Counters and Nonces for Mitigating Ciphertext Side Channels

Moritz Peters[†], Jens Alich, Ashwin Jha, Gregor Leander, Yuval Yarom and Tim Güneysu[‡]

Ruhr University Bochum, Bochum, Germany, firstname.lastname@rub.de
[†]moritz.peters-v41@rub.de, [‡]tim.gueneysu@rub.de

Abstract. Recent works have highlighted the risk of deterministic memory encryption, as used for example in AMD SEV. In the so-called ciphertext side-channel attack, an adversary leaks sensitive information, such as cryptographic keys, from an encrypted VM by observing changes in the encrypted contents of the memory. To date, countermeasures have been quite restricted in scope and have not addressed the root cause. Alternatively, some studies suggest adding freshness to memory encryption. However, this approach is considered impractical due to the performance overhead introduced by fetching such freshness values for every single memory access.
In this work, we propose an efficient approach for eliminating most of the overhead of fetching the aforementioned freshness. Our core idea is to repurpose the ECC memory area to efficiently store random nonces or counters. We propose a range of implementations with varying trade-offs between security guarantees and performance overhead, and demonstrate that we can achieve a solid baseline security even with small random nonces. By leaving a portion of ECC memory unused, we show that it is possible to efficiently integrate protection mechanisms such as memory integrity and memory tagging, while limiting the overall performance overhead to approximately 2%.

Keywords: Ciphertext Side Channel · Memory Encryption · Countermeasure · AMD SEV-SNP · Intel SGX/TDX

1 Introduction

Today, workloads are increasingly moving from single isolated systems to large-scale cloud computing environments, which enable better scalability by sharing physical hardware across workloads. The key technology is Virtual Machines (VMs), which allow isolating different workloads from each other while running on the same physical hardware. While different VMs usually cannot interfere with each other, the hypervisor, which manages the VMs, is not that restricted. Having privileged access to the system allows the hypervisor to read out and potentially modify all data a VM is using.

To further isolate the hypervisor from the VMs, Trusted Execution Environments (TEEs) can be used. In a nutshell, they view the hypervisor as a potentially malicious third-party. By augmenting VMs with memory confidentiality and integrity, they hinder data extractions and other means of influencing a VM. All major processor vendors either have already released or are actively working on integrating TEEs into their server CPUs. Most prominently, those are AMD Secure Encrypted Virtualization (SEV) [KPW16], Intel Trusted Domain Extension (TDX) [Int22a], and ARM Confidential Compute Architecture (CCA) [ARM25].

Despite the isolation guarantees offered by TEEs, a fundamental issue remains: the underlying hardware is shared, and VMs continue to use common resources such as caches and other microarchitectural structures. Multiple works on cache timing side channels showed how to exploit the shared nature of a cache to gather information like cryptographic keys from isolated processes or VMs [DKPT17; GMWM16; HRY24; OST06; PTV21; TOS10; YF14]. Another attack vector are Rowhammer attacks, which allow an attacker to flip bits in adjacent rows in the Dynamic Random Access Memory (DRAM) [FVH+20; GJR25; GMM16; JNH+24; KJQ+22; SD15; VDVFL+16]. If the affected row contains data read by a VM, it could cause it to change its control flow or crash.

In 2021, Li et al. [LZW+21] presented a first side-channel attack on AMD SEV called CIPHERLEAKS, leaking information through encrypted register states in the VM Save Area (VMSA). A follow-up work [LWW+22] further formalized this type of side channel in 2022. The attack observes that TEEs use deterministic memory encryption, where the same plaintext always produces the same ciphertext. By observing changes in the ciphertext, an attacker can gain information about the inner state of a VM or algorithm running inside a VM, possibly revealing private keys or other secrets.

All this happens under the condition that the ciphertext is readable by the hypervisor. Intel Software Guard Extension (SGX), as well as the newer Intel TDX [Int22a], prohibit read access to the memory of encrypted VMs. Thus, attacks from software like [LZW+21] are not possible on these architectures. AMD already deployed a microcode patch that mitigates CIPHERLEAKS by introducing nonces into the encryption of register states in the VMSA [AMD21]. Because these nonces only apply to the VMSA, the rest of the VM memory is still exposed to ciphertext side-channel attacks [LWW+22]. Starting with Zen 5, AMD introduced Ciphertext Hiding [AMD25] to mitigate ciphertext side-channel attacks. It prevents the hypervisor from reading the ciphertext of protected guest VMs. Instead of reading the ciphertext, the hypervisor will see fixed values like 0xff. While it prevents such attacks from software, more sophisticated attacks that directly target the memory bus [CSB+26; LJF+20; MWO+25; SDA+25] are not mitigated. Chuang et al. [CSB+26] show that it is possible to extract cryptographic keys from Intel TDX and AMD SEV-SNP with Ciphertext Hiding for the first time. This raises the need for countermeasures that solve the issue closer to its root: the deterministic memory encryption.

Wichelmann et al. [WPWE23] propose a software-based countermeasure for ciphertext attacks. Their approach tracks secret data to detect critical memory accesses, which they then mask with random values to avoid repeating the plaintext and thus the ciphertext, making the encryption non-deterministic. While the approach is flexible and is easy to roll out, it fails to address the vulnerable hardware and suffers from a substantial performance overhead.

Our Contribution

In this work, we propose hardware-based approaches for securing encrypted memory against ciphertext side channel attacks. Our core idea is to introduce freshness into the encryption to reduce the probability of the same plaintext producing the same ciphertext. This mitigates attacks from the hypervisor via software as well as directly via the memory bus, but raises the question of where to store the values introducing the freshness without affecting performance too much. We propose re-purposing some of the error correction bits of the additional Error Correction Code (ECC) memory.

The main challenge of the approach is that the number of ECC bits is limited: 64 ECC bits for each block of 64 memory bytes in DDR4 and 128 bits in DDR5. Moreover, ECC bits are currently used for ensuring memory integrity, and there are proposals for also using them for memory tagging [LUSM24]. Synergy [SNR+18] uses the additional ECC space to store parity bits and MACs protecting an integrity tree and data. However, they still store the full tree in the main memory, requiring additional memory accesses per request. Thus, we have to carefully manage the available ECC bits to close existing

ciphertext side channels while maintaining memory integrity protection.

We present and evaluate a range of implementation options that offer various trade-offs between security guarantees, performance overhead, and chip area costs. Our basic implementation starts by storing random 30-bit nonces in the ECC bits in DDR4 memory, increasing attack complexity by a factor of 2^{15}, or more than four orders of magnitude. In DDR5, the nonce can be expanded to 60 bits due to the larger ECC memory, further increasing attack complexity to 2^{30}. We achieve this with a negligible hardware overhead and a small performance overhead of less than 2%. Moreover, the approach can be easily combined with prior techniques for ensuring memory integrity and tagging [JLK+23; LUSM24] by using the remaining 34 or 68 ECC bits in DDR4 and DDR5, respectively.

