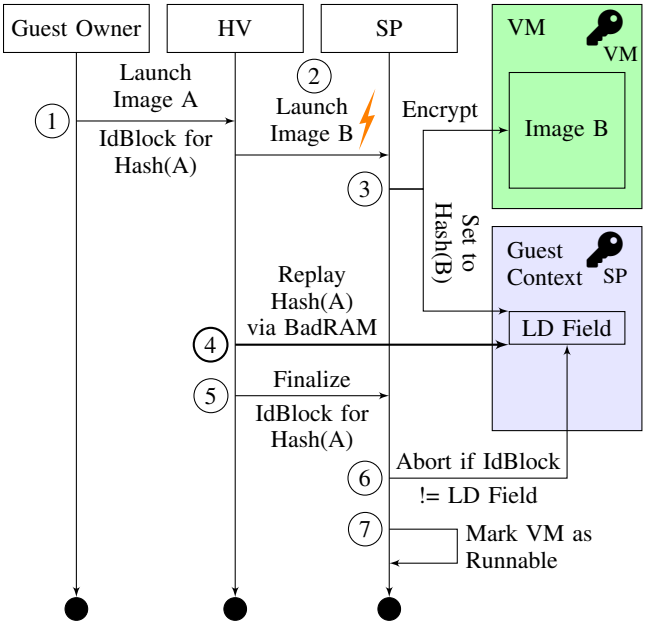

Figure 7. Online phase of the launch digest replay attack that breaks SEV-SNP's attestation. In step ① the correct image, together with a signed representation of the expected launch digest, is transferred to the hypervisor. In step ②, the hypervisor modifies the requested VM image to contain a backdoor. As a result, the launch digest after step ③ does not match the launch digest of the original image. However, in the offline phase (not depicted), the hypervisor captured the ciphertext of the correct launch digest using the BadRAM primitive, which it now replays in step ④. As a result, the launch digest again matches the expected value, passing the checks in steps ⑤ and ⑥.

In the following, we will show how the hypervisor can use the BadRAM primitive to manipulate the initial VM content, while still presenting the guest owner with the expected launch digest regardless of the used attestation mechanism.

Replaying the Launch Digest. Our attack comprises an offline and an online phase, exploiting that after receiving the VM image, the hypervisor can create an arbitrary amount of SEV-SNP instances for the image without having to interact with the guest owner. Figure 7 shows an overview of the online phase of the attack on IdBlock-based attestation.

In the offline phase, the hypervisor starts the VM without any modifications to the initial memory content and

captures the encrypted launch digest from the guest context page, before terminating the VM. As the hypervisor initially allocates the guest context page before donating it to the SP, it knows the physical address of the guest context page. In Figure 7, this would be equal to launching the requested image without modifications in step ②, capturing the ciphertext of the launch digest (LD) field in step ④ before terminating the launch process early.

In the online phase, the hypervisor donates the same physical memory page as in the offline phase to be used as the guest context page. Every time an SEV VM is created, the SP assigns it a fresh memory encryption key. Thus, the hypervisor cannot use any of the VM's ciphertext from the offline phase for replay attacks. However, the memory encryption key of the SP itself is only regenerated when the system reboots. Crucially, the guest context pages of all SEV VMs are encrypted with the SP's single memory encryption key. Since both the physical memory location of the guest context page and the memory encryption key are the same between the online and the offline phase, the hypervisor can replay the ciphertext of the benign launch digest from the offline phase in the online phase. Thus, after receiving the requested VM image in step ① the hypervisor can make arbitrary changes in step ②, as it can simply replay the previously captured benign launch digest in step ③ before finalizing the VM in step ④, which triggers a check of the launch digest in step ⑤. Due to the replay, the check succeeds, and the SP marks the VM as runnable in step ⑥. Note that the alternative, VM-triggered attestation procedure can only take place *after* step ⑥ and is, thus, also rendered useless by the launch digest replay.

To determine the exact offset of the 48-byte launch digest inside the encrypted guest context page, we use an empirical approach and dump the ciphertext of the guest context page between calls to `SNP_LAUNCH_UPDATE`. Next, we compute the difference between these dumps, revealing that only the 64 bytes from offset `0x460` to `0x4A0` change every time. Due to the 16-byte block size of SEV's memory encryption, the difference is only 16-byte granular. We correlate this information with the publicly available source code of the SP's firmware [1, `sev_rmp.h:226`], indicating that the array for the measurement is not 16-byte aligned, causing our replay to also overwrite the `IMIEn` field as well as parts of the 16-byte `GOSVW` field [4, Table 6]. However, both fields represent configuration options that cannot change between capturing and replaying, since we have to start the SEV-SNP VM with the same configuration options both times anyway.

End-to-End Attack. We analyze the use case where SEV-SNP is used together with a disk image, to bring up a fully-fledged VM, as described in [24], [56], [65], [79]. The disk image needs to use regular Linux full disk encryption to ensure the confidentiality and integrity of its content, as SEV-SNP does not offer any protection for virtual disks. The VM first boots into a minimal environment that runs a small server to provide the attestation report to the guest owner and subsequently establishes a secure communication channel.

TABLE 4. VULNERABILITY OF POPULAR TEES TO BADRAM ATTACKS.

TEE	Crypto	Ciphertext access			Mitigations
		Read	*Write*	*Replay*	
SEV-SNP (§4)	AES-XEX	✓	✓	✓	–
Classic SGX (§5.1)	AES-CTR	✓	✗	✗	Strong crypto
Scalable SGX (§5.2)	AES-XTS	✗	✗	✗	Alias check
TDX (§5.3)	AES-XTS	✗	✗	✗	Alias check
Arm CCA† (§5.4)	AES-XEX/ QARMA	Design suggests need for alias check			

†Only based on design documents as hardware is not yet available.

This minimal environment corresponds to the image that gets loaded in step ① in Figure 7 and thus is protected by the SEV-SNP. After verifying the report, the guest owner uses the secure channel to send the disk decryption key to the VM, which subsequently unlocks the disk and boots into the rich Linux environment contained inside the now-unlocked disk.

In our attack, we insert a backdoor in the minimal VM boot environment to leak the secret disk encryption key to the hypervisor. After a successful launch digest replay with BadRAM, the attestation report does *not* reveal this malicious modification, and the guest owner sends the disk encryption key as usual, thereby leaking it to the hypervisor. Using the key, the hypervisor has full read and write access to the encrypted disk image, allowing for arbitrary modifications or data leakage. After bootstrapping into the (now compromised) rich environment, the minimal boot environment is no longer accessible to the guest owner, making the attack undetectable. Alternatively, the malicious code could also be enhanced to delete itself before allowing the guest owner to log into the VM.

We fully implemented and ran the end-to-end attack on the AMD_1 system, using a single BadRAM memory module (cf. Table 3). To facilitate the replay of the launch digest, we modified the Linux kernel on the host system to ensure that the offline phase and the online phase use the same physical memory address for the guest context page. Furthermore, we modified the kernel code that calls the `SNP_LAUNCH_FINISH` command of the SP to capture the launch digest in the offline phase and to replay it in the online phase.

5. Analyzing DRAM Trust in Popular TEEs

In this section, we extend our analysis to the security assumptions that Intel's and Arm's TEE designs place on the memory subsystem and discuss whether their assumptions can be undermined by our BadRAM primitive. Our findings are summarized in Table 4.

