DDRop: Active Memory Interposer Attacks on Confidential VMs by Dropping DDR5 Writes

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

Stefan Gloor
stefan.gloor@alumni.ethz.ch
ETH Zurich
Zurich, Switzerland

Patrick Jattke
pjattke@ethz.ch
ETH Zurich
Zurich, Switzerland

Daniel Moghimi
danielmm@google.com
Google
California, United States

David Oswald
david.f.oswald@durham.ac.uk
Durham University
Durham, United Kingdom

Martin Thompson
martin.j.thompson@durham.ac.uk
Durham University & ZF Automotive UK Ltd
Durham, Solihull, United Kingdom

Kaveh Razavi
kaveh@ethz.ch
ETH Zurich
Zurich, Switzerland

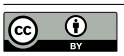
Ingrid Verbauwhede
ingrid.verbauwhede@esat.kuleuven.be
COSIC, KU Leuven
Leuven, Belgium

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

Abstract

Trusted Execution Environments (TEEs) are increasingly deployed in the cloud to protect sensitive workloads through hardware-enforced isolation, remote attestation, and transparent memory encryption. However, to meet memory performance and size demands, modern TEEs omit cryptographic freshness guarantees, leaving them vulnerable to replay attacks by adversaries with physical memory access. Prior work demonstrated low-cost active interposition attacks on DDR4 without requiring expensive specialized equipment, but these techniques do not extend to DDR5, where existing approaches are limited to passive ciphertext side-channel analysis and rely on bus downclocking to accommodate legacy memory bus analyzers.

We present the first low-cost (<200$) DDR5 RDIMM interposer capable of active fault injection at native speeds. By injecting targeted parity errors to silently discard cache line writebacks, we introduce DDRop, a new primitive that exploits the absence of cryptographic freshness to break the integrity of Intel TDX, Scalable SGX, and AMD SEV-SNP. Building on this primitive and targeting the APIs exposed by the TDX module and AMD Secure Processor, we show that adversaries can gain ciphertext access, copy arbitrary victim pages, and inject malicious secure page-table entries. We demonstrate end-to-end attacks on an up-to-date TDX platform, including forcing any TD into debug mode and forging attestation reports. While software-level mitigations, including timing-based interposer detection and API hardening, may reduce the attack surface, our results demonstrate that active DDR5 bus interposition is practical at low cost, highlighting the need for robust cryptographic memory integrity protections against physical adversaries.



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846578

CCS Concepts

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

Keywords

Hardware Security, DRAM, TEE, Confidential Computing

ACM Reference Format:
Jesse De Meulemeester, Stefan Gloor, Patrick Jattke, Daniel Moghimi, David Oswald, Martin Thompson, Kaveh Razavi, Ingrid Verbauwhede, and Jo Van Bulck. 2026. DDRop: Active Memory Interposer Attacks on Confidential VMs by Dropping DDR5 Writes. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846578

1 Introduction

Confidential computing aims to address the fundamental security and privacy concerns in multi-tenant cloud environments. Unlike traditional cloud architectures where the provider retains unrestricted access to deployed code and data, Trusted Execution Environments (TEEs), like Intel Software Guard Extensions (SGX) [26, 34], Intel Trust Domain Extensions (TDX) [21], AMD Secure Encrypted Virtualization (SEV) [27], and Arm Confidential Compute Architecture (CCA) [6], enforce hardware-level memory access control and transparent memory encryption to protect tenant workloads from the cloud provider itself.

Robust memory encryption designs, such as the one in Intel's Client SGX [16], provide confidentiality against bus snooping attacks [11, 46] and cold boot attacks [55], alongside cryptographic integrity and freshness to defeat ciphertext side channels [29, 31, 32] and active physical interposers [13]. However, enforcing cryptographic integrity and freshness requires additional metadata to be stored and verified on every memory access, severely restricting the size of the protected memory region in practice [16, 26]. Consequently, modern scalable TEEs, including Intel Scalable SGX, Intel TDX, AMD SEV, and Arm CCA, provide only confidentiality, or

at most limited cryptographic integrity, in order to support large workloads and lift-and-shift Confidential VMs (CVMs).

Modern deterministic memory encryption designs are, hence, theoretically exposed to adversaries with physical access who can exploit the absence of cryptographic freshness and integrity. While such adversaries were initially assumed to require high-speed logic analyzers prohibitively priced upwards of $100,000 [29], a recent line of work has demonstrated custom, *low-cost* physical interposers [11, 13, 14, 46]. One approach introduces aliases that can be used to modify or replay encrypted memory locations. While modern TEEs include trusted firmware checks that prevent static DRAM address aliasing [3, 14, 26], active hardware interposers have been shown to defeat boot-time checks by dynamically manipulating the memory address bus at runtime [13]. Importantly, however, these attacks—which rely on simple, low-speed address-bit manipulations—have so far only been demonstrated on DDR4 [13]. DDR5 introduced a more complex, multi-cycle command encoding that renders this approach fundamentally infeasible.

A concurrent line of work on *passive* interposers [11, 46], on the other hand, exploits a separate consequence of the missing freshness protections: observing repeated accesses to the same memory location reveals whether its contents have changed via the ciphertext side channel [31, 56]. To reduce costs, prior work relied on intrusive downclocking of the memory bus to the lowest DDR5 frequency bin at boot time in order to accommodate legacy second-hand logic analyzers [11]. Such drastic frequency reduction is potentially detectable, however, for example, by extending the existing boot-time alias checks that already validate DRAM configurations. Moreover, ciphertext side channels can be mitigated at the software or compiler level [4, 40, 51], or might conceivably be mitigated via modest hardware changes [41], such as the "cache line versioning" mitigation Intel recently announced as a possible mitigation for their next-generation platforms [25].

This Work. Given this context, we aim to address the following fundamental research questions: *What does it take to mount active interposer attacks on DDR5 memory? What is the cost? What are the implications for today's confidential computing architectures, and what are possible mitigations?*

We present the first *active* DDR5 Registered DIMM (RDIMM) interposer to directly exploit the lack of cryptographic freshness protections in scalable memory encryption designs. Building on open-source Rowhammer research for DDR5 Unbuffered DIMMs (UDIMMs) [15], we construct a low-cost, <200$ custom DDR5 RDIMM interposer. In contrast to prior passive DDR5 interposers [11], our interposer operates stealthily at native DDR5 speeds and actively manipulates the Command/Address (CA) bus. Sitting between the CPU and Dynamic Random-Access Memory (DRAM), our interposer can be toggled at runtime to induce rank-level aliasing or, more generally, inject parity errors to drop any command sent to commercial off-the-shelf DDR5 RDIMMs. Due to the lack of freshness, our primitive, which we dub DDRop, causes dirty cache lines to be silently discarded, forcing the victim process to read back stale memory contents.

We evaluate DDRop against the latest, fully-updated Intel TDX, Scalable SGX, and AMD SEV-SNP platforms, demonstrating that all three confidential computing technologies are affected. Furthermore, by targeting the trusted API exposed to the malicious hypervisor, we construct CVM-agnostic attack primitives that do not require victim-specific gadgets. Concretely, we show how an active adversary can exploit page relocation to access victim ciphertext and copy arbitrary victim pages, and leverage TDX's Secure Extended Page Table (SEPT) architecture to inject malicious SEPT entries. We demonstrate practical end-to-end attacks on Intel TDX, including switching any Trust Domain (TD) into debug mode and spoofing arbitrary attestation reports.

Our results demonstrate that high-speed, active memory interposers are a practical threat to modern cloud infrastructure. While software-based API hardening or probabilistic delay detection may raise the bar, our work exposes a fundamental limitation in today's scalable memory encryption designs: the absence of cryptographic freshness. Addressing this limitation ultimately requires hardware changes, calling for more robust yet scalable memory encryption designs in next-generation confidential computing architectures.

Contributions. In summary, our main contributions are:

(1) We design a low-cost (<$200), active hardware interposer that reliably drops writes to DDR5 memory modules on demand at native bus speeds.
(2) We introduce the DDRop primitive and show how it breaks freshness guarantees for encrypted memory on Intel TDX, Scalable SGX, and AMD SEV-SNP.
(3) We use DDRop to exploit the trusted API exposed by the TDX module and AMD Secure Processor, creating practical end-to-end attacks without requiring CVM-specific gadgets.
(4) We discuss implications for effective countermeasure design and evaluate a preliminary probabilistic detection approach.

Responsible Disclosure. We disclosed our findings to both Intel and AMD, with further details provided in Appendix B.

2 Background and Related Work

We provide the necessary background on DDR5 memory (Section 2.1) and modern TEEs (Section 2.2) to understand the design and implications of our active interposer attack.

2.1 DDR5 Command Encoding

DDR5 Dual In-line Memory Modules (DIMMs) are organized into two independent 32-bit subchannels, with up to two package ranks, selected via dedicated Chip Select (CS) lines. Each rank comprises multiple DRAM chips structured into up to 32 banks of rows and columns. To maximize memory bandwidth, the memory controller usually interleaves subsequent cache lines across all available channels, subchannels, and ranks. This interleaving is typically confined within a single Non-Uniform Memory Access (NUMA) node and, by default, does not span across physical sockets.

Unlike previous generations, DDR5 utilizes a multi-cycle command encoding. The exact encoding depends on the memory type: commands take 2 or 4 cycles over the 14- or 7-bit-wide Command/Address (CA) bus for consumer UDIMMs and server-grade RDIMMs, respectively. Figure 1 details the command encoding on DDR5 RDIMMs for write operations. The full encoding is included in Figure 11 in Appendix D.

			CA						
	CK	$\overline{CS}$	0	1	2	3	4	5	6
WR	L	L	H	L	H	H	L	$\overline{BL}$	BA0
	H	L	BA1	BG0–2			CID0–1		CID2/ DDPID
	L	H	V	Column 3–8					
	H	H	C9–10		V	$\overline{AP}$	$\overline{WP}$	V	CID3

Figure 1: Encoding for writes on DDR5 RDIMMs [23]. Each line can be high (H), low (L), or valid (V).

To maintain signal integrity, RDIMMs feature a Registering Clock Driver (RCD) that buffers the CA, CS, and clock lines between the CPU and DRAM chips. Additionally, the RCD enforces parity checking on the CA bus, detecting corrupted commands, and alerting the CPU through the ALERT line.