Our second implementation replaces the random nonces with counters. This increases the number of modifications before ciphertexts repeat to 2^{30} in DDR4 and 2^{60} in DDR5, but re-introduces determinism, which may reduce security in some cases. The performance of this implementation is similar to that of the basic implementation, but counters need to be cached, increasing the area of the LLC by about 5% or 11% for DDR4 and DDR5, respectively.

To overcome the security issues with small counters, we propose adding frame-level counters, stored in a reserved region in the DRAM. With 32-bit counters, we achieve a 62-bit security at the additional cost of 0.1% of the main memory. Frame-level counters also incur a larger performance overhead of around 6% on average.

We observe that, typically, only a small fraction of the physical memory frames mapped to a VM are updated frequently. Hence, for most frames, the frame-level counter is not needed. We optimize the system for the common case and use frame counters only on frames that change frequently. The approach needs one bit to track frequently-changing frames, achieving a security of 61 bits. The overall performance of this implementation is comparable to that of random nonces, but it achieves a security level of 61 bits.

2 Background

Here, we introduce relevant background to Intel and AMD trusted execution environments, ciphertext side channels, and Rowhammer.

2.1 Intel SGX and TDX

Intel Software Guard Extension (SGX) extends the x86_64 instruction set to provide a mechanism to create trusted execution environments, so-called enclaves, protecting the data and code within them against read and write access from the outside. For this, SGX uses a memory encryption engine called Total Memory Encryption (TME) [Int22b]. Directly implemented in the memory controller, TME de- and encrypts data using AES-XTS with a key generated at boot. The tweak used in XTS is based on the physical memory address of the data. SGX also allows remote attestation to verify that the code runs on a trusted Intel platform. In 2021, Intel introduced Trusted Domain Extension (TDX) [Int22a], lifting enclaves to fully-capable VMs, now called trusted domains (TDs). The memory encryption is strengthened by assigning each TD its own ephemeral key rather than using a single key for all of them. Optionally, a SHA-3 based Message Authentication Code (MAC) can be enabled to protect the integrity of the data in DRAM.

2.2 AMD SEV

Introduced in 2016, AMD Secure Encrypted Virtualization (SEV) safeguards virtual machines (VMs) from untrusted hypervisors by encrypting VM data with unique, hardware-managed AES keys [KPW16]. This process occurs transparently within the memory

controller, requiring no additional software. The system utilizes AES-128 encryption, transitioning from XE [DYM+17] to XEX [Rog04] mode in later versions. Encryption uses a tweak function T based on the physical memory address a. To address the low entropy (32-bit) in initial tweak vectors identified by Wilke et al. [WWME20], AMD implemented 128-bit vectors starting with the Zen 2 architecture [SSB20]. Since the introduction of SEV, AMD has released two extensions to further strengthen its security. With the first one, SEV-ES [Kap17], register states are now encrypted during VMEXITs to prevent attacks targeting the VMCB or control flow [HB17; WMA+19]. With the second one, SEV-SNP, the Reverse Map Table (RMP) was introduced to track page ownership, preventing the hypervisor from remapping VM pages, a vulnerability exploited in previous research [HB17; MHHW18]. Starting with Zen 5, AMD implemented Ciphertext Hiding [AMD25], which mitigates ciphertext side-channel attacks like [LZW+21] by returning default values (0xff) when the hypervisor attempts to read encrypted VM memory.

2.3 Ciphertext Side Channel

Li et al. [LZW+21] observe that monitoring the ciphertext for changes allows the hypervisor to detect changes in the plaintext. Combining the information with knowledge of the behavior of the code can leak information about the data being processed. This underlying cause of the issue is that the memory encryption employed by SEV is deterministic, always encrypting the same plaintext in a memory address to the same ciphertext. By now observing the ciphertext holding the register states of a VM in the encrypted VMSA during VMEXITS, they were able to attack previously secure cryptographic constant-time implementations. While the attack vector in [LZW+21] has been fixed by AMD [AMD21], the root problem still exists, allowing the attacks that target other memory locations [LWW+22]. Specifically, Li et al. [LWW+22] identify two core primitives that all ciphertext side-channel attacks have in common. The first, *dictionary* attack, relies on a pre-built dictionary of known plaintext-ciphertext pairs. Such a dictionary can be obtained by recording ciphertext for known writes to the same memory address. In the second, known as *collision* attack, the attacker only observes changes in the ciphertext without the need to recover plaintext, for example, when targeting a constant-time swap operation.

Originally, CIPHERLEAKS only affected AMD SEV because the hypervisor on Intel TDX cannot read encrypted memory, thus losing one crucial attack primitive. With AMD rolling out the ciphertext hiding feature starting with Zen 5, this primitive is also no longer available on SEV when enabled. There is, however, one issue with read restrictions either in TDX or with AMD's ciphertext hiding. Those only prohibit the hypervisor, and thus software running on the Central Processing Unit (CPU) from read the encrypted memory Chuang et al. [CSB+26] demonstrate that it is possible to observe memory traffic directly on the bus, bypassing any read restrictions, and thus (re-)enabling ciphertext attacks on fully patched Intel and AMD systems.

2.4 Rowhammer

In 2013, Kim et al. [KDK+14] demonstrated that an adversary can exploit the cell density in DDR3 DRAM and make unrelated cells interfere with each other. More specifically, repeatedly opening and closing a row causes increased leakage in nearby cells, causing a discharge that is faster than the refresh rate, therefore flipping bits. Later works show that the same effect is still present in DDR4 and DDR5 technology [GJR25; JNH+24; KJQ+22]. This, in turn, sparked various attacks. For example, Seaborn and Dullien [SD15] demonstrated how *hammering* two rows surrounding a victim can increase the probability of a bit flip. They then employed this technique to flip bits within the page table, leading to unprotected access to large areas of memory, including the kernel. Gruss et al. [GMM16]

showed that the Rowhammer attacker can be mounted on a user's hardware using malicious JavaScript. Van Der Veen et al. [VDVFL+16] ported Rowhammer to mobile devices with ARM CPUs and were able to flip bits in the page table to gain privileged access.

Multiple mitigations for Rowhammer have been developed in academia and the industry. Initially, ECC was seen as a possible mitigation. However, it has been shown that induced bitflips may not be detected by ECC if carefully crafted [CRGB19]. Starting with DDR4, manufacturers included a hardware mitigation called *Target Row Refresh*, which monitors row accesses. Once a certain threshold is reached, additional *refresh* commands are issued to protect neighboring cells. TRR-aware Rowhammer variants exist, which employ access patterns that do not trigger the TRR [FVH+20]. Juffinger et al. [JLK+23] propose a solution that removes the ECC memory and replaces it with a 56-bit MAC and 8 parity bits. This allows them to not only detect errors and their location within the 512-bit memory block, but also enables a software-based approach to correct errors by flipping bits in the corrupted memory block until the MAC matches again.

3 Overview

In this section, we present our approach for a memory encryption scheme secure against ciphertext side channels. We first define our attacker model. Then, we introduce our design goals as the general idea to mitigate ciphertext side channels.