5.1. Classic Intel SGX

Memory Encryption. The original Intel SGX architecture considered the DRAM as entirely untrusted storage, featuring a dedicated Memory Encryption Engine (MEE)

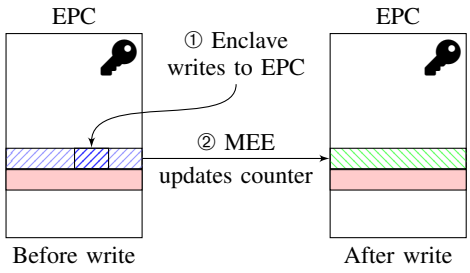

Figure 8. Using the BadRAM aliases, an adversary can monitor the EPC of classic SGX for changes in ciphertexts, revealing the write pattern. As SGX uses a fresh counter on every write, the ciphertext changes on every write, independent of the plaintext.

to encrypt and integrity-protect all enclave data stored in memory [26]. The MEE uses an AES-CTR-based cryptographic scheme that ensures the confidentiality, integrity, and freshness of the ciphertexts. It features a Merkle tree that guarantees the freshness of the counter values by storing the root of the tree in on-chip memory, inaccessible to malicious DRAM, with a fresh counter value generated on every memory write. Additionally, a 56-bit Message Authentication Code (MAC) ensures the integrity of the ciphertext. Therefore, both replayed and manipulated ciphertexts can be detected, locking the processor upon such violations.

In addition to strong cryptographic memory protection, SGX also prevents any read and write attempts by privileged software to enclave memory. This is implemented by reserving a physically contiguous memory range called the Enclave Page Cache (EPC), early during boot. Afterward, the hardware can prevent unauthorized accesses via a simple bounds check on the physical address.

Write Access Patterns. Crucially, BadRAM attackers can circumvent SGX's contiguous EPC check, enabling read and write access to enclave ciphertexts from software. While ciphertext modifications would be detected by the MEE, BadRAM attackers may still monitor the ciphertext for changes as a capable side-channel. Figure 8 shows that, when the enclave writes to its private memory, the corresponding ciphertext *always* changes, regardless of whether the underlying plaintext has changed. This is because the MEE is explicitly designed to assign a fresh counter on every write, protecting against the content-based ciphertext side-channels demonstrated on SEV platforms [50], [53].

Therefore, although the contents of memory accesses are effectively protected by the MEE, BadRAM adversaries who monitor relative changes in ciphertext over time can precisely deduce the location of write operations. As the CPU always writes back a whole cache line at a time, the obtained write address pattern has a spatial resolution of 64 bytes, which can be observed at a maximal, instruction-level temporal granularity using a single-stepping framework like SGX-Step [74]. This provides a more fine-grained leakage compared to an adversary that is only able to deterministically monitor page faults [83]. Additionally, in contrast to cache side channels [59], the obtained write pattern is deterministic and noiseless.

The sole previous study [49] that showcased DRAM address leakage on SGX necessitated continuous physical access and equipment costs reaching $170,000, making it unfeasible for most adversaries. Notably, however, such a full interposer-based setup can also reveal the read pattern, which is not possible with the BadRAM aliasing attack since reading does not alter the ciphertext.

Evaluation. We experimentally validated that classic SGX does not contain any mitigations against memory aliases by successfully booting the Intel_1 system with a single BadRAM memory module (cf. Table 3). We implemented an elementary enclave that writes to a random offset within a 4096-byte page-aligned buffer. A privileged software adversary monitoring page faults would be unable to distinguish two writes to different offsets within this buffer as they fall within the same page. With the BadRAM aliases, however, we were able to deterministically infer the offset at a 64-byte, cache line granularity by comparing the ciphertexts in the EPC before and after the victim wrote to it.

5.2. Scalable SGX

Memory Encryption. The requirement to maintain a dedicated Merkle tree in the classic SGX design does not scale well to large EPC sizes, prompting Intel to transition to a new, "Scalable" SGX design for Xeon server processors [31]. Scalable SGX abandoned the MEE and instead repurposes Intel Total Memory Encryption (TME).

TME uses AES-XTS to encrypt the entire physical memory range, but does not offer cryptographic integrity or replay protection. Thus, an adversary that can write to the EPC memory range, for instance, through an alias, would be able to corrupt or replay ciphertexts similar to our attacks on SEV-SNP. Indeed, similar to SEV, the tweak values used for AEX-XTS solely depend on the physical address and do not change during runtime. Thus, adversaries capable of reading encrypted EPC data would also be able to perform ciphertext side-channel attacks [50], [53].

Alias Checks. Notably, we found that Scalable SGX comes with dedicated architectural countermeasures against BadRAM aliasing attacks [31], [38]. Particularly, Intel differentiates between "outside-in" and "inside-in" aliasing.

First, outside-in aliasing refers to the situation where an EPC page has an alias that itself is not part of the EPC. To protect against these kinds of attacks, Scalable SGX repurposes one of the DRAM Error Correcting Code (ECC) metadata bits as an "ownership" bit to specify whether the cache line is part of the EPC [31]. If the CPU is not currently executing an SGX enclave, the memory controller returns a fixed pattern for read accesses to cache lines that have the ownership bit set. This mitigates ciphertext side-channel attacks from software. Writing to EPC memory while the CPU is not executing an SGX enclave clears the ownership

bit, which will be detected the next time an enclave tries to access the corrupted memory location.

Second, inside-in aliasing refers to the situation where an EPC page has an alias that itself is also part of the EPC. To protect against these, a trusted code module explicitly checks the physically contiguous EPC range for aliases before enabling SGX at boot time. On Xeon CPUs with Scalable SGX, the EPC can be as large as 512 GB, totaling 1 TB for a dual-socket system [39]. Initially, this check was part of SGX's `MCHECK` authenticated code module [32]. However, starting with 4th generation Xeon scalable processors, alias checking is now handled by a dedicated Alias Checking Trusted Module (ACTM) [38] that builds on Intel TXT [37] to run without trusting the BIOS.

Evaluation. We experimentally validated this behavior on Intel_2 (cf. Table 3), where we introduced BadRAM aliases by incrementing the number of row bits on all DDR4 DIMMs. On this system, the second most significant physical address bit maps to the unused row address bit and thus defines the aliases. When configuring a small EPC size such that there are no inside-in aliases, SGX enclaves could be instantiated. As specified, reading enclave memory from the aliases returned a fixed, all-zero value. When increasing the EPC size in order to create aliases within the EPC, the system did not boot into the operating system and reported a system initialization error (code 91). While the error code is unspecific, it suggests that the inside-in alias check detected an alias, as reverting to smaller EPC sizes cleared the error.

5.3. Intel TDX

Intel Trusted Domain Extensions (TDX) is Intel's latest TEE, moving away from the enclave-based paradigm to support confidential VMs, similar to AMD SEV, which are also referred to as Trusted Domains (TDs) [34]. Similar to SGX on Xeon scalable processors, TDX relies on Total Memory Encryption-Multi-Key (TME-MK) to provide memory confidentiality by encrypting its contents with AES-XTS. In contrast to SGX, however, TDX does not come with the concept of a fixed-size EPC, lifting any artificial limits on the total amount of memory used for TDs. Instead, TDX only uses the approach introduced with Scalable SGX to provide logical integrity protection and to prevent outside-in aliasing via a dedicated "TD-owner" ECC bit. Additionally, TDX introduces optional cryptographic integrity protection by storing a 28-bit MAC in the ECC bits for each cache line. This MAC is computed over the ciphertext, address-based tweak, TD-owner bit, and MAC key, and ensures the integrity of the cache line. If the integrity check fails, the memory location is marked as "poisoned" to prevent an attacker from brute-forcing the MAC [40, §16.2.1.1].