2.2 Trusted Execution Environments

Intel TDX. Intel Trust Domain Extensions (TDX) [20, 21] is a hardware-based confidential computing technology that isolates virtual machines, known as Trust Domains (TDs), from the untrusted host. It introduces a new CPU mode, Secure Arbitration Mode (SEAM), which hosts the TDX module—a secure, Intel-provided software component that acts as a trusted intermediary between the Virtual Machine Monitor (VMM) and guest TDs. To protect TD data stored in external memory, TDX relies on Total Memory Encryption - Multi-Key (TME-MK), which uses AES in Ciphertext Stealing (XTS) mode with an address-based tweak and domain-specific keys [22]. Furthermore, TDX enforces strict access control via the Secure Extended Page Table (SEPT)—managed by the TDX module—preventing the hypervisor from accessing TD-private memory or mapping unauthorized pages into the TD's address space.

A key feature of TDX memory protection is its approach to handling integrity. By default, TDX provides a logical integrity mode, in which each cache line repurposes an Error-Correcting Code (ECC) bit to store the TD Owner metadata flag. This bit partitions memory into private and shared regions, and is checked by the memory controller on every read. For defense-in-depth against DRAM-level attacks such as Rowhammer or physical memory tampering, TDX optionally supports a cryptographic integrity mode. When enabled, the memory controller computes a 28-bit cryptographic Message Authentication Code (MAC) for each cache line, stored in the ECC bits. This MAC is computed over the ciphertext, Host Physical Address (HPA), and ephemeral memory key, ensuring protection against aliasing attacks. To minimize performance overhead, TDX employs a deferred verification strategy: if a MAC check fails, the memory controller marks the cache line as poisoned, and a fatal exception is triggered only when the CPU attempts to consume data from that line. Crucially, while this architecture provides confidentiality and integrity, it fundamentally lacks cryptographic freshness, meaning it cannot detect stale ciphertexts.

Establishing trust in a TDX environment relies on a secure hardware foundation, a robust remote attestation mechanism, and the TDX module. During platform initialization, a hardware-controlled check known as MCHECK verifies the CPU configuration and memory ranges (such as convertible memory ranges), storing these measurements securely. The TDX module then uses this verified environment to manage TD lifecycles and enforce access policies. To prove its trustworthiness to external parties, TDX relies on remote attestation. During this process, the TDX module generates a cryptographic report containing measurements of the TD's build state and configuration (the Build-Time Measurement Register of TD, or MRTD). This report is signed by a hardware-backed platform-specific key, producing a verifiable quote that a remote verifier can use to assess the integrity and authenticity of the TD before provisioning sensitive data.

Table 1: DDRop is the first active interposer attack on DDR5 RDIMMs, exploiting the lack of cryptographic freshness to break both the confidentiality (C) and integrity (I) of modern scalable memory encryption designs.

	Cost	Mechanism	Mem.	Active	Target			Impact	
					SGX	TDX	SEV	C	I
DDRop	<$200	**Write suppr.**	**DDR5**	●	●	●	●	●	●
TEE.fail [11]	<$1,000†	Bus snooping	DDR5	○	●	●	●	●	○
Battering RAM [13]	<$50	Aliasing	DDR4	●	●	○	●	●	●
WireTap [46]	<$1,000†	Bus snooping	DDR4	○	●	○	○	●	○
Membuster [29]	~$170,000	Bus snooping	DDR4	○	◐	○	○	◐	○

† Using legacy logic analyzers purchased on secondhand marketplace.

AMD SEV-SNP. AMD SEV-SNP [2] is AMD's third-generation hardware-based confidential computing technology. In comparison to previous generations (SEV and SEV-ES), SEV-SNP adds validation of guest-physical address translations to defend against a malicious hypervisor. The primary goal is to prevent attacks such as data replay, memory remapping, and memory aliasing. Central to SEV-SNP is the Reverse Map Table (RMP), a hardware-checked data structure that tracks page ownership and permissions to ensure that only the designated guest Virtual Machine (VM) can access its private memory pages and that the hypervisor cannot map a physical page to multiple guest addresses or read guest-private memory. SEV-SNP provides static memory encryption using AES in XOR-Encrypt-XOR (XEX) mode with address-based tweaks and per-guest keys, without cryptographic integrity or freshness guarantees.

Beyond guest-physical address validation, SEV-SNP introduces enhanced access control and attestation mechanisms. It supports Virtual Machine Permission Levels (VMPLs), which enable privilege separation within the guest VM (e.g., isolating a security monitor from the guest OS). For remote attestation, SEV-SNP allows a guest to request a cryptographic report from the AMD Secure Processor (SP). This report includes measurements of the guest's initial state, its configuration, and the current platform Trusted Computing Base (TCB) version. The report is signed by a versioned chip-specific key derived from hardware fuses, enabling a remote verifier to validate the integrity and authenticity of the SEV-SNP guest.

2.3 Related Work

Passive Interposers. As summarized in Table 1, TEEs have been shown to be vulnerable to passive bus snooping. Membuster demonstrated how Client SGX traffic could be captured with a commercial DDR4 interposer and logic analyzer [29]. While Client SGX features counter-based encryption, the unencrypted address

bus allowed them to observe secret-dependent victim addresses. WireTap [46] targets Intel Scalable SGX, demonstrating that the weakened memory encryption enables ciphertext side-channel attacks, ultimately extracting SGX attestation keys. Following this, TEE.fail [11] targets DDR5 memory, expanding the scope to CVMs, including SEV-SNP and Intel TDX. However, these passive approaches are inherently limited to application-specific side-channel leakage originating from the lack of ciphertext randomization, which may be mitigated generically at the software or compiler levels. Furthermore, they either rely on expensive, high-speed logic analyzers or on bus downclocking, which could, in principle, be detected by trusted firmware like Intel's MCHECK. In contrast, DDRop uses inexpensive hardware to exploit the fundamental lack of cryptographic freshness, rendering software mitigations ineffective.

Active DRAM Attacks. Investigating hardware trojans, Hopkins et al. demonstrate how an FPGA interposer can actively redirect memory pages for DDR3 DIMMs [18]. However, they rely on extensive downclocking, making it incompatible with DDR4 and DDR5. In the context of Rowhammer, mFIT shows how a DDR4 RDIMM interposer can be used to suppress refresh commands [12]. Their technique, however, is specific to the DDR4 command encoding, and does not extend to DDR5. REFault similarly suppresses refreshes to enable Rowhammer on DDR5 UDIMMs, using an interposer to transform DRAM commands by tampering with the CA bus [15]. Because RDIMMs feature multi-cycle command encoding and parity checking, REFault does not translate to server memory.

Targeting CVMs, BadRAM [14] demonstrates practical memory aliasing attacks against SEV-SNP by maliciously modifying the DIMM's Serial Presence Detect (SPD) chip. The authors create alias mappings that map distinct physical addresses to the same DRAM location, allowing the attacker to bypass memory isolation without requiring complex hardware modifications. This aliasing can enable arbitrary code execution in the AMD SP [47], and has been shown to affect Arm TrustZone [17] and RISC-V PMP [33]. Battering RAM [13] bypasses the boot-time checks introduced to mitigate BadRAM by using a DDR4 DRAM interposer to create dynamic memory aliases on Intel Scalable SGX and AMD SEV-SNP.

Crucially, these attacks do not extend to Intel TDX: commercial TDX deployments only support DDR5 and protect against memory aliasing attacks via the proprietary MCHECK firmware that detects misconfigurations. Furthermore, higher bus speeds and complex command encoding make simple bit manipulations to create address aliasing impractical on DDR5 RDIMMs. In this paper, DDRop overcomes these limitations by relying on write suppression, compromising the integrity of scalable memory encryption designs.

Fault Attacks on Confidential Computing. Plundervolt [38], VoltJockey [42], and V0LTpwn [28] show how software adversaries can undervolt the CPU via control registers, inducing faults in Client SGX. Following this, VoltPillager demonstrates how a hardware adversary can send malicious command packets to the voltage regulator to achieve similar attacks [10]. Chen et al. similarly show how under- and overvolting attacks can be carried out from the baseboard management controller [9]. Looking at SEV-SNP, Buhren et al. show how voltage glitching can be used to leak the endorsement keys, enabling spoofed attestation reports [8]. CacheWarp [57] abused the fact that older versions of SEV-SNP allowed cache-invalidation instructions to be used to drop dirty cache lines of CVMs, which has since been mitigated. In contrast, DDRop demonstrates precise and deterministic memory-bus-level faults affecting the integrity of the latest, up-to-date SEV-SNP and Intel TDX.

Attacks Exploiting or Using the Trusted API. Beyond physical and fault attacks, recent work has exposed vulnerabilities that leverage or exploit the API exposed by the AMD SP or TDX module. Heracles [44] and Relocate-Vote [54] demonstrate that the page relocation API in AMD SEV-SNP can be abused to perform ciphertext side-channel attacks. In response, AMD introduced a guest policy to optionally disable move or swap operations on guest pages [4]. RMPocalypse shows how insufficient memory barriers allow dirty cache lines to persist during RMP initialization, overwriting the RMP, and enabling arbitrary ciphertext read/write access to guest memory [43]. Similarly, in Fabricked, malicious interconnect configurations are exploited to block writes during RMP initialization [45].

Aktas et al. perform a security analysis on TDX, identify several flaws in the TDX module API, and discuss how the page block/unblock API can be abused to perform controlled-channel attacks [1]. Building on this, TDXray uses this API to collect page-granular TD access traces [19]. Swidowski et al. extend the security analysis to TDX module 1.5 and build TDXplore, a toolkit to interact with the TDX module from userspace [49].

While the above vulnerabilities mostly relied on implementation flaws since addressed through firmware mitigations, DDRop introduces the first practical hardware attack that weaponizes the trusted API of TDX and SEV without relying on logical bugs.

3 DDRop Interposer

This section describes the design of our active DDR5 memory interposer, presenting the threat model, primitives, and interposer.

3.1 Threat Model

We adopt a threat model consistent with prior interposer attacks on confidential computing [11, 13, 46], assuming an adversary with privileged software execution and physical access to the target system. Privileged software access aligns with the standard TEE threat model, assuming root privileges on the host OS, access to the system configuration (e.g., BIOS), and the ability to launch attacker-controlled enclaves or CVMs. Additionally, the adversary requires brief, one-time physical access to the victim server—such as during routine maintenance—to install the interposer.

Unlike previous DDR5 attacks that require bulky logic analyzers and detectable memory downclocking [11], our custom interposer is highly compact, operates at native DDR5 speeds, and can be installed in a few minutes, leaving no immediate footprint in the system's boot or operations. The interposer does require a connection to an attacker-controlled system to orchestrate the attack; however, this could conceivably be the compromised target system itself, as the host is already assumed to be under attacker control.