Attacker Model. As in prior work [WPWE23], we consider an attacker who tries to leak secret information from a TEE. The attacker has full control over the system that runs the TEE and can read the entire encrypted memory. However, they cannot decrypt or modify it. Unlike previous works, we do not limit the attacker to only read the ciphertext via software, i.e., via the hypervisor, but also directly from the memory bus [CSB+26]. Lastly, the attacker has complete knowledge of the code executed by the victim and can trace and interrupt its control flow.

Design Goals. Our goal is to secure memory encryption deployed in AMD SEV-SNP and Intel TDX against ciphertext side channels in the presence of a passive listener on the memory bus, with lightweight changes to the on-chip memory encryption. Although software solutions are more flexible and easier to design, their implementation is far from trivial and requires a high level of expertise. While there are recent works to automatically secure software [WPWE23], we believe that this problem is best addressed at its root. Our design should introduce non-determinism in the memory encryption in hardware so that the same plaintext never results in the same ciphertext under a fixed memory address. Furthermore, the design should be composable with past approaches for protecting memory, such as integrity protection and memory tagging.

Idea. Currently, SEV and TDX employ AES in XEX/XTS mode, using the physical memory address as a tweak to encrypt data written to the main memory. Writing the same data multiple times to the same memory address always yields the same ciphertext. Thus, changing the tweak every time data is written to memory yields a different ciphertext every time, effectively mitigating ciphertext side channels. Introducing additional freshness into the tweak, e.g., a random value or a counter, can force such a change. The important part is that the additional freshness does not repeat or at least does so very rarely, such that attacks become infeasible.

The main challenge is that the freshness needs to be stored somewhere alongside the ciphertext. Intel TDX's cryptographic integrity mode and other previous works used the extra memory on ECC DRAM to store additional information for stronger memory integrity protection [JLK+23] or even memory tagging [LUSM24]. This seems like a

practical approach since this extra memory can be loaded in parallel to its associated normal data, eliminating latency overheads in theory. However, using this additional ECC memory simply for our freshness leads to a different issue: using all available extra memory for freshness, we lose all integrity protection. Since memory integrity protection is also crucial to overall system security, losing it completely to gain something else is undesired. Instead, this work explores ways to share the ECC memory for storing both the freshness used for memory encryption and integrity data needed for error correction and detection. This leads to security as well as performance tradeoffs, which we explore in this paper.

4 Freshness in Memory Encryption

In this section, we introduce two types of freshness that can be integrated into the memory encryption engine and stored in the ECC memory: fully random nonces and counters. We discuss the advantages as well as security drawbacks of both types, especially when dealing with small values. For this initial discussion, we abstract over implementation details to better highlight the general properties of the approaches. After this first look, we take a deeper look at the options to integrate such freshness into the memory ecosystem, exploring both existing DDR4 and DDR5 memory technologies.

4.1 Random Nonces

A random nonce is a uniformly distributed random n-bit value to be used as part of a tweak input to a Tweakable Block Cipher (TBC) or as input to a counter-mode cipher. The main advantage of random nonces is that encrypting some plaintext does not depend on any previous values and can thus be sampled independently. This has two implications. First, choosing a new nonce is inherently non-deterministic. Knowing the current nonce does not improve the guessing probability of subsequent nonces beyond 2^{-n}. Second, an existing nonce can be overwritten without needing to know its previous value. This reduces the overall number of read accesses needed to the nonce, which is an important aspect of our implementation.

In contrast, their main disadvantage is their limited security. To understand why, one needs to look at nonce collisions and why they leak information. Imagine two 128-bit plaintexts PT_a, PT_b being encrypted with the same key and two 30-bit nonces T_a, T_b. When $T_a \neq T_b$, the ciphertexts will be different with high probability regardless of whether PT_a and PT_b are equal or not. There is a small chance that two different plaintexts will yield the same ciphertext under different nonces, but this is negligible in reality, as most of the time the plaintext space is limited. The interesting case is $T_a = T_b$. For two different plaintexts, the corresponding ciphertexts will still be different, but for two identical plaintexts, the ciphertexts will also be identical. Thus, an attacker who observes two different ciphertexts cannot determine whether the corresponding plaintexts are different, because they do not know whether the plaintexts or the nonces differ. However, if the attacker observes two identical ciphertexts, they can determine with a high probability that the plaintexts are also identical. This leads to the conclusion that a nonce collision has the potential to leak information about the plaintexts, forming the core problem of collision attacks presented in [LWW+22].

Collisions of random nonces can be modeled with the help of the birthday paradox. For uniformly random n-bit values, we are likely to start observing collisions after $\sqrt{2^n}$ sampled values.

4.2 Counters

Counters are incremented by 1 on every update. Some plaintext is encrypted using a counter value of x. The next plaintext is then encrypted with $x + 1$, and so on. This introduces two implications. First, compared to nonces, a counter changes predictably. If the current counter value is known, the next one can be computed easily. Second, the old counter value needs to be known before an encryption, because it needs to be incremented. This stands in contrast to nonces, which can be chosen independently of the old value. Hence, writing to memory requires additional memory accesses to determine the previous counter value before updating it.

The advantage of counters over nonces is that their security is not bound by the birthday paradox. Collisions occur after 2^n writes instead of $\sqrt{2^n}$ for an n-bit counter. However, a collision occurs on every memory write with 100% certainty at this point.

5 Implementation

ECC DRAM memory modules feature an additional chip on the DIMM to store additional data for the purpose of error detection and correction. Most DDR4 memory modules come without ECC capabilities, whereas recent DDR5 modules include ECC more frequently. Due to the increasing memory densities, bit-flips happen more often in DDR5, making error correction and detection necessary. In ECC-DDR4 memory, 64 bits are stored on the additional memory chip per 512-bit block of memory. To reduce latency, the data bus with a width of 72 bits is wider than typical DDR4 memory without ECC capabilities. Thus, data and the corresponding ECC bits can be loaded in parallel, eliminating the need for separate loads. ECC-DDR5 doubled the available ECC bits to 128 bits per 512-bit block of memory or 64 bits per 256-bit block. Future generations of DRAM might include even more bits for ECC as data densities and errors increase. Thus, we also consider an imaginary architecture with ECC 256 bits per block. In this paper, we will refer to the additional memory for ECC bits as *ECC memory*.

Being able to load additional data without additional overhead makes ECC memory an attractive place to store associated data, like the freshness values used for encryption. Nonces can be read and written directly from and to the ECC memory since they can be updated independently of their previous value. Hence, no additional read or write operations are needed, and the nonce values can be modified fully in parallel to the associated normal data operations. For counters, the previous value has to be known in order to increment it. This leads to the issue that the counter needs to be fetched separately before writing data to memory. Hence, each memory write is preceded by a memory read, significantly slowing down overall memory performance. To counteract this slowdown, we can increase the cache line size to also store the counters, eliminating the need to fetch the counter when updating the contents of memory. While mitigating the performance impact, this will lead to an overhead in cache size of $\frac{n}{512}$ with n being the number of bits used for the counter. For example, using 64-bit counters would lead to 12.5% overhead in cache size.