To prevent inside-in aliasing, TDX also relies on the ACTM to ensure that the BIOS configured the system correctly and that there are no aliases. However, to our understanding, the whole physical memory now has to be checked for aliases. As a result, it should not be possible to enable TDX in the presence of any memory alias, preventing all BadRAM attacks.

Evaluation. We experimentally verified the alias check behavior on Intel_3 (cf. Table 3). In contrast to the prior experiment on Intel_2 for Scalable SGX, the TDX-enabled system still booted, but we were unable to instantiate TDX or SGX when using either a single or multiple BadRAM modules. From this, we assume that the ACTM does indeed check the entire physical memory space for aliases. The different behavior is most likely a refinement in the alias check handling.

5.4. Arm CCA

Similar to SEV and TDX, Arm CCA is a VM-scoped TEE, though it is not yet commercially available. Like the aforementioned TEEs, CCA considers some attacks on DRAM to be in scope [6], [8] and features memory encryption as its primary defense. Additionally, the most critical data structures—belonging to the EL3 monitor that warrants CCA's security—receive extra protection [6]. They are stored either in on-chip memory, inaccessible to an attacker controlling external memory, or in external memory, but with additional integrity guarantees. The only exception is the Granule Protection Table (GPT) [7, §4], which is stored entirely in external memory and enforces memory isolation against software attackers, similar to SEV's RMP.

For memory encryption, CCA recommends AES-XEX or QARMA, which do not offer cryptographic integrity protection nor freshness. As a result, Arm CCA could be susceptible to BadRAM attacks, unless an alias check is performed similar to Intel's ACTM check for Scalable SGX and TDX. While the most critical data structures appear to be protected against these attacks through on-chip memory, the GPT and memory belonging to the realm management monitor and the realms themselves may still be vulnerable. Thus, as with SEV, both outside-in and inside-in aliasing may be possible, though we were unable to verify these claims due to the unavailability of CCA hardware.

6. Discussion and Mitigations

In this section, we discuss the feasibility of BadRAM attacks that are performed solely in software, without the need for one-time physical access. Next, we discuss mitigations and possibly more advanced DRAM attacks that may impact currently employed countermeasures.

6.1. Software-Only Adversaries

Up to this point, we have assumed an attacker with one-time physical access to the DIMMs, allowing the attacker to disable the module's write protection and overwrite the contents of the SPD. However, the SPD EEPROM may also be exposed over the SMBus or SidebandBus, allowing read and write access to the EEPROM by a privileged software-based adversary. Additionally, as the initialization of the

memory controller performed by the BIOS is based on the reported SPD values, a malicious BIOS may spoof these values for a similar effect.

SMBus & SidebandBus. The SPD chip is connected to the rest of the system via the SMBus or SidebandBus for DDR4 and DDR5, respectively. This interface may be exposed to software, like the `decode-dimms` utility from Linux's `i2c-tools` that provides comprehensive information on the connected memory. Performing BadRAM attacks by leveraging this interface requires the ability to write to the SPD base parameter section to change the addressing information. However, a DIMM may set its SPD write protection to disable writes to this section. Additionally, some memory controllers have protections in place that prevent writes to the SPD chip from software (e.g., through the SPD Write Disable (SPDWD) bit on the Intel PCH [35]), though some manufacturers allow this protection to be disabled in the BIOS, for instance to support on-DIMM RGB lighting [17].

BIOS. The BIOS reads out the SPD contents and configures the system based on the reported values, as shown in Figure 1. An adversary in control of the BIOS could change these values before configuring the system. This effectively enables BadRAM attacks without the requirement for physical access. However, as the BIOS is a complex, proprietary component (which might be cryptographically authenticated [20], [63]), this attack vector is significantly more complicated than modifying the SPD directly. While there are some efforts to create open-source firmware, such as coreboot, they currently do not support the newest TDX and SEV-SNP platforms.

Spoofing the SPD contents is done in practice, for instance, on devices with soldered memory, such as certain laptops and smartphones. As these devices do not have a physical SPD chip, their memory characteristics are stored in the BIOS image, allowing the BIOS to configure the system. In case of multiple memory configurations, a jumper selects the correct SPD contents. Furthermore, in Section 3, we observed on some of our evaluation platforms that the BIOS caches SPD data based on the DIMM's serial number, providing further evidence of the BIOS's ability to spoof SPD data.

6.2. Countermeasures

The key weakness exploited in our BadRAM attacks is the implicit trust placed on the BIOS to correctly configure the memory controller. The BIOS, in turn, trusts the information it reads from the DIMM's SPD chip.

Improving SPD Security. To increase the complexity of the attack, future DRAM generations could consider allowing permanent write protection on the base configuration blocks within the SPD. In fact, this was possible up to DDR3 [43], though not required. Removing the ability to modify the addressing parameters in the EEPROM requires the attacker to either physically replace the entire SPD chip, or modify the part of the BIOS that programs the memory controller, significantly increasing the complexity of the attack. This does not prevent attacks, though, as shown in Appendix C.

Validating Memory Layout. A more principled mitigation is to check the memory configuration during system boot to ensure there are no aliases. However, as such alias checks become part of the system's trusted computing base, the code must be protected from manipulations, e.g., by the BIOS, essentially requiring a low-level TEE.

A straightforward way to ensure that there are no aliases is to iteratively scan the entire DRAM memory space of each DIMM. However, this requires at least one read and one write operation to each address, making it impractical for systems with many or large DIMMs. Instead, each address bit can be verified separately. For each bit, we can consider two addresses that only differ in that specific bit. By writing a random value to the first address, we can check if this bit is used by reading the second address. If the read returned the same value we wrote to the first address, both addresses point to the same physical location, and the bit we considered is not used by the DIMM. The memory controller could even use this technique to discover the DIMM topology without relying on the BIOS or SPD to provide this information. If we ensure the integrity of the memory controller's firmware, this could be one solution to isolate the alias-checking code from the untrusted system. However, in the face of a hardware-level attacker, scanning for aliases is likely susceptible to time-of-check to time-of-use attacks, as discussed in Section 6.3.

As described in Sections 5.2 and 5.3, Intel has implemented an alias check for Scalable SGX and TDX using their TXT technology [37], [38]. We experimentally confirmed the presence of such alias checks, but did not find any documentation on the specific implementation and scope of the employed scanning algorithm.

Strong Cryptography. Using strong cryptographic primitives for memory encryption that provide memory integrity and freshness almost entirely mitigates the security risks introduced by memory aliasing. In addition, they can uphold their security guarantees against hardware attacks on external memory without the risk of time-of-check to time-of-use attacks. However, practical designs often face scalability limitations. While MACs can be used to ensure data integrity and integrity trees to provide freshness, both methods introduce memory overheads that scale linearly with the total amount of protected memory. Additionally, ensuring freshness requires storing the root of the integrity tree in on-chip SRAM, which is expensive. Furthermore, these primitives introduce performance overhead for every memory access: verifying the integrity of a read requires traversing the tree and computing the corresponding MACs, while writes similarly require updating these values. As a result, the depth of the tree must be limited in practice, constraining the amount of protected memory and thus making large memory sizes impractical. For instance, Classic Intel SGX—one of the few commercial TEEs providing cryp-

tographically secure memory integrity protection—supports only up to 128 MB or 256 MB of protected memory while incurring a performance overhead of up to 14 % [26].