We target dual-socket machines with DDR5 memory. These represent the majority of cloud deployments and can support both Intel Scalable SGX/TDX and AMD SEV-SNP. As memory is typically not interleaved across NUMA nodes, this ensures the interposer affects only a single socket, avoiding critical memory regions.

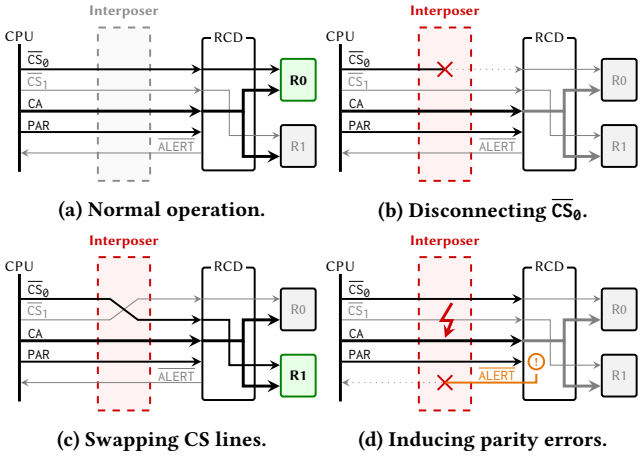


Figure 2: Schematic representation of the three primitives through which a malicious interposer can interfere with DDR5 RDIMM commands. The RCD buffers the commands between the CPU and the ranks (R0 and R1). In single-rank memory modules, only rank 0 is connected.

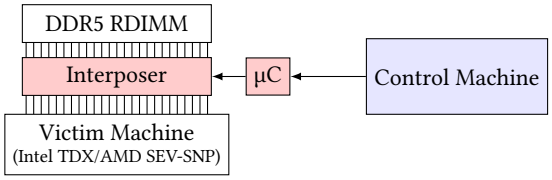


Figure 3: The interposer manipulates signals between the victim machine and the unmodified DDR5 memory via an attacker-controlled Microcontroller (µC).

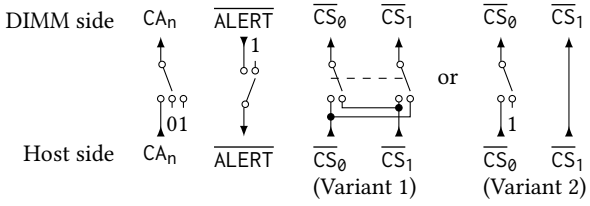


Figure 4: Interposer switches, which swap the CS lines (Variant 1) or disconnect them (Variant 2).

Hardware-Software Coordination. Mounting DDRop requires coordinating the hardware interposer with privileged host software. No fine-grained synchronization or single-stepping [50, 52] is required: we toggle the interposer manually before and after the targeted API call or CVM invocation. Some application-specific attacks require insight into the guest virtual-to-physical address mappings, which may be inferred through for instance page faults [19, 36] or avoided entirely by instead targeting CVM metadata structures with physical addresses known to the malicious host (cf. Sections 6.2 and 6.3).

3.2 Interfering with DDR5 Command Encoding

Prior work introduced dynamic memory aliases by physically changing the address bits transmitted over the memory bus [13]. Their technique relies on the observation that certain DDR4 CA lines only carry address information. However, due to changes in command encoding, this observation does not extend to DDR5. Unlike DDR4, which used single-cycle commands that enabled low-cost tampering, DDR5 introduces multi-cycle commands, spanning two or four cycles for UDIMMs and RDIMMs, respectively. Consequently, manipulating the transmitted address by grounding CA lines, as demonstrated by the Battering RAM attack [13], is no longer viable, as grounding two CA lines to bypass single-bit parity checking would affect eight logical bits of the transmitted command on a DDR5 RDIMM module. This would corrupt the transmitted command, making precise address manipulation impossible. Instead, we explore three alternative primitives to interfere with DDR5 DRAM addressing.

Disconnecting Ranks. On multi-rank DIMMs, command lines are shared across all ranks. The chip-select lines, $\overline{\text{CS}}_0$ and $\overline{\text{CS}}_1$, dictate which rank responds to the issued command, as shown in Figure 2a. If the corresponding CS line is not asserted, the rank simply ignores the command. A physical adversary can exploit this by severing a CS line, dropping any command routed to that rank (see Figure 2b).

Swapping Ranks. Alternatively, an adversary can physically swap the two CS lines. As these lines select which rank should respond to the request, swapping them redirects the command to the other rank. This effectively allows an adversary to read from or write to an address other than that originally issued by the CPU (see Figure 2c).

Parity Errors. Besides tampering with the chip select lines, a third approach involves deliberately injecting parity errors into the CA bus. Where previous physical attacks viewed the parity signal as an obstacle [13], an adversary can weaponize this mechanism to drop commands. When the RCD detects a parity error, it rejects the command and asserts the $\overline{\text{ALERT}}$ line to signal a transmission error to the CPU, prompting it to re-issue the command. By physically disconnecting the $\overline{\text{ALERT}}$ line, the adversary ensures the RCD silently drops the command without informing the CPU (see Figure 2d).

3.3 Interposer Design

To support the primitives introduced in the previous section, the interposer must be capable of electrically manipulating DDR5 bus signals on the fly (Figure 3). Given the demanding timing and signal-integrity constraints, the interposer design can support only a limited number of manipulations on a subset of the bus signals. In particular, we decide to implement the capability to swap $\overline{\text{CS}}_0$ and $\overline{\text{CS}}_1$, force CA lines high or low, and disconnect $\overline{\text{ALERT}}$ on the interposer. Further, the rank-swapping switches on the interposer Printed Circuit Board (PCB) can be modified to support disconnecting $\overline{\text{CS}}_0$ and forcing it high (inactive), instead of swapping it with $\overline{\text{CS}}_1$ (see Appendix C). With this, we can achieve two interposer variants covering all desired primitives. Figure 4 shows a schematic representation of the interposer switches for both variants.

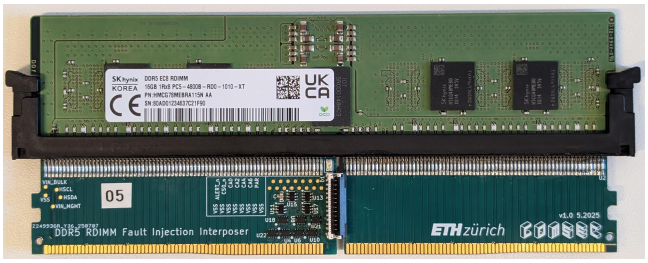


Figure 5: Our custom DDR5 RDIMM interposer with a memory module installed. Analog switches pin the CA lines and tamper with the CS lines; the flat-flex connector on the front connects to the microcontroller.

Table 2: BOM for a single interposer system, totaling $159.

	Item	Supplier	Price ($)
Interposer	PCB with stencil	JLCPCB	45
	Electronic parts	Digikey, LOTES	30
Controller Board	PCB	JLCPCB	4
	Electronic parts	Digikey	40
	Teensy 4.1	Digikey	40

Implementation. We base our design on REFault [15], which presents an interposer for DDR5 UDIMMs. We use the same solid-state high-frequency switches (TMUX136) to disconnect or swap bus signals. In contrast to RDIMMs, UDIMMs have a wider command bus, consisting of 14 bits rather than 7. Additionally, they lack parity signaling, allowing single command bits to be flipped without raising an alert. We port the REFault interposer to RDIMMs by implementing the necessary hardware to support the desired primitives (Section 3.2), accounting for RDIMM pinout and mechanical dimensions, and installing the switches on a single subchannel.

PCB Design. Since DDR5 operates at high data rates, the PCB design is not trivial. We verified that individual traces have the same length in order to maintain equal propagation delays among the signals. Further, we account for the physical PCB stackup to estimate trace impedances and reduce reflections. In Table 2, we provide the Bill Of Materials (BOM) and the total cost to build a single interposer system (at a total quantity of 10 systems). This does not include research and development costs or manual labor required to assemble and test the system.

Figure 5 shows a DDR5 memory module installed on our custom interposer. The switches are connected via a flat-flex cable to a Teensy 4.1 microcontroller board mounted on a custom baseboard. This device drives the switches and controls the interposer's behavior. This microcontroller is under the attacker's control and is used to synchronize the interposer with the attacker's workload.

4 Evaluation of Interposer

We first evaluate the performance and feasibility of the DDR5 interposer in isolation, before mounting practical attacks on real-world TEEs in the next sections.

4.1 Experimental Setup

Table 3 summarizes the evaluation systems, shown in Figure 6, used throughout this paper. Our evaluation is split into two stages. We first evaluate the basic physical primitives discussed in Section 3.2 using a single-socket testbench (Intel_1). While this is a single-socket machine, the BIOS allows memory interleaving to be limited to only four DIMMs instead of eight, ensuring the interposer remains isolated to a limited physical memory range.

For the subsequent end-to-end evaluation of DDRop against CVMs, we use two dual-socket servers supporting Intel TDX, Scalable SGX, and AMD SEV-SNP (Intel_2 and AMD_1, respectively). To maintain host stability during exploitation, we specifically target dual-socket systems. Because memory is typically not interleaved across NUMA nodes, these systems naturally provide two isolated, contiguous memory ranges, thus limiting the impact of the interposer to a single socket. We interpose a single memory module on the second socket, and reserve this socket's memory range through the `memmap` kernel parameter. This ensures that all critical memory regions (e.g., kernel memory) and default memory allocations are exclusively assigned to the first, unaffected socket. We apply custom kernel patches to both hosts to mark the second socket as usable for TDX and SEV-SNP while keeping it reserved through `memmap`. For Scalable SGX, we repurpose the modified `isgx` driver from Battering RAM [13], which provides control over Enclave Page Cache (EPC) page allocations. By selectively allocating victim memory pages on the affected socket, the adversary obtains fine-grained control to selectively fault victim operations, while maintaining host stability.

The interposer is connected to an external control machine, which talks to the Teensy microcontroller. Throughout the experiments and case studies, we manually enable and disable the interposer, requiring no fine-grained synchronization with the target.

Host Configuration. We disable memory scrubbing in the BIOS, as these scrubbers can cause spurious memory reads while the interposer is active, which may crash the system. On AMD, we ensure private memory regions are located on the first socket by setting the BIOS option to "Consolidate to 1st DRAM pair."

In contrast to previous work, which requires downclocking to the lowest DDR5 frequency bin (*i.e.*, 3200 MT/s) [11], our interposer is designed to operate reliably at default DDR5 frequencies. We do not modify the memory speed and let the BIOS select the highest supported frequency for the configuration.