5.1 Sharing the ECC with Memory Integrity and Tagging

Using the available ECC memory fully for freshness yields the best security against ciphertext side channels because tweak collisions become very rare when using nonces or take a long time to occur. However, we then lose the capability to detect and correct errors, the main purpose of ECC memory. This leaves us with the question of how to effectively reintroduce integrity protection to get the best of both worlds.

A Look at Related Work. To arrive at an optimal split between freshness and error correction capabilities, we analyzed two major prior works that focus on error detection and correction, also using available ECC memory. CSI:Rowhammer [JLK+23] repurposes ECC memory to store a MAC with a small part reserved for parity bits. The MAC computation is done via a PMAC construction using the QARMA cipher, which may be parallelized, as each memory block can be encrypted independently. Voodoo [LUSM24] adds memory protection via tagging and authenticated encryption, adding fine-grained memory access capabilities. The overall structure is similar to CSI:Rowhammer. The main difference is that they include memory encryption implicitly, as the message is encrypted in any case, and combine these into a final MAC using finite field arithmetic with a secret value H. In the context of AMD SEV, Voodoo could be advantageous as it only requires one encryption operation per memory block. CSI:Rowhammer requires the system to encrypt the plaintext twice. Once when it is stored in memory, and a second time when the corresponding MAC is computed. Theoretically, we can use either of these schemes as a basis for our implementation, but due to the availability of an artifact, we build on CSI:Rowhammer. We find that by changing the underlying encryption as well as carefully managing the available ECC space, we can combine memory integrity of CSI:Rowhammer with the tagging capabilities of Voodoo, while also providing ciphertext side channel protection.

Splitting the ECC Memory. Before introducing the changes made to the MAC construction, we discuss how much ECC space could be spared on different memory architectures. The simplest solution is to split the ECC memory into two halves. The lower half stores the freshness, whereas the upper half stores a MAC. For DDR4 memory, that would result in 32 bits used as freshness and 32 bits used as MAC, 64/64 bits for DDR5, and 128/128 bits for an imaginary architecture with 256-bit ECC per cache line. In the case of DDR5, CSI:Rowhammer could be applied almost as is, since the 64 bits fit their original design. For our imaginary architecture, QARMA with a larger block size or even AES could be used to calculate a 128-bit MAC. For DDR4 with only 32 bits available, we choose to use SIMON, offering a 32-bit block mode. The actual split could be changed to 25% fresheness and 75% MAC or vice versa, trading security against ciphertext side channels for stronger integrity protection. However, it is important that the number of bits used as MAC is a power of two, as we will see next when introducing memory tagging into the scheme.

Integrating Memory Tagging. While Voodoo could also be applied directly here, we found that their proposed tag encoding schemes could also be used to integrate a tag into the MAC of CSI:Rowhammer. To include a memory tag, we evaluated applicable solutions proposed by Voodoo, mainly a simple pattern, e.g., the most significant bits must be 0, or that valid tags must have a certain Hamming weight. We found that both perform equally well, but opted for the Hamming weight tags. Our tag space is confined to 8 bits. To be able to successfully encode and decode memory tags into the MAC, it needs to be decryptable by the block cipher used to create it and thus be of the same block size. This conflicts with the original construction in CSI:Rowhammer since they truncate their 64-bit MAC to fit additional parity bits. Our solution to this problem is to slightly shrink the bits available for freshness and use them to place the parity bits. Because the parity bits are directly linked to the error correction efficiency, we again have a trade-off between error correction capabilities and security against ciphertext side channels.

When integrating memory tagging, we have to take care that data errors, wrong tags, and errors in the ECC bits are correctly detected and handled. An error in the data and accesses with a wrong tag are identified the same way. We calculate $\widetilde{\text{tag}} = Dec(\widetilde{\text{MAC}}) \oplus \text{MAC}'$ where $\widetilde{\text{MAC}}$ is the MAC stored in the ECC bits and MAC' is the freshly calculated intermediate MAC before xoring the tag. If $\widetilde{\text{tag}}$ is a valid tag with respect to its encoding, a wrong tag was provided by the access. Otherwise, there is an

error in the data or the ECC bits storing the MAC. Data and ECC errors are corrected as-a-search or by approximate equality, respectively, as proposed in [JLK+23]. Errors in both the data and ECC are corrected as-a-search, trying to find an approximate MAC. This assumes a correct tag on access. Errors in data, ECC bits, and a wrong tag lead to a correction of the data that will not succeed. This will ultimately lead to relying on more advanced correction procedures by the OS, like page reloading.

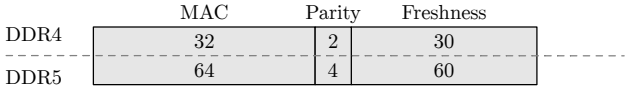


Figure 1: Possible allocation of an ECC entry in DDR4/DDR5 memory.

Wrapping Up ECC Sharing. Concluding these integration steps, Figure 1 gives an overview of a 50%/50% split on DDR4 and DDR5 architectures. In DDR4, we would reserve 30 bits for freshness, 2 bits for parity, and 32 bits for our combined MAC and tag construction in DDR4 memory. In a DDR5 with double the capacity of ECC, we can fit 60 bits of freshness, 4 bits of parity, and 64-bit MACs.

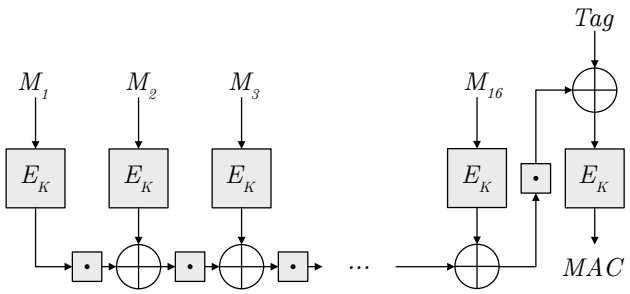


Figure 2: MAC computation in DDR4 with embedded memory tag according to [JLK+23; LUSM24]. It uses 17 instances of SIMON 32/64 in a PMAC construction and a multiplication by 2 in $GF(2^{32})/[x^{32} + x^{22} + x^2 + x^1 + 1]$ before each intermediate XOR.

Tweaks in SIMON. Using SIMON as a 32-bit block cipher for the MAC leaves one last issue. Unlike QARMA, SIMON does not specify tweaks, which were used to tie each memory block to its physical address. Thus, an adversary would be able to swap the location of memory blocks inside a 512-bit chunk without generating an erroneous MAC. In order to solve this problem, our MAC construction includes a field multiplication, which reintroduces a position dependency. In hardware, the multiplication can be realized via a shift and an XOR. A complete construction of the 32-bit MAC using SIMOIN with integrated memory tags can be seen in Figure 2.

Table 1: Security bounds of nonce and counter values when allocating all or sharing the ECC memory with a MAC. Amounts of parity bits are 2, 4, and 8, respectively.