Protecting large amounts of memory with both strong cryptography and acceptable overhead is a challenging research question. Recent academic works provide more scalable designs by employing skewed [70] or mountable Merkle trees [23], increasing the arity of the integrity tree by reducing the counter size [67], [71], dynamically adjusting the tree's height and arity [75], and changing the underlying cryptographic primitives [30]. While these approaches enable integrity protection for larger memory sizes, they have not been adopted by industry. Furthermore, even with strong integrity and freshness guarantees, attacks on the external memory may still enable high precision, sub-page access pattern leakage, as discussed in Section 5.1. Thus, software still needs to follow oblivious, constant-time programming paradigms to avoid such leakages.

6.3. Unverified Trust in DRAM Hardware

The analysis in this paper shows that most recent TEEs place some degree of trust in the memory system and DRAM without verifying it. Partial exceptions are Intel Scalable SGX and TDX, which check for memory aliasing at boot time, detecting permanent manipulations to the DRAM addressing. One example of this unverified trust is the way Intel, AMD, and Arm implement replay protection for their VM-based TEEs. Instead of using cryptographic freshness to prevent replay attacks, they use an access rights mechanism to prevent an attacker from writing to protected memory. This protection breaks if the attacker is able to modify the DRAM content via a channel that is not subject to the access control mechanisms.

The SPD manipulation from this paper essentially shows a data-driven attack against an otherwise benign BIOS. However, another attack angle would be a full DRAM interposer [49] or a DRAM module with manipulated hardware that, e.g., allows arbitrary read and write to the memory content via a second interface that is only available to the attacker. Such a hardware attacker could easily hide any manipulations from a boot time alias check, like the one performed by Intel. Hopkins et al. [29] discuss such modified memory modules for DDR3 and also implemented an FGPA-based prototype that attaches to the DRAM slot and acts as an interposer. It allows the attacker to redirect memory accesses to protected regions. We found one company that sells memory modules that come with data processing units, allowing to execute custom code directly on the DIMM [73]. However, their DIMM is currently restricted to a few mainboards, none of which support TDX.

7. Related Work

We start this section by reviewing existing work on memory aliasing attacks. Next, we discuss existing attacks on AMD SEV-SNP and how BadRAM re-enables some attacks previously mitigated by SEV-SNP. Finally, we survey hardware attacks on TEEs.

Memory Aliasing. Breuer et al. consider memory aliasing in a security context [10] and define two types: "software aliasing," where multiple logical addresses map to one physical address, and "hardware aliasing," where multiple physical addresses map to one logical address. They observe that the latter case can, e.g., arise when the number of address lines exceeds the bit width of the CPU arithmetic and develop methods to "certify" the safety of machine code in this scenario. More recently, Intel contributed a vulnerability class, "CWE-1257: Improper Access Control Applied to Mirrored or Aliased Memory Regions" [58], to MITRE's Common Weakness Enumeration list.

In research on virtualization security, Wojtczuk [82] speculated in 2016 that malicious memory aliasing may be induced by modifying the contents of a DIMM's SPD but did not further evaluate this attack surface. Furthermore, they only discuss software access to the SPD, which does not work if the SPD is locked. However, as shown in Table 7, most manufacturers seem to lock the SPD. In BadRAM, we show how to unlock the SPD chip with a low-cost setup and explore the resulting attacks in depth. For the opposite case of software aliasing, Guanciale et al. show that virtual aliases with different attributes can be used to construct cache-based side-channel attacks [25].

Software Attacks on AMD SEV. Initial attacks [28], [77] exploit that, prior to SEV-ES, the unencrypted Virtual Machine Control Block (VMCB) allowed read and write access to the VM's register file during context switches.

A long line of attacks [28], [60], [61], [62] exploits the hypervisor's control over nested page tables, breaking the integrity of the VM's memory layout. The SEVered attack [61] uses this attack primitive to trick services inside the VM to encrypt and decrypt arbitrary data. SEV-SNP was designed to mitigate this class of attacks by introducing the RMP that provides integrity to the VM's memory layout. With BadRAM, we break the RMP, re-enabling these attacks.

In [22], [80], the authors unveil the details of SEV's tweaked encryption mode, showing flaws that allow reverse engineering the tweak values. Exploiting the known tweaks, [52], [80] show that by adjusting for the tweak differences, moving ciphertexts in memory allows building mechanisms to encrypt and decrypt arbitrary data. All SEV-SNP-enabled CPUs use strong tweak values, mitigating these attacks. Buhren et al. [11] exploit the missing integrity protection to perform fault attacks by flipping ciphertext bits, which is mitigated with SEV-SNP by the RMP's write protection. Our BadRAM primitive re-enables this attack. Li et al. [50], [53] introduce ciphertext side-channel attacks, showing that the boot-time fixed tweak values used by SEV allow leaking access patterns of the executing code. This attack is not mitigated on SEV-SNP. Starting with revision 1.55 from September 2023, the SEV-SNP spec [4] mentions a "ciphertext hiding" feature but does not provide further details.

Schlüter et al. [68], [69] exploit the hypervisor's ability to inject unexpected interrupts. In combination with insufficient sanitization by the Linux kernel running in the VM, they are able to change the VM's register values, allowing them to eventually read/write memory and execute arbitrary code. While SEV-SNP does provide additional hardware features to mitigate these attacks, there is currently no software support. Wilke et al. [81] show that the external APIC timer interrupt can be used to single-step SEV-SNP VMs, enhancing the resolution of side-channel attacks. There is no mitigation for SEV-SNP. Single-stepping is also used by [69], [85]. Cachewarp [85] exploits a microcode bug to drop cache write-backs, which has been fixed by an update. CrossLine [51] exploits improper ASID checks prior to SNP.

In summary, SEV-SNP is currently vulnerable to the following software attacks: interrupt injection [68], [69], ciphertext side-channel [50], [53], and single-stepping [81]. Using the BadRAM primitive from this paper, we re-enable fault attacks [11] as well as SEVered [61], [62] attacks, essentially downgrading SEV-SNP back to SEV-ES. With the attack on the attestation presented in this paper, we break all trust in the SEV-SNP ecosystem.

Physical Attacks on DRAM and TEEs. An attacker with physical access to the CPU can, for example, manipulate the CPU voltage, which may introduce faults within the code running on the system [9]. The VoltPillager and PMFault attacks show how an attacker can inject faults into SGX enclaves by sending packets on the various voltage regulator interfaces used on modern Intel CPUs [14], [15]. Similarly, Buhren et al. glitch the AMD Secure Processor over the same interface to break the confidentiality and attestation features of SEV-SNP [12], [13].