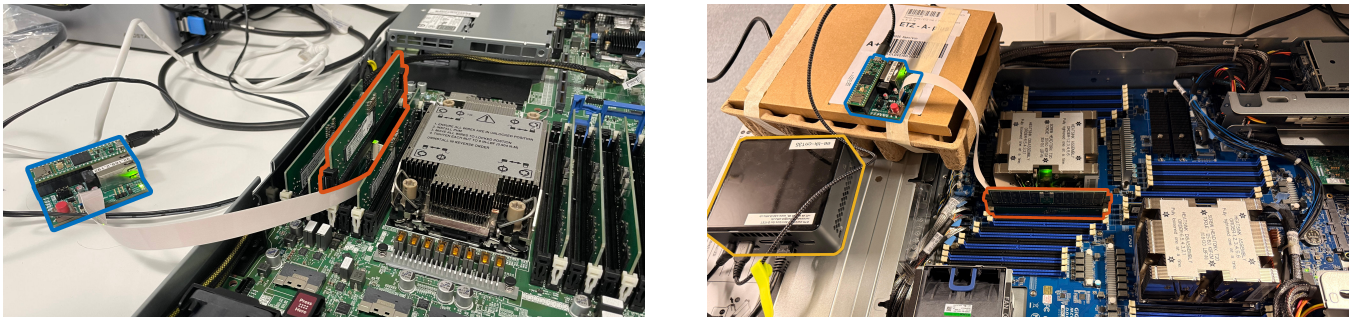
4.2 Interposer Evaluation

We now evaluate the interposer described in Section 3.3 on Intel_1. To observe the behavior of the interposer, we create a rudimentary proof-of-concept that interacts directly with physical memory, using BadRAM's `read_alias` kernel module to obtain direct, uncached access to host memory [14].

To verify write suppression, we first initialize a target memory page with a known value (e.g., `0xaa`). We then activate the interposer and overwrite the entire page with a different value (e.g., `0xbb`). Finally, we deactivate the interposer and read back the page. If the memory operations executed normally, we expect the page to contain entirely the last value we wrote (*i.e.*, `0xbb`). Any bytes still containing the original value indicate successfully dropped writes.

Table 3: Overview of evaluation systems used in this paper. The interposer is installed on one of the underlined DIMMs.

CPU	TEE	API	Mainboard	CPU	Microcode	DIMMs	Memory Speed
$Intel_1$	–	–	ASRockRack SPC741D8UD-2T/X550	1×Xeon Silver 4410Y	`0x2b000603`	6×M321R2GA3BB6-CQKET 2×KSM48R40BS8KMM	4000 MT/s
$Intel_2$	Scalable SGX TDX	– 1.5.16	HPE ProLiant DL360 Gen11	2×Xeon Silver 4509Y	`0x2b000661`	12×MTC10F1084S1RC48BA1 4×HMCG78MEBRA115N	4400 MT/s
AMD_1	SEV-SNP	1.58.4	Gigabyte MZK3-LM0	2×EPYC 9555	`0xb00215a`	1×HMCG94AGBRA181N 1×HMCG94AGBRA179N	4800 MT/s



(a) $Intel_2$. **(b) AMD_1.**

Figure 6: Custom DDR5 RDIMM interposer setup, showing the interposer, microcontroller, and control machine.

Dropping Writes. We confirm that both disconnecting the CS line and inducing parity errors successfully suppress write commands. For parity checking, specifically, we pull `CA2` low, which ensures that the write command always triggers a parity failure (see Figure 1). When reading back the target memory page, we see that some cache lines retain their original `0xaa` value. However, due to memory interleaving, the interposer does not affect the entire page. Specifically, on $Intel_1$, only eight 64-byte cache lines per page are affected, since the page is interleaved across four DIMMs, each with two subchannels.

Dropping Reads. The interposer affects all commands traveling to the memory module. This includes not only write commands, but also read operations. When attempting to read from an interposed memory location while the interposer is configured to drop commands, our testbench immediately crashes. Because the RCD ignores the read command, the DRAM chips will not drive the data bus. Consequently, the CPU will sample a floating memory bus, leading to invalid data and eventually causing a fatal error. Therefore, to maintain system stability during the attack, the adversary must ensure no read operations are performed on the affected memory regions while the interposer is active.

Swapping Ranks. Swapping the CS lines requires dual-rank memory modules. As a result, we change our $Intel_1$ configuration to include one dual-rank DIMM (MTC20F2085S1RC48BA1) installed on the interposer alongside one single-rank module (KSM48R40BS8KMM).

To test memory redirection, we now write the cache line's physical address to the target page for each cache line, rather than a fixed pattern, while the interposer is enabled. After restoring the original CS routing, we iterate over the page and read the addresses back. We confirm that swapping the CS lines successfully redirects memory writes to alternate locations on our testbench. The redirected writes land exactly 128 bytes offset from their intended location.

We also evaluate this primitive on AMD_1, as it is also equipped with dual-rank memory modules. However, we find that swapping the CS lines on that system causes ECC errors, resulting in every command sent to the DIMM being dropped. The exact cause of these errors remains unclear, as swapping the ranks should not affect either the transmitted command or its parity bits.

Discussion. We demonstrated that all three primitives proposed in Section 3.2 can enable an adversary to interfere with DDR5 commands. During our testing, we observed a recoverable ECC error for every dropped write command. However, these did not affect system stability and, as will be shown in the next sections, will not inhibit an adversary from targeting CVMs. We observed no system crashes while dropping writes to the isolated second socket, and the effects of the interposer were entirely deterministic.

Of these mechanisms, dropping writes is the most practical, applying to all DDR5 modules. Swapping ranks, on the other hand, requires dual-rank DIMMs. While this rank-swap primitive can be used to redirect addresses, in practice, ranks are interleaved at a fine-grained level, resulting in both the original and modified addresses belonging to the same page (e.g., 128 bytes apart on $Intel_1$).

Swapping the ranks does, therefore, not allow an adversary to alias with arbitrary victim pages, like prior work [13, 14], but requires specific victim gadgets to exploit. Furthermore, while the interposer is active, the original memory location becomes inaccessible, further complicating the attack. The rank-swap primitive also does not generally succeed across all our systems: we observed that AMD_1 drops commands when the CS lines are swapped. Hence, the remainder of this paper will focus on dropping writes, which we call DDRop. In particular, our interposer drops writes by inducing parity errors through switching CA2 to ground.

5 Breaking Scalable Memory Encryption

We now investigate the impact of DDRop on Intel TDX, AMD SEV-SNP, and Intel Scalable SGX. While these technologies offer additional memory protections for protected workloads via memory encryption, the underlying memory operations remain identical to those of regular host memory, leaving them vulnerable to our interposer. Additionally, the lack of ciphertext freshness means the Memory Encryption Engine (MEE) cannot cryptographically detect dropped memory commands.

5.1 Dropping Writes

We follow the same approach described in Section 4.2. However, instead of running the payload bare-metal on the host, we now run it inside a CVM running on Intel TDX or AMD SEV-SNP. For this, we provision a standard, unmodified Ubuntu 24.04 LTS enlightened guest image as the victim CVM. For Scalable SGX, we create a rudimentary proof-of-concept enclave that follows the same steps. As these three TEEs do not provide cryptographic freshness guarantees, we expect DDRop to affect them similarly.

In contrast to the previous evaluation, where we could access the interposer memory locations directly, the guest has no knowledge or direct control over its memory layout in physical memory. Instead, it relies on the hypervisor to allocate an HPA for each Guest Physical Address (GPA). Since we consider the hypervisor untrusted, we assume it has direct control over the victim's allocation. For instance, they can use the page relocation API, exposed by the TDX module and AMD SP, to relocate target GPAs to interposed memory regions. For our proof-of-concept implementation, we assume these GPAs are known to the hypervisor. While learning these GPAs can present a notable hurdle in practice—typically requiring adversaries to monitor page faults to discover relevant addresses [19, 36]—our API attacks presented later bypass this requirement entirely, exploiting DDRop without any knowledge of the victim's GPAs. For Scalable SGX, we statically allocate the target buffer to affected EPC pages using the custom SGX driver introduced by Battering RAM [13].

Evaluation. We confirm that a workload running inside a CVM or enclave is equally affected by DDRop on Intel TDX, AMD SEV-SNP, and Intel Scalable SGX. After reading back the affected memory buffer, the proof-of-concept workload sees stale data. Each experiment was repeated several times with deterministic results, and no crashes or system instability. Again, the interposer does not affect every cache line inside the targeted page due to memory interleaving.

5.2 Exploring the Trusted API

While dropping writes originating from victim CVMs forms a powerful primitive, it heavily relies on specific exploitable gadgets within the CVM and requires precise control over the victim's execution flow. To construct a more practical and universal threat, we focus on the trusted entities that enforce the security boundaries of the CVMs: the Intel TDX module and the AMD Secure Processor (SP).

These entities are responsible for setting up encrypted memory, enforcing integrity, and acting as proxies for hypervisor-driven memory management. Because they provide a standardized API to the untrusted host for provisioning and managing CVMs, exploiting these firmware components yields a more generic attack vector that can compromise any CVM running on the host system.

In this section, we show that physically dropping writes during trusted API execution allows an adversary to arbitrarily copy plaintext between victim pages and inject malicious SEPT entries.

API Access. We use TDXplore [49] to access the API exposed by the TDX module. It provides a custom kernel module that exposes the TDX module SEAMCALLs to userspace through a Python interface. For AMD, we patch AMD's `snp-host-latest` Linux kernel to support `SNP_PAGE_MOVE`. In contrast to TDX, where the TDX module manages guest page tables, this responsibility lies with the host on SEV. As the SP only moves the GPA contents between HPAs during page relocation, we patch our (host) kernel to also perform the necessary page table manipulations to reflect the moved pages.

5.2.1 Targeting Page Relocation (TDX, SEV). While the host is responsible for physical memory management, it cannot directly access protected memory. To support memory management functionality such as memory defragmentation, both TDX and SEV expose APIs—`TDH.MEM.PAGE.RELOCATE` and `SNP_PAGE_MOVE`, respectively—allowing the host to request the relocation of a GPA to a new HPA. The TDX module or AMD SP performs the actual decryption and re-encryption, ensuring the host never observes the victim plaintext.

Crucially, neither the TDX module nor the AMD SP clears the source location. The host is expected to zero out the source page before using the freed HPA [21]. In the case of TDX, this is to avoid integrity violations, which are only checked on reads, and can be reset by writing to the location.