	DDR4 (64-bit ECC)		DDR5 (128-bit ECC)		Future DDR (256-bit ECC)	
Allocation	Nonce	Counter	Nonce	Counter	Nonce	Counter
Full	32-bit	64-bit	64-bit	128-bit	128-bit	256-bit
25% MAC	23-bit	46-bit	46-bit	92-bit	92-bit	184-bit
50% MAC	15-bit	30-bit	30-bit	60-bit	60-bit	120-bit
75% MAC	7-bit	14-bit	14-bit	28-bit	28-bit	56-bit

Theoretical Security Bounds When Sharing The ECC. Finally, connecting back to Section 4, we collect the theoretical security bounds of random nonces and counters in Table 1. We differentiate between the full allocation of the ECC memory, i.e., using all available bits for freshness, and the shared allocation, where the ECC memory is split.

5.2 Counter Extensions

We now discuss the option of extending counters to further enhance security when sharing the ECC memory with bits for error detection and correction. While it would be possible to further extend the space we allocate with each cache line, it would again double the area overhead. This idea was first proposed by Yan et al. [YEP+06] in 2006. They split encryption counters into minor 8-bit counter per cache line and major 64-bit counter per 4KB page and use a small separate 32KB cache to keep recently used counters (major and minor) available without costly memory loads. When en-/decrypting a cache line, those counters are concatenated, forming a single 72-bit counter. In our work, we use a slight variation of this idea which will be outlined in the following.

Frame-level counters. Similar to [YEP+06], we extend our block counters stored in the ECC memory with 32-bit counters, shared by all cache lines inside one 4 KiB physical memory frame. At boot time, those counters are initialized to zero and only incremented when a block counter overflows inside the corresponding frame. This effectively extends the 30-bit block counters to 62-bit counters in DDR4 and from 60 to 92 bits in DDR5. However, incrementing the page-level counter requires the full memory frame to be re-encrypted. Otherwise, we would not be able to access the remaining cache lines in it, since we would not store the old counter. The only exception is the one cache line that was written when the counter was incremented. Re-encrypting the full frame is costly because it requires all remaining cache lines to be read, decrypted, encrypted, and finally written back to memory. This happens 63 times, resulting in a total of 126 DRAM bursts as well as 126 en-/decryptions. Fortunately, this happens infrequently. In the worst case, when writing a cache line at full speed, given a DRAM latency of around 200ns, it would take five minutes for the 32-bit counter to overflow and trigger a re-encryption. We can further reduce the frequency of re-encryptions by resetting all other block counters to 0 when incrementing the frame-level counter. Thus, we avoid the case where other block counters may be overflowing soon, triggering multiple re-encryptions in a shorter time frame.

Our frame-level counters are stored in a dedicated DRAM region, protected and only accessible by the memory controller, removing the necessity for additional storage on other ends. Furthermore, the footprint in the memory is as small as 0.1%, which is insignificant compared to, for example, an overhead of 5.85% when storing 30-bit counters in cache lines. The downside is that fetching the frame-level counter is slow. Without any additional measures, the memory latency for a single access (read & write) doubles, since the counter has to be fetched first. We add a small 1 KB 2-way cache to the memory controller to keep recently fetched counters available at much lower access latencies. Compared to [YEP+06], we only need to cache frame counter because block counter are stored together with their associated cache line in the CPU caches. This solves the problem for most workloads, keeping the performance overhead to a minimum. However, the performance for workloads that access a lot of different frames in quick succession (low locality) significantly drops even with the cache, as it can only hold that many counters at once.

Indicator Bit. To further reduce the overhead of fetching the frame-level counter and potential cache contention, we add a single *indicator bit* to each cache line and store it permanently in the ECC by freeing up one bit from either the counter or parity. This increases the total area overhead since one additional bit needs to be stored next to

each cache line. It does not include the additional cache for the frame-level counters when choosing to keep the counter length unchanged. Since the frame-level counters are initialized to zero and only change when a block counter overflows, most of them stay at zero for a long time. Especially when data is not written very frequently, the indicator bit tells the memory controller if a frame-level counter was incremented at some point or is still at zero. That way, we can immediately en-/decrypt without fetching the counter in the case the bit indicates that it has not changed yet. This further reduces the performance overhead, even for workloads frequently cycling through different frames. First, a block counter has to overflow for a frame-level counter to be even cached in the dedicated cache. In turn, there then need to be enough counters that overflow in different frames to, at some point, fill the dedicated cache to its maximum.

5.3 Memory Encryption Mode

For the actual encryption scheme, we went away from AES-XEX and used AES-CTR with ZIP-AES [FGL+24]. We choose the counter mode because we cannot use the raw tweak, i.e., the concatenation of address and freshness, for the cipher and have to first run it through some kind of tweak derivation function. Since the counter mode already utilizes the concatenation of address and freshness as cipher input, we obtain the tweak derivation for free. As a result, we are able to merge two steps, tweak derivation and encryption, into one.

Flórez-Gutiérrez et al. [FGL+24] propose a general method for building PRFs with low latency, based on the observation that, in a nutshell, the sum of two permutations $E_1 \oplus E_2$ is as secure as the composition $E_2 \circ E_1^{-1}$. Thus, by taking a secure block cipher and splitting it into equal parts—along with some additional analysis—one obtains a secure PRF.

A concrete instance of this framework is ZIP-AES – depicted in Figure 3 – a variant built as the sum of two 5-round AES instances. As explained in [FGL+24], this yields a secure PRF with half the latency of AES-CTR and twice the security in terms of data complexity. Thus, luckily, this well-understood primitive matches our needs perfectly. Figure 3 shows the complete memory encryption with the full tweak and the ZIP-AES construction. It would be implemented in hardware and replace the dedicated memory encryption hardware found in processors.

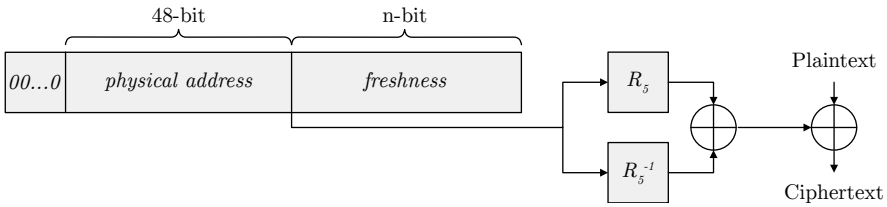


Figure 3: Counter mode encryption with ZIP-AES [FGL+24] with five rounds of AES encryption (R_5) and five rounds of decryption (R_5^{-1}).

6 Evaluation

In this section, we evaluate proposed techniques regarding security, reliability, performance, and overall area overhead.

6.1 Security against Ciphertext Side Channel

Following the introduction of our encryption scheme, both notions of freshness and MAC integration, we now evaluate their security. This includes assessing the security of counters and nonces against ciphertext side-channel attacks and the Heracles attack on AMD SEV [SWS25], the robustness of the MAC and memory tags, and the impact of active and passive DRAM bus attacks.

Our analysis is based on DDR4 and the split discussed in the previous section: 30-bit nonce/counter and 32-bit MAC with 2 parity bits. This setting has the fewest bits for freshness, providing a lower bound on security. Counters are not extended.