For DRAM specifically, Hopkins et al. [29] present a DDR3 interposer that remaps attacker-controlled addresses to protected ones when inserted between a DIMM and the CPU. They discuss placing the interposer directly on the DIMM's IC but opt for an FPGA-based implementation for their prototype that only supports DDR3 up to 800 MHz. Similarly, Lee et al. use a commercial interposer, with a purchase price of $170,000, to capture the addresses on the DRAM bus [49]. While SGX encrypts the EPC contents, it does not protect the addresses. Their attack, dubbed Membuster, uses the captured access pattern as a side-channel to uncover secrets in non-constant-time code. On simpler, embedded systems, merely shorting or connecting an address line with tweezers or a sewing needle may, in certain cases, suffice to overcome security functionality, such as the memory protection in the Nintendo Wii [78] or the boot process of an embedded Linux system [21].

In contrast to BadRAM, the attacks from this section require attaching additional physical hardware to the system, limiting their applicability, e.g., in data centers with strict physical access checks and inspections.

8. Conclusion

In this paper, we presented a novel primitive that challenges the notion in modern TEEs that scalable memory encryption in combination with software-based access control suffices to provide integrity guarantees against untrusted DRAM. While commonly assumed to require expensive equipment and extensive physical modifications, we showed how integrity guarantees can be practically invalidated using off-the-shelf components for approximately $10 and one-time physical access to the DRAM module. To this end, we modified the DIMM's SPD data to create aliases in the physical address space that can effectively circumvent software-based access restrictions. Moreover, incorrectly configured DIMMs may even enable software-only attacks.

We demonstrated how the BadRAM primitive can be used to invalidate the newly introduced integrity guarantees provided by AMD SEV-SNP, breaking all trust by replaying critical attestation reports in an end-to-end attack. Additionally, we analyzed the boot time countermeasures baked into Intel's Scalable SGX and TDX to fend off aliasing attacks. Since our BadRAM primitive is generic, we argue that such countermeasures should be considered when designing a system against untrusted DRAM. While advanced hardware-level attacks could potentially circumvent the currently used countermeasures, further research is required to judge whether they can be carried out in an impactful attacker model.

Acknowledgements

This work was supported by the Research Fund KU Leuven, the Research Foundation – Flanders (FWO) via grant #1261222, and the Flemish Government (Cybersecurity Research Program) via grant VOEWICS02. In addition, this work is supported by the European Commission through Horizon 2020 (ERC #101020005 BELFORT), and Horizon Europe (#101070008 ORSHIN). This research was also partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/X03738X/1, EP/V000454/1, and EP/R012598/1. The results feed into DsbDtech. Additionally, this work was supported by the German BMBF project SASVI. Jesse De Meulemeester is funded by an FWO fellowship (11PFE24N).

References

[1] AMD, "AMD-ASPFW," https://github.com/amd/AMD-ASPFW, commit 3ca6650.

[2] ——, "BIOS and kernel developer's guide (BKDG) for AMD family 15h models 00h-0fh processors," AMD, Tech. Rep. 42301, Rev. 3.14, 2013.

[3] ——, "AMD SEV-SNP: Strengthening VM isolation with integrity protection and more," January 2020.

[4] ——, "SEV secure nested paging firmware ABI specification," AMD, Tech. Rep. 56860, Rev. 1.55, September 2023.

[5] ——, "AMD64 architecture programmer's manual volume 2: System programming," AMD, Manual 24593, Rev. 3.42, March 2024.

[6] Arm, "Arm CCA security model," Tech. Rep. Arm DEN 0096, Version 1.0, August 2021.

[7] ——, "Learn the architecture - introducing Arm confidential compute architecture," Tech. Rep. Arm DEN 0125, Version 3.0, June 2023.

[8] M. Bartock, M. Souppaya, R. Savino, T. Knoll, U. Shetty, M. Cherfaoui, R. Yeluri, A. Malhotra, and K. Scarfone, "Hardware-enabled security: Enabling a layered approach to platform security for cloud and edge computing use cases," National Institute of Standards and Technology, Tech. Rep. NIST IR 8320, 2021.

[9] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults (extended abstract)," in *Proceedings of the 16th International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, ser. Lecture Notes in Computer Science, vol. 1233. Springer, 1997, pp. 37–51.

[10] P. T. Breuer and J. P. Bowen, "Certifying machine code safe from hardware aliasing: RISC is not necessarily risky," in *Software Engineering and Formal Methods (SEFM)*, ser. Lecture Notes in Computer Science, vol. 8368. Springer, 2013, pp. 371–388.

[11] R. Buhren, S. Gueron, J. Nordholz, J. Seifert, and J. Vetter, "Fault attacks on encrypted general purpose compute platforms," in *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2017, pp. 197–204.

[12] R. Buhren, H. N. Jacob, T. Krachenfels, and J. Seifert, "One glitch to rule them all: Fault injection attacks against AMD's secure encrypted virtualization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021, pp. 2875–2889.

[13] R. Buhren, C. Werling, and J. Seifert, "Insecure until proven updated: Analyzing AMD SEV's remote attestation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 1087–1099.

[14] Z. Chen and D. Oswald, "PMFault: Faulting and bricking server CPUs through management interfaces or: A modern example of halt and catch fire," *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, vol. 2023, no. 2, pp. 1–23, 2023.

[15] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. F. Oswald, and F. D. Garcia, "VoltPillager: Hardware-based fault injection attacks against Intel SGX enclaves using the SVID voltage scaling interface," in *30th USENIX Security Symposium*, 2021, pp. 699–716.

[16] coreboot, "coreboot," https://review.coreboot.org/coreboot.git, Git Repository.

[17] Corsair, "RAM: How to enable SPD write on your ASUS Z690 motherboard." [Online]. Available: https://help.corsair.com/hc/en-us/articles/5718039999117-RAM-How-to-enable-SPD-Write-on-your-ASUS-Z690-motherboard

[18] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[19] T. W. David Kaplan, Jeremy Powell, "AMD memory encryption," October 2016.

[20] Dell, "How to Resolve a Signed BIOS Firmware Message on a Latitude, Precision or OptiPlex System," https://www.dell.com/support/kbdoc/en-uk/000126560/how-to-resolve-a-signed-bios-firmware-message-on-a-latitude-precision-or-optiplex-system, 2021.

[21] B. Dixon, "pin2pwn: How to root an embedded Linux box with a sewing needle," DEF CON 24, 2016.

[22] Z. Du, Z. Ying, Z. Ma, Y. Mai, P. Wang, J. Liu, and J. Fang, "Secure encrypted virtualization is unsecure," *arXiv preprint*, 2017.

[23] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the PENGLAI enclave," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021, pp. 275–294.

[24] A. Galanou, K. Bindlish, L. Preibsch, Y. Pignolet, C. Fetzer, and R. Kapitza, "Trustworthy confidential virtual machines for the masses," in *Proceedings of the 24th International Middleware Conference*, 2023, pp. 316–328.

[25] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 38–55.

[26] S. Gueron, "A memory encryption engine suitable for general purpose processors," *IACR Cryptology ePrint Archive*, 2016.

[27] H. Hartikainen, "Hacking DDR3 SPD," May 2018. [Online]. Available: https://hannuhartikainen.fi/blog/hacking-ddr3-spd/

[28] F. Hetzelt and R. Buhren, "Security analysis of encrypted virtual machines," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2017, pp. 129–142.

[29] B. D. Hopkins, J. Shield, and C. North, "Redirecting DRAM memory pages: Examining the threat of system memory hardware trojans," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016, pp. 197–202.