Dropping Page Relocation. We exploit the lack of page-clearing to achieve an intra-VM plaintext-copy primitive. This primitive relies on two key observations. Firstly, the absence of memory clearing allows an adversary to initialize any HPA with valid, encrypted victim contents by relocating a victim page to the HPA and back. Secondly, the adversary can block the writes of the relocate API by relocating a guest page to a location affected by the interposer. By first initializing this target location as described in the first point, the adversary can effectively copy plaintext between two victim GPAs: When the victim accesses the GPA for which the writes were dropped, it will see the plaintext contents of the GPA with which the page was initialized. Figure 7 provides an overview of this primitive.

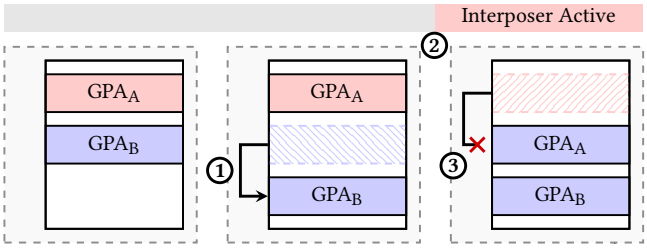


Figure 7: Overview of the page-relocation primitive. The malicious hypervisor first ① relocates a victim page (GPA_B) to a different HPA, leaving stale data at its original HPA. Next, the attacker ② enables the interposer and ③ relocates the target victim page (GPA_A) to that same HPA. Because the interposer drops the memory writes, GPA_A inherits the stale contents left behind by the initial relocation.

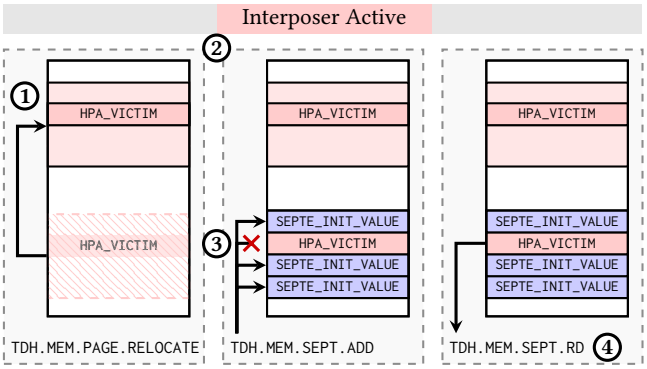


Figure 8: SEPT injection primitive. ① A page initialized as a valid SEPT entry in an attacker TD is relocated, leaving stale data at the target HPA. ② With the interposer enabled, a new SEPT page is added whose ③ dropped cache lines inherit the stale values. ④ After disabling the interposer, reading the entry returns the malicious value, giving full control over the GPA-to-HPA translation.

Evaluation. We evaluate this primitive by running a small proof-of-concept program inside the CVM that allocates two page-aligned memory buffers and initializes them to different values. We execute the primitive on the hypervisor side as described above, first initializing a target HPA with the contents of the first buffer, and then relocating and dropping the contents of the second buffer. After performing the attack, the victim's second buffer will now contain the contents of the first.

We find that this primitive is entirely deterministic in copying the victim's plaintext contents. Not all cache lines of the second buffer, however, are affected: only those mapping to the affected subchannel will be dropped (*i.e.*, one out of 16 subchannels on $Intel_2$ and one out of two subchannels on AMD_1). On $Intel_2$, we verify that in each page, four cache lines now contain the plaintext values of the first buffer. Similarly, we successfully executed this primitive on AMD_1, copying half of the buffer's cache lines.

Discussion. The root cause that enables this primitive is the firmware's reliance on the untrusted host to clear the source locations after operations like relocation or page removal. Clearing the page after such events would prevent the injection of plaintext contents, but would not preclude attacks from dropping memory writes during the relocation event. Alternatively, SEV provides an optional flag to disable the relocation API [44], though TDX lacks such a feature.

Furthermore, it is the deterministic memory encryption employed by both architectures that enables the adversary to reuse the stale memory contents. As the writes originate from the same domain, the stale contents remain undetected by TDX's logical integrity protection. While not supported by our system, some TDX deployments additionally offer an optional cryptographic integrity mode that stores a MAC along with the data. However, as the target HPA contains valid ciphertext encrypted under the correct memory encryption key, it should have a valid MAC, leaving it undetected by the integrity check. Due to the lack of support for cryptographic integrity on our system, we only experimentally verified the page-relocation primitive under logical integrity.

Beyond copying data within the same VM, an adversary could capture a victim page's state by relocating it twice and "replay" it later by dropping the writes during a subsequent relocation, rolling back the memory state without detection. Additionally, under logical integrity, the same technique can be used to copy ciphertext between a victim and an attacker's CVM. While both CVMs will be encrypted with different memory encryption keys, the encryption is deterministic, allowing an adversary to learn when the victim's plaintext changed [31, 32]. Furthermore, if the attacker can control certain plaintext pages within the CVM, e.g., via network packets [44], they can leverage this to modify any victim pages, including code pages.

5.2.2 Corrupting the Secure EPT (TDX). Traditionally, the hypervisor manages GPA-to-HPA translations via the Extended Page Tables (EPTs), which has historically enabled memory remapping [35–37] and controlled-channel attacks [53]. To mitigate this, TDX shifts the translation management to the trusted TDX module via the Secure Extended Page Table (SEPT). The SEPT resides in host memory but is encrypted under the respective TD's key. When a host allocates a page to a guest TD, it must first call the TDX `TDH.MEM.SEPT.ADD` API to initialize a new SEPT page at a host-provided HPA. The TDX module initializes this new page by writing empty entries (`SEPTE_INIT_VALUE`). Afterward, when adding the GPA to the TD, it writes the corresponding SEPT entry to that page.

Injecting Malicious SEPT Entries. By blocking the DRAM commands during the `TDH.MEM.SEPT.ADD` operation, an adversary can drop the TDX module's initialization writes, forcing the SEPT to inherit any stale ciphertext at the target HPA. Because the SEPT is encrypted with the respective TD's key, the host cannot forge an entry directly. However, they can use the same idea as before to initialize a target HPA with valid ciphertext, allowing them to craft malicious entries for their own TD. Figure 8 provides a schematic overview of the attack. Note that, as before, memory interleaving causes only a limited number of cache lines from the new SEPT page to be dropped. However, because a SEPT Entry (SEPTE) is

Table 4: Overview of DDRop primitives and TDX case studies across TEE architectures and configurations.

		DDRop Primitives			TDX Case Studies		
TEE	Crypto	Write Dropping (§5.1)	Relocate Dropping (§5.2.1)	SEPT Injection (§5.2.2)	Ciphertext Access (§6.1)	Debug Mode (§6.2)	Quote Spoofing (§6.3)
Intel TDX	AES-XTS						
Logical Integrity		●	●	●	●	●	●
*Cryptographic Integrity**		●	●	●	○	○	●
AMD SEV-SNP	AES-XEX						
Default		●	●	○	–	–	–
*PAGE_SWAP_DISABLE**		●	○	○	–	–	–
Scalable SGX†	AES-XTS	●	–	–	–	–	–
Client SGX*	AES-CTR	○	–	–	–	–	–
Arm CCA‡	AES-XEX/ QARMA	?	–	–	–	–	–
NVIDIA CC*	?	○	–	–	–	–	–

● affected ○ not affected ? not documented/implementation-defined – not applicable.
*Inferred from documentation; all other entries experimentally demonstrated.
†Demonstrated in Logical Integrity; Cryptographic Integrity also theoretically affected.
‡No commercial hardware available.

smaller than a cache line, dropping a single cache line is already sufficient to inject malicious entries.

Evaluation. We experimentally verified this primitive on Intel_2. After first crafting a malicious entry in an attacker-controlled TD and then initializing the target HPA using the relocation API, suppressing the writes during SEPT page initialization successfully injects the malicious entry into the TD's SEPT. We confirmed that the entry was successfully injected using `TDH.MEM.SEPT.RD`, which returns the compromised GPA-to-HPA translation.

Discussion. The ability to inject arbitrary SEPT entries grants the adversary arbitrary plaintext read and write access to any page encrypted under the attacker's TD key. This includes the regular guest memory, but also critical control structures such as the TD Control Structure (TDCS), TD Virtual Processor State (TDVPS), and other SEPT pages. Furthermore, on systems not protected by the cryptographic integrity mode, the attacker gains arbitrary ciphertext read and write access to any protected memory page, including those belonging to other TDs and their control structures.

5.3 Other TEE Designs

Beyond Intel TDX, Scalable SGX, and AMD SEV-SNP, other TEEs may be equally vulnerable to the proposed interposer attacks. Table 4 overviews the applicability of DDRop primitives across different TEE designs.

Client SGX. Intel's first-generation SGX featured a strong MEE, ensuring cryptographic confidentiality, integrity, and freshness [16]. Freshness is achieved by assigning counters to each cache line, and building an integrity tree across them, with the root stored in trusted, on-die storage. On every memory write, the counter is incremented, and the tree is updated. As a result, dropping the updated cache line and/or its associated counter will be detected as the integrity tree will no longer be valid. As the root of this tree is stored in on-die storage, it cannot be dropped, ensuring that any tampering with DRAM is detected. Consequently, the robust MEE design by Client SGX completely protects against DDRop by design.

Arm CCA. CCA is Arm's implementation for supporting confidential cloud computing [6]. Similar to TDX and SEV, it features memory encryption to protect against limited hardware adversaries. To this end, CCA relies on AES-XEX or QARMA [5]. While the minimal implementation requires only confidentiality for Realm data, critical monitor data stored in external memory must be integrity-protected. However, implementations can optionally offer additional protections, such as cryptographic integrity or freshness.

As there is no commercially available silicon yet that supports CCA, we are unable to verify whether it would be vulnerable to DDRop.

NVIDIA Confidential Computing. NVIDIA offers confidential computing for GPUs starting with their Hopper architecture, enabling secure AI inference in the cloud [39]. Unlike general-purpose TEEs, such as Intel TDX or AMD SEV, which use RDIMMs, these GPUs use High Bandwidth Memory (HBM). Unlike DRAM, HBM is integrated directly into the package using 2.5D or 3D stacking [24]. As a result, the command and data buses are internal to the package and thus inaccessible to a physical adversary.

6 Case Study on Intel TDX

We now demonstrate how the previously established primitives can completely compromise a TD, capture victim ciphertext, turn any TD into debug mode, and forge arbitrary attestation reports.

Setup. To avoid having to rely on the interposer for every HPA we wish to access, which slows down the attack, we instead inject a malicious SEPTE into the attacker's SEPT, which points to another attacker SEPT page. This allows the attacker TD to modify its own SEPT mappings directly, and thus access any HPA. We use this configuration as a starting point for the following attacks.