We start with generic dictionary and collision attacks, which form the underlying primitives for more specific ciphertext side-channel attacks. Following these, we look at two attacks on actual cryptographic implementations by Li et al. [LWW+22]. The first is a dictionary attack on constant-time ECDSA, the second is a collision attack on constant-time EdDSA. In both attacks, the authors were able to reconstruct the secret value.

Dictionary Attack. The dictionary attack requires the attacker to build a list of plaintext-ciphertext pairs. By observing ciphertexts for known plaintext, a dictionary containing those pairs can be constructed. Some examples of places to collect those are given in [LWW+22]. However, the attacker is limited by the number of possible ciphertexts. Imagine a 128-bit block of memory that contains a single 8-bit variable the attacker is interested in. The remaining bits are not used and will not change over the course of the observations. There are 2^8 possible ciphertexts for the attacker to observe. If the memory block now additionally contains a variable that changes frequently, e.g., a loop counter, the number of possible ciphertexts can quickly rise, making such an attack infeasible. By introducing randomized nonces or counters into the encryption, we increase the number of possible ciphertexts. Keeping the example with the 8-bit variable, introducing 30 bits of entropy either through a random nonce or counter increases the dictionary size from 2^8 to 2^{38}. Building such a dictionary would take a significant amount of time and space (4 TB), significantly limiting the attacker. The complexity for random nonces is bound by the coupon collectors problem, while for counters, the attacker needs to collect and observe exactly 2^{38} encryptions. Further increasing the counters, as proposed previously, would increase the complexity for the attacker. However, there are variants of dictionary attacks that do not require building a full dictionary, which we will discuss shortly.

Collision Attack. Instead of building a dictionary to directly infer the plaintext from the ciphertext, it might be sufficient to only know if a ciphertext changed after some operation or not in some cases. The prominent example is the constant-time swap operation. It always executes in constant time and does reads/writes to the same memory addresses independently of the secret. However, the actual value written to memory depends on the secret, leaking information when observing changes in the ciphertext. When introducing a random nonce or a counter, the attacker will, in almost all cases, see that the ciphertext changed after an operation. In the case that no change is observed, the plaintext did not change, and a nonce collision happened with high probability.

For a 30-bit random nonce, the probability that the same nonce is chosen again is 2^{-30}. While not impossible, it will take the attacker a substantial amount of time to leak every non-changing occurrence of the plaintext. As a simple example, imagine a function consuming a secret bit by bit. It always first writes A, then depending on the current bit, it writes A again on a 0 and B on a 1. Now, an attacker would only be able to leak all positions of zero bits. For each position, they must encounter a nonce collision on the second write with a probability of 2^{-30} each time. Additionally, the attacker cannot tell when they have discovered all zero positions in the secret. They can only say that after

some time, the probability of finding another position is so low that they end the attack. On the other side, counters fully mitigate this attack vector for consecutive writes. Even if a few writes are between the observations, the counter will never repeat in such a small number of writes if sufficiently large, which is the case for us, using at least 30 bits for the counter.

Dictionary Attack on Constant-time ECDSA. Li et al. [LWW+22] presented a dictionary attack on a constant-time ECDSA implementation in OpenSSL. They exploited that context switches to the Linux kernel cause the current register states to be written to a fixed structure `pt_regs`. In the constant-time ECDSA implementation, a 384-bit nonce k is processed, which can be used to immediately infer the used secret key if leaked. Observing the return values stored in RAX from `BN_is_bit_set`, called for each bit in k, immediately leaks the value of k. For every bit in k, they recorded all 384 ciphertexts of the RAX register stored in `pt_regs`. Since RAX takes only two distinct values, they were able to infer k with only 1 bit of entropy.

Introducing a 30-bit random nonce or counter would increase the amount of entropy to 31 bits. With a high probability, each ciphertext in the trace would be different. Thus, they would have to guess for each bit separately if it is 0 or 1 in that case. Interestingly, if two ciphertexts are identical, their plaintext must also be identical with high probability. This effectively reduces the guessing space since the attacker knows that those two bits are equal. Now, if the attacker is able to redo the experiment multiple times, they could record ciphertext collisions and identify equal bits until all bit positions are grouped to be either 0 or 1, reducing the guessing entropy down to 1 bit again. The number of times the experiment would need to be repeated depends on how fast nonce collisions are encountered. We implemented this as a series of experiments in a short Python script. Each experiment takes as input a secret string of 0s and 1s with a total length of 384 and encrypts a 0 or 1 for each of the respective positions in the string. For each encryption, we record the ciphertext and at which position of the string it occurred, and check if we have seen the ciphertext in previous experiments. If this is the case, we record the tuple of positions in the secret string where they occurred. This leaves us with a list of tuples which we can build into two transitive groups, reducing guessing entropy down to 1 bit. For 30-bit random nonces, we found that the experiment would have to be repeated around 5000–8000 times until all positions are grouped. Based on the average time for one experiment, which the authors stated to be around 5.8s, doing all 6800 experiments would take around 14h. For 30-bit counters, the experiment would have to be repeated until the counter repeats. That is $2^{30}/384 = 2796203$ times, taking a total of 275 days. However, repeating the experiment is only possible if the nonce or any other secret in similar attacks would be deterministic or fixed over all experiments. If this is not the case, the attacker can only hope to see equal ciphertexts in a single trace. This equals the probability of finding at least one colliding nonce, which is around 0.0000684%.

Collision Attack on Constant-time EdDSA. This attack presented in [LWW+22] focuses on leaking a value through a collision attack on the constant-time swap operation in a constant-time EdDSA algorithm. In this setting, a constant-time swap is performed where the actual value being swapped depends on a bit from a secret value. By comparing the ciphertexts before and after the swap, the attacker can deduce the bit that was used for that swap and reconstruct the secret value. Using a counter would completely deny this attack, since the counter would not repeat in that short time the swap executes, and thus would never leak any information. This is different for random nonces. When the nonce and the plaintext before and after the swap are identical, both ciphertexts will match. Otherwise, they will always be different with high probability. The attacker now knows that if they encounter two identical ciphertexts, the plaintexts were also identical, leaking

one bit of our secret. The probability for the nonce being identical before and after the swap is 2^{30} for a 30-bit random nonce. The expected number of ciphertext pairs to observe to find two that are identical can be modeled via the geometric distribution and is 2^{30}. The authors reported an average runtime of 7.9s to recover an 84-bit secret value. Hence, an attacker needs to observe 2^{30} ciphertext pairs on average to find one that actually matches, making this attack infeasible for a 30-bit random nonce.

Heracles. Schlüter et al. [SWS25] found that the hypervisor is able to move encrypted pages of a VM. While the page is re-encrypted after the move, this re-encryption is still deterministic. By carefully choosing the locations where pages are moved to, they were able to create a chosen plaintext oracle. With the help of such an oracle, they could leak private guest memory at block granularity.