[30] A. Inoue, K. Minematsu, M. Oda, R. Ueno, and N. Homma, "ELM: A low-latency and scalable memory encryption scheme," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 2628–2643, 2022.

[31] Intel, "Supporting Intel SGX on multi-socket platforms," https://web.archive.org/web/20220822150148/https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf, Intel, White Paper, 2021.

[32] ——, "Xucode: An innovative technology for implementing complex instruction flows," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/xucode-implementing-complex-instruction-flows.html, 2021.

[33] ——, "Intel architecture memory encryption technologies," Intel, Manual 336907-004US, Rev. 1.4, August 2022.

[34] ——, "Architecture specification: Intel trust domain extensions (Intel TDX) module," Intel, Specification 344425-005US, Feb. 2023.

[35] ——, "Intel 500 series chipset family on-package platform controller hub," Intel, Datasheet 630747-015, Feb. 2023.

[36] ——, "Intel trust domain extensions," 2023.

[37] ——, "Intel trusted execution technology (Intel TXT)," Intel, Manual 315168-017, Rev. 017.4, April 2023.

[38] ——, "Intel Xeon scalable processors: NEX eagle stream platform, Intel platform security," Intel, Tech. Rep. 784473, August 2023.

[39] ——, "Intel processors supporting Intel SGX," https://web.archive.org/web/20240515140432/https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions-processors.html, 2024, accessed on May 15th, 2024.

[40] ——, "Intel Trust Domain Extensions (Intel TDX) module base architecture specification," Intel, Specification 348549-004US, Mar. 2024.

[41] P. Jattke, M. Wipfli, F. Solt, M. Marazzi, M. Bölcskei, and K. Razavi, "ZenHammer: Rowhammer attacks on AMD Zen-based platforms," in *33rd USENIX Security Symposium*, 2024.

[42] JEDEC, "Annex L: Serial presence detect (SPD) for DDR4 SDRAM modules," JEDEC, Standard 21-C, Section 4.1.2.L-6, Nov. 2020.

[43] ——, "Definition of the EE1002 and EE1002A serial presence detect (SPD) EEPROMS," JEDEC, Standard 21-C, Section 4.1.3, May 2022.

[44] ——, "Definitions of the EE1004-v 4 kbit serial presence detect (SPD) EEPROM and TSE2004av 4 kbit SPD EEPROM with temperature sensor (TS) for memory module applications," JEDEC, Standard 21-C, Section 4.1.6, May 2022.

[45] ——, "DDR5 serial presence detect (SPD) contents," JEDEC, Standard JESD400-5B, Oct. 2023.

[46] ——, "SPD5118 hub and serial presence detect device standard," JEDEC, Standard JESD300-5B.01, May 2023.

[47] D. Kaplan, "Protecting VM register state with SEV-ES," 2017.

[48] Z. Kemble, "Modifying RAM SPD data," Mar. 2016. [Online]. Available: https://blog.zakkemble.net/modifying-ram-spd-data/

[49] D. Lee, D. Jung, I. T. Fang, C. Tsai, and R. A. Popa, "An off-chip attack on hardware enclaves via the memory bus," in *29th USENIX Security Symposium*, 2020, pp. 487–504.

[50] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang, "A systematic look at ciphertext side channels on AMD SEV-SNP," in *43rd IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 337–351.

[51] M. Li, Y. Zhang, and Z. Lin, "CrossLine: Breaking "security-by-crash" based memory isolation in AMD SEV," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021, pp. 2937–2950.

[52] M. Li, Y. Zhang, Z. Lin, and Y. Solihin, "Exploiting unprotected I/O operations in AMD's secure encrypted virtualization," in *28th USENIX Security Symposium*, 2019, pp. 1257–1272.

[53] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "CIPHERLEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel," in *30th USENIX Security Symposium*, 2021, pp. 717–732.

[54] Linux, "The kernel's command-line parameters," https://www.kernel.org/doc/html/v6.9/admin-guide/kernel-parameters.html, 2024.

[55] Z. Liu, "Gigabyte motherboard firmware update: Saving your DDR5 RAM from corruption," Sep. 2023. [Online]. Available: https://www.tomshardware.com/news/gigabyte-motherboard-firmware-update-saving-your-ddr5-ram-from-corruption

[56] S. López, "libkrun," https://github.com/containers/libkrun, GitHub Repository.

[57] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *2nd ACM International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[58] Mitre, "CWE-1257: Improper access control applied to mirrored or aliased memory regions," https://cwe.mitre.org/data/definitions/1257.html, 2020.

[59] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *19th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[60] M. Morbitzer, M. Huber, and J. Horsch, "Extracting secrets from encrypted virtual machines," in *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2019, pp. 221–230.

[61] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, "SEVered: Subverting AMD's virtual machine encryption," in *Proceedings of the 11th European Workshop on Systems Security (EuroSec)*, 2018, pp. 1:1–1:6.

[62] M. Morbitzer, S. Proskurin, M. Radev, M. Dorfhuber, and E. Q. Salas, "SEVerity: Code injection attacks against encrypted virtual machines," in *IEEE Security and Privacy Workshops (S&P Workshops)*, 2021, pp. 444–455.

[63] K. Okupski, "Exploring AMD Platform Secure Boot," https://labs.ioactive.com/2024/02/exploring-amd-platform-secure-boot.html, 2024.

[64] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *25th USENIX Security Symposium*, 2016, pp. 565–581.

[65] D. Pontes, F. Silva, E. D. L. Falcão, and A. Brito, "Attesting AMD SEV-SNP virtual machines with SPIRE," in *12th Latin-American Symposium on Dependable and Secure Computing (LADC)*, 2023, pp. 1–10.

[66] P. Rogaway, "Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC," in *Proceedings of the 10th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, ser. Lecture Notes in Computer Science, vol. 3329. Springer, 2004, pp. 16–31.

[67] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 416–427.

[68] B. Schlüter, S. Sridhara, A. Bertschi, and S. Shinde, "WESEE: Using malicious #VC interrupts to break AMD SEV-SNP," in *45th IEEE Symposium on Security and Privacy (S&P)*, 2024.

[69] B. Schlüter, S. Sridhara, M. Kuhne, A. Bertschi, and S. Shinde, "HECKLER: Breaking confidential VMs with malicious interrupts," in *33rd USENIX Security Symposium*, 2024.

[70] J. Szefer and S. Biedermann, "Towards fast hardware memory integrity checking with skewed merkle trees," in *Hardware and Architectural Support for Security and Privacy (HASP)*, 2014, pp. 9:1–9:8.

[71] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing paging overheads in SGX with efficient integrity verification structures," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2018, pp. 665–678.

[72] B. M. S. B. Talukder, V. Menon, B. Ray, T. J. Neal, and M. T. Rahman, "Towards the avoidance of counterfeit memory: Identifying the DRAM origin," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 111–121.

[73] UPMEM, "UPMEM," https://www.upmem.com.

[74] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A practical attack framework for precise enclave execution control," in *2nd Workshop on System Software for Trusted Execution (SysTEX)*. ACM, Oct. 2017, pp. 4:1–4:6.

[75] S. Vig, R. Juneja, and S. Lam, "DISSECT: Dynamic skew-and-split tree for memory authentication," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 1538–1543.