As our evaluation platform lacks BIOS support for cryptographic integrity mode, we performed all experiments in this section using logical integrity mode.

6.1 Accessing Victim Ciphertext

The maliciously injected SEPTEs allow an adversary to map an attacker-controlled GPA onto a victim-owned HPA, creating an alias between the two GPAs. This grants the attacker direct access to arbitrary victim ciphertexts. While the victim pages are encrypted under a different key, TME-MK does not provide any ciphertext freshness. As a result, this aliasing enables ciphertext side-channel attacks [31, 32], memory corruption [7], or memory replay attacks. To demonstrate this capability, we successfully reproduce the basic BadRAM memory replay attack [14] against a victim TD. By capturing and later replaying victim ciphertext, we effectively roll back the victim's memory state, forcing it to execute on stale data.

6.2 Maliciously Toggling Debug Mode

The ability to manipulate ciphertext is not restricted to standard guest pages; it also extends to critical TD control structures such as the TDCS. The TDCS stores TD metadata, including runtime attributes, attestation measurements, and migration keys. In particular, the debug status of a TD is defined by a single bit within

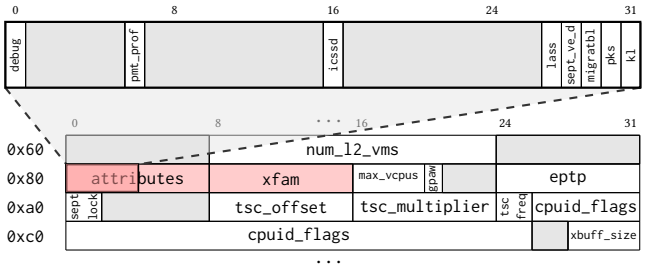


Figure 9: Detail of the TDCS memory layout, showing the 16 bytes that are affected when corrupting the attributes.

the TDCS attributes. Figure 9 provides details of the TDCS memory layout around the attributes.

While the adversary cannot directly modify the debug status bit, as the TDCS page is encrypted under the victim's key, they can write garbled data to this memory location, as the victim will decrypt the ciphertext with a different key. The plaintext read after decryption will be pseudo-random, resulting in a 50 % probability of flipping the debug flag. However, because the AES block size is 16 bytes, the attacker cannot randomize a single bit; instead, they always corrupt a 16-byte block. When targeting the TDCS attributes at offset `0x80`, the attacker randomizes the attributes as well as the Extended Features Available Mask (XFAM), which identifies the TD's set of available extended user and system features. Crucially, among these garbled bytes, the TDX module only accesses the debug bit when calling `TDH.MEM.RD/WR`, thus avoiding any crashes due to corrupted data. After accessing the plaintext data through the debug API, the original TDCS content can be restored by replaying the captured original ciphertext. To prevent other APIs from accessing garbled data, we suspend the victim TD during the attack and resume it once the original TDCS is restored.

We experimentally verify this technique on Intel_2. We write an incrementing counter from the attacker TD to the victim's TDCS attributes. Afterward, we use `TDH.MEM.RD` to test whether the bit was flipped successfully; if not, we re-randomize the attributes by incrementing the counter. We confirm this enables an attacker to access arbitrary victim plaintext through the `TDH.MEM.RD/WR` APIs. After restoring the TDCS by replaying the original TDCS ciphertext, the victim remains unaware of the compromise. We confirm this by querying the TD's attestation report, which remains unchanged after carrying out the attack.

6.3 Spoofing TD Quotes

Finally, we show how an adversary can use the malicious SEPTEs to arbitrarily modify its own attestation report by writing to its own TDCS structure. Remote tenants rely on cryptographic attestation to verify that the untrusted hypervisor has correctly provisioned their TD. During initialization, the TDX module computes a SHA-384 hash of the TD's initial memory contents and configuration. This hash value, the Build-Time Measurement Register of TD (MRTD), is stored inside the TD's TDCS. When the guest requests a quote via `TDG.MR.REPORT`, the TDX module signs a report containing this MRTD, which allows the guest to verify its initial state. If a malicious hypervisor can launch a backdoored TD and spoof the MRTD to match the guest's expected value, the remote tenant will incorrectly assume the TD was launched correctly.

By leveraging the malicious SEPTEs, as described in Section 5.2.2, an attacker can gain direct plaintext access to their own TD's TDCS. As this structure is encrypted under the same key as the attacker's guest pages, this yields arbitrary plaintext read and write access into the structure. This allows an adversary to overwrite their own MRTD field with the expected launch measurement of the legitimate victim workload.

We experimentally verify this capability on our Intel platform. After overwriting the MRTD at offset `0x240` in the TDCS, and clearing the `last_teeinfo_hash_valid` flag at offset `0x3F2`, which forces the TDX module to recompute `TEEINFOHASH`, we confirmed that the quote returned by `TDG.MR.REPORT` now contains the maliciously injected launch measurement.

6.4 Discussion

Stability and Timing. We ran each case study multiple times and observed no guest or host crashes, achieving a 100 % success rate. Our attacks do not require fine-grained synchronization; for our evaluation, we relied on an interactive Python script that pauses to manually toggle the interposer before and after invoking the target API call. Due to the manual coordination, the end-to-end execution of each case study takes up to two minutes, though an automated setup would be significantly faster.

Cryptographic Integrity Mode. TDX optionally supports cryptographic integrity mode, which stores a cryptographic MAC for each cache line. This MAC is computed over the ciphertext and encryption tweak. As these depend on the HPA and Host Key Identifier (HKID), it protects the cache line from snooping or tampering by the hypervisor or another TD. Consequently, the first two case studies—accessing victim ciphertext and toggling debug mode—would be prevented by cryptographic integrity mode, as they involve accessing or modifying ciphertext belonging to a different TD (and therefore a different HKID).

In contrast, spoofing TD quotes overwrites the TDCS from the same TD. As a result, the forged TDCS contents will be written with the correct key, yielding a valid ciphertext and MAC, leaving the tampering undetected by the cryptographic integrity checks. However, as our BIOS does not support cryptographic integrity mode, we were unable to verify this experimentally.

Table 4 summarizes the case studies for both logical and cryptographic integrity mode.

7 Discussion

7.1 Mitigations

Interposer Detection. First, we hypothesized that the additional delay on the memory bus might be detectable in memory timings: DDR5 systems commonly perform a memory training phase at boot time to calibrate the mainboard, memory controller, and memory modules to determine optimal parameters—such as timing, drive strength, and termination—to ensure signal integrity. Unless the memory arrangement changes, the timing parameters

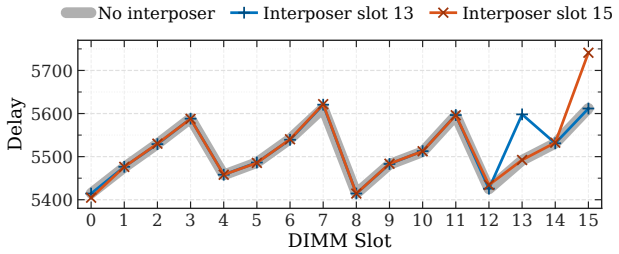


Figure 10: Delay variation across DIMM slots without and with interposer (unspecified units as reported by the BIOS).

should be broadly consistent across boots, even if there are small variations due to environmental factors.

Training sends a known pattern through a variable delay line on each signal line, sampling different delay values to determine the "valid window" in which the system can reliably communicate with the memory. The memory controller then operates at the center of this window. Training results are normally cached for subsequent boots, but a system might enforce re-training and comparison on every boot to detect the installation of an interposer.

The interposer is expected to introduce a small delay to all signals passing through it. With typical PCB delays on the order of 6 ps/mm, the 22.5 mm tall interposer and the additional 5 mm of the DIMM socket give a theoretical one-way delay of approximately 165 ps. The analog switches used to manipulate the signals have a typical propagation delay of 100 ps, but only affect the control signals, not the data lines, which are the majority of the training process.

To explore the feasibility of using memory training timings for detection, we noticed that on our Intel_2 platform, we can capture the Memory Reference Code (MRC) debug log by setting the appropriate message level in the BIOS service menu and redirecting the serial console to the Virtual Serial Port (VSP). We disabled "Memory Fast Training" so that every boot performs full training; the resulting debug messages contain detailed timing and signal-integrity measurements. We used these debug logs to capture the memory training parameters with and without the interposer.

Averaging the `RcvenDelayRank` parameter, which contains multiple measurements across the bit-lanes, for each DIMM slot yields a measure proportional to that slot's average PCB trace length. Figure 10 shows the average value across each DIMM slot. The thick gray line shows the baseline, with four sets of increasing delays. These correspond with our mainboard layout, where the DIMM slots are arranged in four groups, with each successive slot in a group being further from the CPU socket. When installing the interposer, we observed an increase in the measured delay for the slot containing the interposer, indicated by the blue and red lines.

Hence, as hypothesized, we conclude that installing an interposer can, in principle, be detected by observing changes in memory timing. Similar to existing boot-time alias detection checks, Intel could decide to use such measurements to disable TDX. However, deploying this in practice would require extensive validation across mainboards, memory controllers, DDR5 modules, and UEFI implementations to ensure reliable operation. Furthermore, malicious memory modules may integrate the analog switches directly onto the DIMM itself, optimized to reduce observable delays.

Runtime Memory Verification. An alternative, more pragmatic approach is to routinely check at runtime that every memory module retains written values. The memory controller could systematically write a fixed pattern to a small reserved region in each DIMM, and compare the value read back, thus detecting dropped writes. A similar mechanism already exists for ECC errors: memory scrubbers iterate over all cache lines at regular intervals, detecting and correcting any errors they encounter. During our experiments, we noticed that these scrubbers can cause system crashes when the interposer is active, as they attempt to read from the affected module. However, since these scrubbers can be disabled in the BIOS, they do not currently prevent the exploitation of DDRop. Furthermore, several platforms even require scrubbing to be disabled to enable Scalable SGX [30, 48]. Like timing-based detection, this approach is heuristic and may be bypassed by sophisticated interposers that drop writes more selectively.

Software Hardening. As discussed in Section 5.2, the memory-management APIs are a double-edged sword: they let the hypervisor manage memory dynamically, but also expose a large attack surface and generic primitives that an adversary can exploit.

A straightforward defense-in-depth, as presented in Google's security assessment of TDX [49], would be to support disabling these APIs when they are not needed, and attest to the user that an environment with reduced features is in use. For example, SEV-SNP supports the `PAGE_SWAP_DISABLE` policy to disable guest page swapping in response to recent attacks [4]. However, not all APIs can be disabled. For instance, the vulnerable SEPT management API is required for core TDX operations.