Heracles builds on three design flaws in SEV-SNP: the ability of the hypervisor to read ciphertext, the lack of freshness in the memory encryption, and the hypervisor's ability to move pages. The authors state that by fixing one of their flaws, Heracles can be fully mitigated. Because our mitigation introduces freshness into the memory encryption, fixing the second design flaw, we also fully mitigate Heracles.

Active and Passive DRAM Bus Attacks. Chuang et al. [CSB+26] showed that being able to intercept encrypted memory transactions directly on the bus and thus bypassing any read restrictions, completely breaks the security of Intel TDX and AMD SEV-SNP, even with Ciphertext Hiding enabled. In a nutshell, they reintroduced the ciphertext side channel on those platforms, which was previously thought to be closed by restricting read access to ciphertexts from software. In our attacker model, a passive attacker can also observe every ciphertext as well as the used counter or nonce and the MAC. The only advantage the attacker gains with this information is that they definitely know when a nonce or counter repeats. While this lifts some complexity of the attacks, it does not influence the rate at which collisions occur.

An active attacker is additionally able to modify or replay data on the bus. Modifying the nonce or counter would lead to ciphertexts being decrypted into random bytes, likely crashing the system. Due to the malleability of the used counter-mode encryption, bit flips in the ciphertext will carry through to the plaintext. However, since we use a MAC to protect the integrity of the ciphertext, the security of our memory encryption is only reduced to the security of the MAC. For a successful attack, the attacker must forge a valid MAC matching the modified ciphertext. If the MAC check fails, both CSI:Rowhammer and Voodoo are able to correct bit errors up to some degree. When the correction is successful, the attacker gains nothing. How a failed correction is handled is up to the designer of the system, but it would make most sense to lock down the system or let it crash. While it would be feasible to try all 2^{32} possible MACs in a relatively short time, the fact that when a wrong MAC is guessed, either the changes in the ciphertext are corrected or the system crashes, makes such an attack infeasible. Since our random nonces and counters are not authenticated by, e.g., an authentication tree like in Intel SGX [Gue16], we do not protect against replay attacks. Such trees can significantly reduce performance because they are usually stored in the main memory. Several works have proposed optimizations or different tree structures to reduce the loss in performance [GSC+03; HS10; SCG+03; SNR+18; YEP+06].

6.2 Reliability and Serviceability

After evaluating the security of our memory encryption against ciphertext attacks, we turn our attention to memory integrity. More specifically, we evaluate how different MAC sizes affected the detection and correction of different types of faults in the data. Our analysis

is based on the error rates reported by two previous works that evaluated DDR4 [BCG+23; BCG+24] and DDR5 [BCT+25] memory in the field. They reported error rates in FIT (Faults in Time) per DRAM device, representing the fault-rate per one billion device-hours of operation. Errors rates of 72.08 and 96.44 FIT per DRAM device were observed for two different DDR4 memory vendors. Of these, only 17.16 and 22.94 FIT per DRAM Device were multi-bit faults while the rest were single-bit faults, respectively. On DDR5 memory, 9.54, 5.84 and 2.75 FIT per DRAM Device were observed for three vendors, of which 8.64, 4.82, and 2.75 were due to multi-bit errors, respectively. Single-bit errors in DDR5 are corrected by the on-die ECC engine, never reaching the DRAM controller. However, in [BCT+25] some single-bit errors were detected and the authors suspect that they must have occurred outside the control of the on-die ECC engine.

Single-Bit Errors. As proposed by Juffinger et al. [JLK+23], a MAC is used in combination with some additional parity bits. If a single flipped bit causes a second pre-image, the MAC will not detect the corruption. However, it will always be detected by the parity bits. Thus, a single bit flip can never lead to an undetected error. Furthermore, a substantial amount of single-bit errors is corrected on-die in recent DDR5 memory.

Table 2: Expected time to obtain a second preimage due to naturally occurring errors in DDR4 and DDR5 memory for an n-bit MAC.

MAC size (bits)	DDR4 [BCG+23; BCG+24]	DDR5 [BCT+25]
32	4.1×10^{20} years	3.04×10^{21} years
64	1.76×10^{30} years	1.31×10^{31} years
128	—	2.41×10^{50} years

Multi-Bit & Undetected Errors. We will now turn our attention to multi-bit errors and undetectable errors. An error is undetected if it produces a second pre-image, i.e., the MACs of the valid and corrupted data are the same, and the parity check passes. To pass the parity check, at least two bits must flip in one block. Since we assume a secure MAC, the probability of finding a second pre-image is independent of the number and location of the bitflips. To get a second pre-image, approximately 2^n different inputs need to be tested, with n being the MAC size in bits. Using DDR4/DDR5 error rates from previous works [BCG+23; BCG+24; BCT+25] scaled to a single cache line, we give an estimate on how long it would take to get an uncorrectable error by dividing the number of errors needed to find a second pre-image by the rate at which errors occur. Table 2 lists the expected times for all four possible MAC sizes (32, 64, and 128) in both DDR4 and DDR5 memory. We see that even with a small 32-bit MAC, it would take over 10^{20} years for a second pre-image and thus an uncorrectable error to occur naturally.

Rowhammer. After exploring the influence of naturally occurring errors on the MAC, we now turn our attention to actively induced errors via Rowhammer attacks. First, we look at the expected time it takes to generate a second pre-image by randomly hammering a row. We assume the same attacker optimistic scenario as [JLK+23], where two bits always flip in a block after hammering a row for two refresh intervals (128 ms). Using this new rate, we calculate the expected time for all three discussed MAC sizes the same way as for naturally occurring errors. For a 32-bit MAC, it would take the attacker around 25 days to generate a second pre-image and thus a data error. In comparison, a 64-bit or 128-bit MAC would take the attacker more than 10^8 years.

Instead of randomly hammering data, the attacker may also try to precisely induce errors to craft valid data and MAC. The argument here is the same as already discussed

by Juffinger et al. [JLK+23], with the different keys per VM further hindering the attacker from calculate a valid MAC for victim data at a specific location. In short, the attacker has two options: flip the victim data to an exploitable second pre-image without flipping the MAC or flip the victim data and the MAC. For the first approach, the attacker further has to find matching data for the aggressor row, which only changes the data and not the MAC, i.e., another second pre-image. For the second approach, the attacker has to find aggressor data that flips the victim data to be exploitable, which also has a MAC that matches the flipped victim data. Both approaches are computationally infeasible, which is even further complicated by the fact that newer DRAM requires double-sided hammering.

Error Correction. Because we use a MAC to detect errors, we have to rely on the error-as-a-search approach by Juffinger et al. [JLK+23]. However, using small MACs as well as small amount of guiding parity bits quickly becomes an issue as the correction procedure needs to calculate many different MACs, running into the risk of accidentally finding a second pre-image. To detect errors in the stored MAC, it is compared with the calculated MAC. If they differ by only some bits, this is counted as a stored MAC error, and the MAC is overwritten by the calculated one. Depending on how this threshold is set, the security of the MAC is reduced. Juffinger et al. [JLK+23] calculated the security loss for different thresholds and compared them to the number of computed MACs during correction. The calculated values for a 32-bit MAC are collected in Table 3. Using only two parity bits, a 32-bit MAC, and a threshold of two we are only able to correct up to three errors in the data. Increasing the number of parity bits to four gives the correction more confidence to not find a second preimage. Up to four errors can be corrected using an approximate equality threshold of 0. Larger MACs allow correcting larger amounts of errors, as well as allowing for higher thresholds. A 64-bit MAC with four parity bits, as used by us in DDR5, can correct up to 5 bit flips in a reasonable amount of time (2min). Larger errors are still detected but due to the significant increase in correction time, it is usually better to rely on other means of error recovery, e.g., page reloading.