[76] M. Wang, Z. Zhang, Y. Cheng, and S. Nepal, "DRAMDig: A knowledge-assisted tool to uncover DRAM address mapping," in *57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[77] J. Werner, J. Mason, M. Antonakakis, M. Polychronakis, and F. Monrose, "The SEVerESt of them all: Inference attacks against secure virtual enclaves," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2019, pp. 73–85.

[78] Wii Brew Wiki, "Tweezer Attack," https://wiibrew.org/wiki/Tweezer_Attack, 2023.

[79] L. Wilke and G. Scopelliti, "SNPGuard: Remote attestation of SEV-SNP VMs using open source tools," 2024, to appear at SysTEX '24.

[80] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth, "SEVurity: No security without integrity : Breaking integrity-free memory encryption with minimal assumptions," in *2020 IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 1483–1496.

[81] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, "SEV-Step A single-stepping framework for AMD-SEV," *IACR Tansactions on Cryptographic Hardware and Embedded Systems (CHES)*, vol. 2024, no. 1, pp. 180–206, 2024.

[82] R. Wojtczuk, "Analysis of the attack surface of Windows 10 virtualization-based security," Black Hat USA, 2016.

[83] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *36th IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 640–656.

[84] S. F. Yitbarek, M. T. Aga, R. Das, and T. M. Austin, "Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 313–324.

[85] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based fault injection using selective state reset," in *33rd USENIX Security Symposium*, 2024.

TABLE 5. DIMM CONNECTIONS TO INTERFACE WITH THE SPD ON DDR4 AND DDR5.

Pin	DDR4	DDR5 (UDIMM & SODIMM)	DDR5 (RDIMM)
SDA	285	5	5
SCL	141	4	4
Addressing	139, 140, 238	148	148
3.3 V in	284	151	3
5 V in	–	1	–
V_{ss}	283	6	6

TABLE 6. BILL OF MATERIALS FOR THE DDR4/DDR5 SETUP.

Component	Cost [$]
Raspberry Pi Pico	5
DDR4/DDR5 socket	1–5 each
Pull-up resistors	<0.1
9 V battery†	2

†Only for unlocking DDR4.

Appendix A. AMD Response

We include AMD's statement in response to our responsible disclosure below:

> AMD's public SEV-SNP whitepaper states invasive physical attacks are out-of-scope. However, due to the low cost of this physical attack, and the relative ease of implementing a mitigation, AMD has chosen to pursue a mitigation to improve customer security.
>
> To mitigate the vulnerability described in CVE-2024-21944, AMD is adding a basic assurance test to the boot process to ensure that DRAM address aliasing attack cannot be done using SPD spoofing. AMD Secure Boot loader firmware will measure the DRAM's response to address bits and take action to prevent SPD spoofing if the results don't match SPD address bit settings.
>
> AMD Firmware rollout has a complex software supply chain involving IBV, ODM/OEM and cloud providers. This is further complicated when a firmware fix requires system reboot. The rollout process will take some time for AMD to qualify the firmware update, which will then be released into the AMD Platform Initialization (PI) package for integration into customer BIOS.

Appendix B. SPD Setup

The I^2C connections to the SPD EEPROM are exposed on the DIMM. When installed in a system, these connections are used to connect to the SMBus or SidebandBus. However, they can also be used in an offline setup to access the EEPROM with a microcontroller. Table 5 provides the pin mapping for DDR4 and DDR5 to interface with the chip. Note that DDR5 requires different connections for RDIMMs and UDIMMs, as they operate at different voltages.

To modify the SPD contents, we use a Raspberry Pi Pico, which we connect to an additional DIMM socket to avoid soldering to the module's edge connectors directly. Note that these sockets are keyed differently for DDR4 and DDR5, as well as for DDR5 RDIMMs and DDR5 UDIMMs. Figure 2 shows this setup with a DDR5 RDIMM connected to a Raspberry Pi Pico. Table 6 provides the bill of materials and estimated component cost, totaling around $10.

When connecting the addressing pins to ground, the EEPROM will be assigned I^2C peripheral address `0x50`. Any write protection on DDR4 can be cleared by connecting SA0 (pin 139) to V_{HV} (*i.e.*, 7–10 V) and issuing the Clear all Write Protection (CWP) command by writing to peripheral address `0x33` [44]. For DDR5, connecting HSA (pin 148) to ground allows modifications to be made to MR12 and MR13, the registers holding the protection status [46] (cf. Figure 3). In both cases, these changes are persistent.

Appendix C. BadRAM attacks on DDR3

While we mainly considered attacks on DDR4 and DDR5 in this paper, the BadRAM primitive is, in principle, also applicable to older DDR generations as they all use the SPD to store their topology information. These older generations, however, do allow the manufacturer to set Permanent Software Write Protection (PSWP) to the SPD [43]. For DIMMs with this permanent protection set, performing BadRAM attacks would require either physically replacing the SPD chip or performing the attacks through the BIOS, as discussed in Section 6.1. Specifically for DDR3, the required modifications to the SPD content are identical to those required for DDR4, as the addressing encoding and CRC location did not change. The only notable difference is that DDR3 only supports up to 16 row address bits, compared to the 18 bits for DDR4 and DDR5. Additionally, the location of the module's serial number for DDR3 is stored in bytes 122 through 125, which may be required to be modified if the BIOS caches the SPD contents.

We evaluated the DDR3 BadRAM primitive on a Dell OptiPlex 990 DT with a CN-0VNP2H mainboard and an Intel Core i7-2600 with a single DDR3 UDIMM (HMT351U6BFR8C-H9) installed. We physically removed the SPD chip from the DDR3 DIMM as it had PSWP (cf. Table 9) and connected the exposed pads to a Raspberry Pi 3 Model B+, as shown in Figures 9 and 10. We then configured the I^2C interface of the Raspberry Pi to emulate an SPD EEPROM with modified addressing information to make the DIMM appear twice the size. On this system, the introduced ghost bit corresponded to the most significant physical address bit. This experiment shows that older DDR generations are also vulnerable to BadRAM attacks.

TABLE 7. WRITE PROTECTION STATUS AND ADDRESSING INFORMATION FOR VARIOUS DDR4 MODULES.