An alternative strategy would be to ensure that, when critical operations such as initialization or page relocation are performed, the destination page is read back and verified. While this may be easy for `TDH.MEM.SEPT.ADD`, which involves small data blocks with a default value, it may impose non-negligible performance overhead for `TDH.MEM.PAGE.RELOCATE`, as it would require reading and comparing the entire page.

We also observed that the critical data structures that hold the TD's security and runtime state are highly vulnerable to physical attacks such as DDRop. One potential mitigation would be to use higher-tier crypto for memory managed by TDX, such as the TDCS, TDVPS, and TD Root (TDR). Because a system can run only a limited number of CVMs, such critical data structures do not require scalable memory encryption. While this is not viable on today's hardware, a compromise could be to use a separate key or encryption tweak for these memory regions, making it harder for an adversary to initialize SEPT entries with malicious input from the guest or to easily flip the debug bit of a TD.

Finally, the TDX module could be hardened to avoid single points of failure, such as the debug bit. Our work is not the first to exploit the debug bit to compromise a TD; Swidowski et al. [49] showed that a software vulnerability could also flip this bit. Since this debug feature is not intended to be modified after a TD is initialized, the TDX module could add additional integrity protections around it to prevent compromise.

While the discussed software-level mitigations raise the bar for exploitation, these approaches may limit functionality or introduce performance overhead. Furthermore, they ultimately cannot address the underlying hardware vulnerability and may still be susceptible to more sophisticated interposers.

Strong Memory Encryption. While the previously discussed mitigations can limit the exposure to DDRop attacks, they ultimately cannot completely eliminate the root cause. Instead, comprehensive mitigations require both cryptographic integrity and freshness protections, allowing the CPU to detect when it is reading stale data. However, these protections place restrictions on the memory encryption engine; both integrity and freshness require additional storage, cutting into the available space, and require checks for every memory read, introducing performance overhead. While Intel's Client SGX did offer these protections, they came with a restricted protected memory size of at most 256 MB [16].

While TDX optionally provides cryptographic integrity protection by storing a MAC in the ECC bits, it is limited to 28 bits. Recently, Intel has stated that it is exploring updated memory-encryption designs for its next-generation platforms [25]. To improve resilience against physical memory bus tampering, Intel is considering moving away from AES-based encryption toward the lightweight authenticated cipher Ascon and supporting optional freshness via "cache line versioning." It is unclear whether this updated memory encryption design would fully mitigate the write-suppression primitive introduced by DDRop. Similar academic designs have leveraged the ECC bits to store counters or nonces alongside a MAC [41]; while this can provide integrity and randomization, it lacks the freshness checks required to defend against DDRop.

7.2 Limitations and Future Work

Generalizability to Single-Socket Machines. Our evaluation on Intel TDX and AMD SEV-SNP relies on dual-socket machines to provide logically separated memory domains. This ensures the interposer can drop victim writes without inadvertently faulting critical host operations. On single-socket systems where memory interleaving cannot be disabled, an attacker would require precise timing control over the injected faults to avoid corrupting unrelated memory commands. Such precise control is beyond our interposer's capabilities, as we prioritized a low-cost design using analog switches that cannot toggle their output at native DDR5 clock speeds. However, our evaluation on Intel_1 demonstrates that single-socket machines do not inherently preclude DDRop attacks.

Selective Command Interference. Our custom interposer utilizes low-cost analog switches to pin CA lines. As a result, it affects all transactions flowing over the bus and makes no distinction between read and write commands. Because dropped reads result in fatal system crashes (cf. Section 4.2), DDRop is currently limited to victim operations that only perform writes to the target interposed memory location. A more sophisticated interposer, e.g., equipped with an active FPGA controller, could selectively fault operations based on command type or target address. However, this would significantly increase the complexity and cost of the attack.

8 Conclusion

In this paper, we presented DDRop, the first active-interposer attack on DDR5 RDIMMs operating at native speeds without requiring downclocking. We demonstrated that even a cheap interposer with the most limiting ability to simply drop memory transactions is sufficient to bypass the integrity protections of modern Confidential VMs like Intel TDX and AMD SEV-SNP. By weaponizing parity errors on the command bus to silently drop writes, we developed powerful primitives targeting the trusted APIs of these architectures. Specifically, we showed how to perform deterministic plaintext copying between victim pages by exploiting the page relocation API and how to inject malicious SEPT entries by dropping initialization writes. These primitives enabled us to completely compromise TDX platforms by maliciously toggling a victim TD into debug mode and forging attestation reports.

Our results underscore a critical gap in current memory encryption engines that prioritize scalability over freshness checks. The reliance on untrusted host management APIs and the lack of differentiated protection for critical metadata structures create significant attack surfaces. The only deployed memory encryption engine that can fundamentally withstand these attacks is the original Intel SGX design, which relied on expensive Merkle hash trees to protect memory integrity, but is not scalable to the confidential VM use case. We expect our findings to enable further research into scalable or multi-tier memory encryption engines across the CPU and DRAM, as well as software and firmware hardening to protect critical metadata from command-level manipulation.

Acknowledgments

This work was partially supported by the Internal Funds KU Leuven and by 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). This work was also partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/X03738X/1 and EP/X03738X/2. Martin Thompson gratefully acknowledges the support of his employer, ZF Automotive UK Ltd, and the scholarships provided by the Universities of Birmingham and Durham. Jesse De Meulemeester is funded by an FWO fellowship (11PFE24N). This paper was edited for grammar using Grammarly and Google Gemini.

References

[1] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. 2023. *Intel trust domain extensions (TDX) security review.* Technical Report. Google.
[2] AMD. 2020. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More.* Technical Report. AMD.
[3] AMD. 2024. Undermining Integrity Features of SEV-SNP with Memory Aliasing. AMD-SB-3015.
[4] AMD. 2025. SEV Ciphertext Side Channel Attacks. AMD-SB-3021.
[5] Arm. 2021. *Arm CCA Security Model.* Technical Report Arm DEN 0096, v1.0.
[6] Arm. 2023. *Learn the architecture - Introducing Arm Confidential Compute Architecture.* Technical Report Arm DEN 0125, v3.0.
[7] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. 2017. Fault Attacks on Encrypted General Purpose Compute Platforms. In *CODASPY*.
[8] Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *CCS*.
[9] Zitai Chen and David Oswald. 2023. PMFault: Faulting and Bricking Server CPUs through Management Interfaces Or: A Modern Example of Halt and Catch Fire.

CHES 2023, 2 (2023).
[10] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David F. Oswald, and Flavio D. Garcia. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *USENIX Security*.
[11] Jalen Chuang, Alex Seto, Nicolas Berrios, Stephan van Schaik, Christina Garman, and Daniel Genkin. 2026. TEE.fail: Breaking Trusted Execution Environments via DDR5 Memory Bus Interposition. In *S&P*.
[12] Lucian Cojocar, Kevin Loughlin, Stefan Saroiu, Baris Kasikci, and Alec Wolman. 2021. *mFIT: A bump-in-the-wire tool for plug-and-play analysis of rowhammer susceptibility factors.* Technical Report MSR-TR-2021-25. Microsoft Research.
[13] Jesse De Meulemeester, David Oswald, Ingrid Verbauwhede, and Jo Van Bulck. 2026. Battering RAM: Low-Cost Interposer Attacks on Confidential Computing via Dynamic Memory Aliasing. In *S&P*.
[14] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. 2025. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In *S&P*.
[15] Stefan Gloor, Patrick Jattke, and Kaveh Razavi. 2025. REFault: A Fault Injection Platform for Rowhammer Research on DDR5 Memory. In *uASC*.
[16] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive* (2016).
[17] Jacqueline Henes, Matthew Bowden, Mihai Ordean, and David Oswald. 2026. DisARMed: Attacking ARM TrustZone from Userspace with Memory Aliasing. In *20th USENIX WOOT Conference on Offensive Technologies (WOOT)*.
[18] Bradley D. Hopkins, John Shield, and Chris North. 2016. Redirecting DRAM memory pages: Examining the threat of system memory Hardware Trojans. In *HOST*.
[19] Tristan Hornetz, Hosein Yavarzadeh, Albert Cheu, Adria Gascon, Lukas Gerlach, Daniel Moghimi, Phillipp Schoppmann, Michael Schwarz, and Ruiyi Zhang. 2026. TDXRay: Microarchitectural Side-Channel Analysis of Intel TDX for Real-World Workloads. In *S&P*.
[20] Intel. 2023. Intel Trust Domain Extensions. White Paper.
[21] Intel. 2024. *Intel Trust Domain Extensions (Intel TDX) Module Base Architecture Specification.* Specification 348549-004US. Intel.
[22] Intel. 2026. *Intel Architecture Memory Protections for Confidential Computing.* Technical Report 869103-0110. Intel.
[23] JEDEC. 2020. *DDR5 SDRAM.* Standard JESD79-5.
[24] JEDEC. 2025. *High Bandwidth Memory (HBM4) DRAM.* Standard JESD270-4.
[25] Simon Johnson. 2026. Memory Interposer Attacks: Out of scope, but not out of mind. https://youtu.be/FqHekMFxmkk. OC3 Keynote.
[26] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. 2021. *Supporting Intel SGX on Multi-Socket Platforms.* White Paper, ID 843058. Intel.
[27] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD Memory Encryption. White Paper.
[28] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. 2020. V0LTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security*.
[29] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-che Tsai, and Raluca Ada Popa. 2020. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *USENIX Security*.
[30] Lenovo. 2025. *ThinkSystem Server with Intel Xeon SP (4th, 5th Gen) UEFI Manual.* Manual. Lenovo.
[31] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *S&P*.
[32] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security*.
[33] Antonis Louka, Jesse De Meulemeester, Steven Keuchel, Ingrid Verbauwhede, and Jo Van Bulck. 2026. Physical Memory Please: Practical Memory-Aliasing Attacks on RISC-V PMP. In *uASC*.
[34] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *HASP*.
[35] Mathias Morbitzer, Manuel Huber, and Julian Horsch. 2019. Extracting Secrets from Encrypted Virtual Machines. In *CODASPY*.
[36] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *EuroSec*.
[37] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. 2021. SEVerity: Code Injection Attacks against Encrypted Virtual Machines. In *S&P Workshops*.
[38] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*.
[39] Rob Nertney. 2023. *Confidential Compute on NVIDIA Hopper H100.* Technical Report WP-11459-001. NVIDIA.
[40] Anna Pätschke, Jan Wichelmann, and Thomas Eisenbarth. 2025. Zebrafix: Mitigating Memory-Centric Side-Channel Leakage via Interleaving. In *RAID*.
[41] Moritz Peters, Jens Alich, Ashwin Jha, Gregor Leander, Yuval Yarom, and Tim Güneysu. 2026. Counters and Nonces for Mitigating Ciphertext Side Channels. *Cryptology ePrint Archive* (2026).
[42] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2019. VoltJockey: Breaking SGX by software-controlled voltage-induced hardware faults. In *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE.
[43] Benedict Schlüter and Shweta Shinde. 2025. RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP. In *CCS*.
[44] Benedict Schlüter, Christoph Wech, and Shweta Shinde. 2025. Heracles: Chosen Plaintext Attack on AMD SEV-SNP. In *CCS*.
[45] Benedict Schlüter, Christoph Wech, and Shweta Shinde. 2026. Fabricked: Misconfiguring Infinity Fabric to Break AMD SEV-SNP. In *USENIX Security*.
[46] Alex Seto, Oytun Kuday Duran, Samy Amer, Jalen Chuang, Stephan van Schaik, Daniel Genkin, and Christina Garman. 2025. WireTap: Breaking Server SGX via DRAM Bus Interposition. In *CCS*.
[47] Muyan Shen, Hongzhan Ma, Ketong Shang, Ruofei Qu, Yu Qin, and Dengguo Feng. 2026. Jailbreaking the AMD Secure Processor: Enabling Live Analysis of SEV-SNP's Undocumented Security Boundaries. In *USENIX Security*.
[48] Supermicro. 2021. *Instructions on How to Enable Intel SGX Support on Supermicro X12DP Series/X12SP Series Motherboards.* Manual. Supermicro.
[49] Kirk Swidowski, Daniel Moghimi, Josh Eads, Erdem Aktas, and Jia Ma. 2026. Security Assessment of Intel TDX with support for Live Migration. *arXiv:2602.11434* (2026).
[50] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX*.
[51] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. 2023. Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software. In *USENIX Security*.
[52] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. 2024. SEV-Step: A Single-Stepping Framework for AMD-SEV. *CHES* 2024, 1 (2024).
[53] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*.
[54] Yuqin Yan, Wei Huang, Ilya Grishchenko, Gururaj Saileshwar, Aastha Mehta, and David Lie. 2025. Relocate-Vote: Using Sparsity Information to Exploit Ciphertext Side-Channels. In *USENIX Security*.
[55] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd M. Austin. 2017. Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors. In *HPCA*.
[56] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. 2025. CipherSteal: Stealing Input Data from TEE-Shielded Neural Networks with Ciphertext Side Channels. In *S&P*.
[57] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security*.

A Open Science

To support full reproducibility of our findings and foster future research into physical memory security, we provide an open-source artifact at https://github.com/ddropattack/ddrop, including the interposer hardware designs, PCB layouts, microcontroller firmware, and code to reproduce all attacks and case studies.

B Ethical Considerations

All experiments were conducted exclusively on private lab equipment owned by the authors, and no real-world user data was compromised. While this work carries certain risks, as it could be misused by malicious parties, the benefits of coordinated public disclosure outweigh the risks. The vulnerability is generic and stems from the lack of cryptographic freshness in commercial scalable memory encryption engines. Documenting these weaknesses is essential to understand the fundamental limitations and required attacker capabilities, and to support arguments for improved memory-encryption designs. Intel's recent plans to explore improved memory-encryption designs for its next-generation cloud computing platforms [25] underscore this.

We shared our hardware designs, attack primitives, and proof-of-concept exploits with the Product Security Incident Response Team teams of both Intel and AMD. Initial disclosure to Intel occurred on

	CK	$\overline{CS}$	CA 0	1	2	3	4	5	6
ACT	L	L	L	L	Row 0–3				BA0
	H	L	BA1	BG0–2			CID0–1		CID2/ DDPID
	L	H	Row 4–10						
	H	H	Row 11–16						CID3/ R17
WRP	L	L	H	L	L	H	L	H	BA0
	H	L	BA1	BG0–2			CID0–1		CID2/ DDPID
	L	H	V	Column 3–8					
	H	H	C9–10		V	$\overline{AP}$	H	V	CID3
MRW	L	L	H	L	H	L	L	MRA0–1	
	H	L	MRA2–7						V/ DDPID
	L	H	OP0–6						
	H	H	OP7	V	V	CW	V	V	V
MRR	L	L	H	L	H	L	H	MRA0–1	
	H	L	MRA2–7						V/ DDPID
	L	H	L	L	V	V	V	V	V
	H	H	V	V	V	CW	V	V	V
WR	L	L	H	L	H	H	L	$\overline{BL}$	BA0
	H	L	BA1	BG0–2			CID0–1		CID2/ DDPID
	L	H	V	Column 3–8					
	H	H	C9–10		V	$\overline{AP}$	$\overline{WP}$	V	CID3

	CK	$\overline{CS}$	CA 0	1	2	3	4	5	6
RD	L	L	H	L	H	H	H	$\overline{BL}$	BA0
	H	L	BA1	BG0–2			CID0–1		CID2/ DDPID
	L	H	Column 2–8						
	H	H	C9–10		V	$\overline{AP}$	V	V	CID3
VrefCA	L	L	H	H	L	L	L	OP0–1	
	H	L	OP2–6					L	V/ DDPID
VrefCS	L	L	H	H	L	L	L	OP0–1	
	H	L	OP2–6					H	V/ DDPID
REFab	L	L	H	H	L	L	H	CID3	V
	H	L	V	V/ RIR	H	L	CID0–1		CID2/ DDPID
RFMab	L	L	H	H	L	L	H	CID3	V
	H	L	V	V	L	L	CID0–1		CID2/ DDPID
REFsb	L	L	H	H	L	L	H	CID3	BA0
	H	L	BA1	V/ RIR	H	H	CID0–1		CID2/ DDPID
RFMsb	L	L	H	H	L	L	H	CID3	BA0
	H	L	BA1	V	L	H	CID0–1		CID2/ DDPID
PREab	L	L	H	H	L	H	L	CID3	V
	H	L	V	V	V	L	CID0–1		CID2/ DDPID

	CK	$\overline{CS}$	CA 0	1	2	3	4	5	6
PREsb	L	L	H	H	L	H	L	CID3	BA0
	H	L	BA1	V	V	H	CID0–1		CID2/ DDPID
PREpb	L	L	H	H	L	H	H	CID3	BA0
	H	L	BA1	BG0–2			CID0–1		CID2/ DDPID
SRE	L	L	H	H	H	L	H	V	V
	H	L	V	V	H	L	V	V	V
SREF	L	L	H	H	H	L	H	V	V
	H	L	V	V	L	L	V	V	V
PDE	L	L	H	H	H	L	H	V	V
	H	L	V	V	V	H	$\overline{ODT}$	V	V
MPC	L	L	H	H	H	H	L	OP0–1	
	H	L	OP2–7						V
NOP	L	L	H	H	H	H	H	V	V
	H	L	V	V	V	V	V	V	V
PDX	L	L	H	H	H	H	H	V	V
	H	L	V	V	V	V	V	V	V
DES	L	H	X	X	X	X	X	X	X
	H	H	X	X	X	X	X	X	X

Figure 11: Full command encoding for DDR5 RDIMMs [23]. Each line can be high (H), low (L), valid (V), or don't care (X).

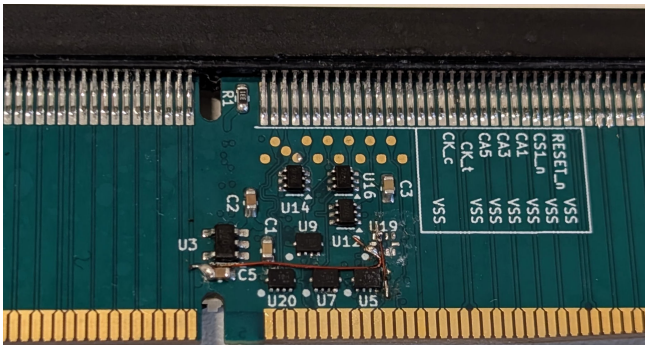


Figure 12: Interposer modifications required to support CS disconnection.

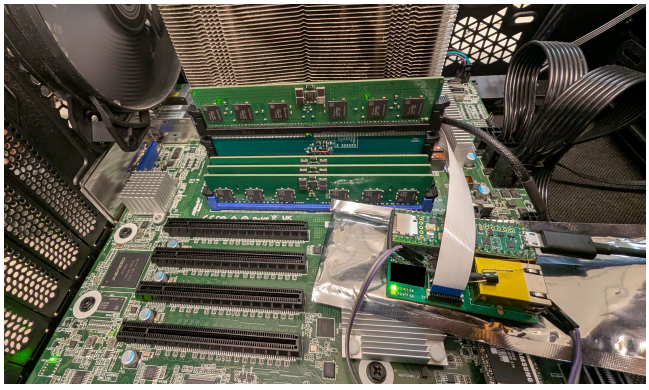


Figure 13: Modified interposer installed on Intel$_1$.

November 4, 2025, with additional primitives and exploits shared on February 19 and April 8, 2026. We disclosed our findings to AMD on January 28, 2026. AMD requested an embargo on our results, and we aligned on a coordinated disclosure date of September 14, 2026, with Intel agreeing to the same timeline. We also shared our results with Arm on April 29, 2026, which, though we did not evaluate DDRop on Arm CCA, may be equally affected. Both Intel and AMD acknowledged our findings and released security bulletins on the coordinated disclosure date.

C Interposer Modifications for CS Disconnecting

Our interposer is configured by default to pin the CA lines and to swap the chip-select lines (cf. Section 3). To support the rank disconnect primitive (cf. Section 3.2), minor modifications are required to repurpose the existing rank-swapping switches. In particular, the input to the switch toggling the DIMM-side $\overline{CS_0}$ line, which originally connects to the CPU-side $\overline{CS_1}$ (cf. Figure 4), must be re-wired to 1.1 V. The other input remains the CPU-side $\overline{CS_0}$. This ensures that, when the switch is toggled, the DIMM will not see the CS line asserted. A similar approach can be applied to $\overline{CS_1}$; however, we decide to bypass its switch and connect it directly to the CPU side. Note that, even on a single-rank DIMM, both CS lines are required during memory training, meaning $\overline{CS_1}$ cannot be left floating.

Figures 12 and 13 show the modified PCB and its installation on Intel$_1$.

D DDR5 RDIMM Command Encoding

Figure 11 provides the full command encoding for DDR5 RDIMMs.