			Write Protection				Addressing Bits				
Manufacturer	Type	Serial Number	WP0	WP1	WP2	WP3	Rank	Bank	Row	Col.	Capacity [GB]
Corsair$_1$	UDIMM	CMV4GX4M1A2133C15	✗	✗	–	–	0	4	15	10	4
Corsair$_2$	UDIMM	CMK16GX4M2E3200C16	✗	✗	✗	✗	0	4	16	10	8
Crucial$_1$	SODIMM	CT16G4SFD824A.M16FE	✓	✓	✓	✗	1	4	16	10	16
Kingston$_1$	RDIMM	KSM32RS8/8HDR	✓	✓	✗	✗	0	4	16	10	8
Kingston$_2$	RDIMM	KSM32RS8L/16MFR	✓	✓	✗	✗	0	4	17	10	16
Kingston$_3$	RDIMM	KTH-PL432/16G	✓	✓	✗	✗	0	4	17	10	16
Kingston$_4$	UDIMM	KVR26N19S8/8	✓	✓	–	–	0	4	16	10	8
Micron$_1$	RDIMM	MTA36ASF4G72PZ-3G2R1	✓	✓	✓	✗	1	4	17	10	32
Micron$_2$	RDIMM	MTA36ASF8G72PZ-3G2F1	✓	✓	✓	✗	1	4	18	10	64
Micron$_3$	RDIMM	MTA9ASF1G72PZ-3G2E2	✓	✓	–	–	0	4	16	10	8
Micron$_4$	RDIMM	MTA18ASF2G72PZ-2G9J3R	✓	✓	✓	✗	0	4	17	10	16
Samsung$_1$	UDIMM	M378A2K43DB1-CTD	✓	✓	✗	✗	1	4	16	10	16
Samsung$_2$	SODIMM	M471A2K43EB1-CWE	✓	✓	✗	✗	1	4	16	10	16
SK hynix$_1$	RDIMM	HMA82GR7DJR4N-XN	✓	✓	✗	✗	0	4	17	10	16
SK hynix$_2$	RDIMM	HMAA4GR7AJR8N-XN	✓	✓	–	–	1	4	17	10	32
SK hynix$_3$	UDIMM	HMA41GU6AFR8N-TF	✓	✓	✗	✗	1	4	15	10	8
SK hynix$_4$	UDIMM	HMA82GU6JJR8N-VK	✓	✓	✗	✗	1	4	16	10	16
SK hynix$_5$	SODIMM	HMA41GS6AFR8N-TF	✓	✓	✗	✗	1	4	15	10	8

TABLE 8. WRITE PROTECTION STATUS AND ADDRESSING INFORMATION FOR VARIOUS DDR5 MODULES.

			Write Protection		Addressing Bits				
Manufacturer	Type	Serial Number	MR12	MR13	Rank	Bank	Row	Col.	Capacity [GB]
Kingston$_5$	RDIMM	KF548R36RB-16	0xff	0x3c	0	5	16	10	16
Kingston$_6$	RDIMM	KSM48R40BS8KMM-16HMR	0xff	0x00	0	5	16	10	16
Kingston$_7$	RDIMM	KSM48R40BD8KMM-32HMR	0xff	0x00	1	5	16	10	32
Samsung$_3$	RDIMM	M321R2GA3BB6-CQK	0xff	0x01	0	5	16	10	16
SK hynix$_6$	RDIMM	HMCG78MEBRA115N	0xff	0x01	0	5	16	10	16
SK hynix$_7$	UDIMM	HMCG78AEBUA084N	0xff	0x01	0	5	16	10	16

TABLE 9. WRITE PROTECTION STATUS AND ADDRESSING INFORMATION FOR THE DDR3 MODULE USED IN APPENDIX C.

					Addressing Bits				
Manufacturer	Type	Serial Number	SWP	PSWP	Rank	Bank	Row	Col.	Capacity [GB]
SK hynix$_8$	UDIMM	HMT351U6BFR8C-H9	–	✓	1	3	15	10	4

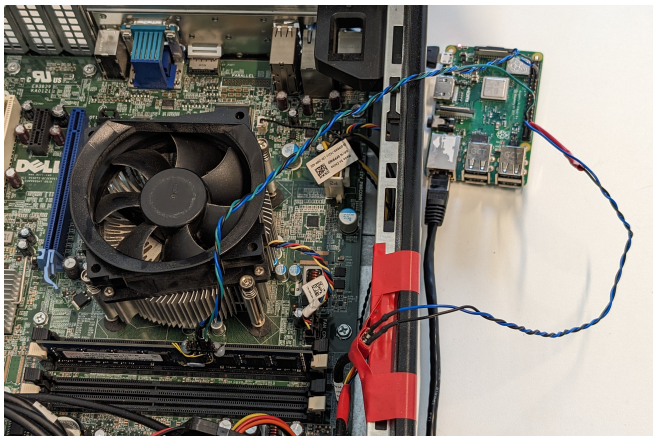

Figure 9. DDR3 setup to perform BadRAM attacks. The DDR3 DIMM has its SPD chip removed and is instead connected to a Raspberry Pi. The additional wires are connected to the power button, but are not required for the attack.

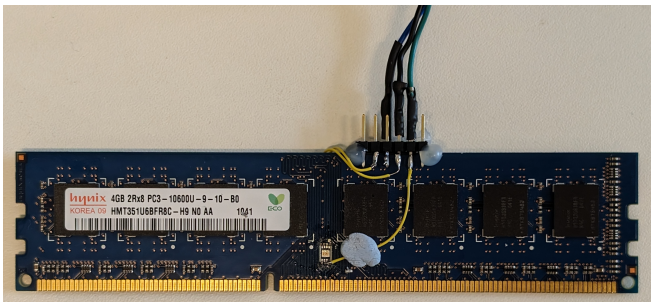

Figure 10. Closeup of the removed SPD chip from the DIMM in Figure 9. Note that some connections are made to the vias on the backside.

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

This paper demonstrates an attack that exploits the lack of authentication and protection of DIMM information that is stored and retrieved via the Serial Presence Detect interface to bypass the memory integrity guarantees of AMD SEV-SNP. The paper shows how manipulation of the information (via reprogramming through the SPD interface) can result in an incorrect view of available memory leading to memory aliasing.

D.2. Scientific Contributions

- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field
- Independent Confirmation of Important Results with Limited Prior Research
- Establishes a New Research Direction
- Creates a New Tool to Enable Future Science

D.3. Reasons for Acceptance

1) This paper identifies an impactful vulnerability. This paper demonstrates an architectural gap in AMD SEV-SNP that allows bypassing memory integrity protections for confidential VMs. The paper exploits the SPD interface available on DIMMs that allow re-programming of DIMM metadata (incompliance with the JEDEC standard) to present the host SoC with a 'bad' view of memory that leads to memory aliasing. The memory aliasing is then used to manipulate the RMP table that holds security metadata for individual memory pages and thereby, bypass the protections affered by the RMP. The paper demonstrates and end-to-end attack where the measurement in the attestation doesn't match the measurement of the actual memory content of a confidential VM.
2) This paper provides a valuable step forward in an established field and independently confirms results with limited prior research. Memory aliasing attacks by manipulation of DIMM metadata have been known for a while. Intel SGX and TDX provide protections against such attacks through a proprietary set of checks (whose details are not public). This paper actually confirms that the checks implemented by these two technologies are effective against the attacks outlined in the paper which provides for the first time, independent verification of the security claims. The paper also explores for the first time, the SPD interface, requirements underlying the SPD interface that allow reprogramming of DIMM metadata by the JEDEC standard as well as effects of manipulating the DIMM metadata via the SPD interface. Since there are no existing mechanisms/standardized ways to protect this DIMM metadata, the identified vulnerabilities will need a workaround (just like TDX and SGX do). The paper also demonstrates the viability of affecting the security posture of a TEE whose trust boundary is confined to the SoC via corruption of a system/platform component that is likely vulnerable to supply chain attacks.
3) This paper establishes a new research direction. Existing countermeasures to the outlined attacks have been proprietary (and emerging ones will likely also be so) to detect memory aliasing. Changing DIMM standards to include more systematic countermeasures will likely have a long tail. So, this paper highlights the need to devise mechanisms that can work with existing DIMM standards in the public domain—ones that lend themselves to systematic analysis instead of a heuristic that relies on randomly checking for aliases but still scale for use in cloud settings.
4) The paper creates a new tool for future science: The authors are committed to making their attack framework available for other researchers to explore other offensive and defensive countermeasures.