Table 3: Number of MAC computations versus the strength of a 32-bit MAC. The strength depends on d which is the number of flips allowed in the MAC to be corrected by approximate equality. c is the number of different MAC computations needed to correct the data with 2 and 4 parity bits, respectively.

#BitFlips	$log_2(c)$	d	MAC str.
1	7.21 / 6.06	3	19.57
2	15.04 / 13.32	2	22.95
3	22.92 / 20.65	1	26.95
4	30.63 / 27.99	0	32

Comparison to Currently Deployed ECC Implementations. We will now briefly compare our error correction and detection capabilities with currently deployed schemes by Intel and AMD. The ECC implementation on Intel Cascade Lake servers uses SEC-DED with a code with a minimal Hamming distance of 4, which was reverse-engineered in [KWS+25]. SEC-DED with a minimal Hamming distance of 4 can correct a single error and theoretically detect up to three errors. Additionally, Intel advertised SDDC capabilities, correcting all errors from a single failing x4 DRAM device [KWS+25]. While there is no clear information on the code used by AMD, we assume that they use one with similar capabilities. Newer AMD CPUs are equipped with AMD AMDC [AMD20], which is able to correct all errors from a single failing x4 DRAM device. Furthermore, recent DDR5 memory has on-die

correction capabilities, correcting single-bit errors without them reaching the memory controller [BCT+25].

We are able to detect any number of bit flips probabilistically due to the use of a secure MAC. Error correction on DDR4 with a 32-bit MAC and two or four parity bits lacks slightly behind SDDC and AMDC capabilities of correcting four errors from a single chip. However, we are capable of correcting three errors spread over any number of chips. Setting the threshold for approximate equality to 0 allows us to also correct four errors at the cost of not being able to correct errors in the MAC. Moving to DDR5, we are able to store double the bits for MAC and parity. This shows that even with small MACs and a restricted number of parity bits, errors detection and correction capabilities can be on par with currently deployed schemes while our scheme additionally offers enhanced memory encryption by introducing freshness.

Table 4: Base configuration for the gem5 system.

Processor & CPU		**Memory & Cache**		**Sim. Environment**	
Architecture	x86-64	L1I Cache	32 kB	Mode	Full System
CPU Type	Simple Timing	L1D Cache	64 kB	Workload	Ubuntu 22.04
Cores / Freq	1 Core, 3 GHz	L2 Cache	2 MB (LLC)	Kernel	Linux 5.9.2
		DRAM	3GB DDR4_2400_8x8		
		CTR Cache	1 kB, 2 Ways		

6.3 Simulation using gem5

We validate our approaches for the memory encryption and combinations with memory integrity and tagging using the gem5 simulator [BBB+11]. We build upon the artifact provided by Juffinger et al. [JLK+23], as they already implemented memory integrity, where we only had to add our memory encryptions as well as the memory tagging as Lamster et al. [LUSM24] proposes. As there is no ECC memory model in gem5, the memory controller was already modified by Juffinger et al. [JLK+23] to simulate DDR4 ECC DIMMs. They extended the memory bus to 72 bits to accommodate the additional memory on such DIMMs. Additionally, they implemented their MAC computation and verification directly into the memory controller. We add parts of our memory encryption also in the memory controller, as well as directly into the abstract memory model. While frame-level counters and indicator bits are handled by the memory controller, the actual encryption and nonces/counters are handled by the abstract memory model in gem5. This is purely because of software implementation reasons, as everything could also have been implemented inside the memory controller, but this would have come with some complications. Either way, the outcome of the simulation is the same. All configuration parameters of the gem5 simulation, including our frame-level counter cache, can be found in Table 4.

6.4 Performance

In the following, we want to give a detailed view of the performance implications of all our tested configurations and their combinations. To summarize, these are:

1. Memory Encryption with random nonces/counters
2. Memory Encryption with additional frame-level counters (+ indicator bit)
3. Adding Memory Integrity with MAC and Tagging

For the workloads, we choose the well-known PARSEC benchmark suite [Bie11] running in full-system emulation mode. We were interested in how many ticks gem5 needs to

simulate a given workload while using the configurations above. This gives us an estimate of the overhead a tested configuration entails. The baseline we tested against had neither encryption nor MAC enabled.

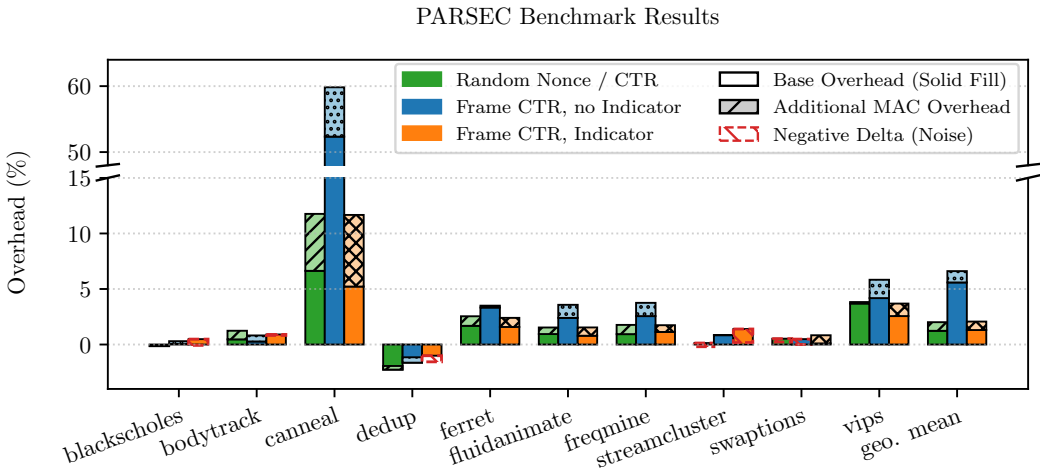


Figure 4: The overhead of the memory encryption and MAC using random nonces / block counters vs. with frame-level counters vs. with indicator bit (DDR4).

Memory Encryption. Memory encryption adds latency in two ways: the en-/decryption itself and fetching the frame-level counter, if implemented. As there is currently no full hardware implementation of ZIP-AES available, we had to estimate its latency based on the average latency of one round. Avanzi [Ava17] reported an average of 1.58 ns per round. Thus, we impose a total latency of $5 \cdot 1.58$ ns $= 7.9$ ns for each encryption. We do not impose any latency to fetch the random nonces/block counters, as those are read/written together with the actual data in parallel. The latency for fetching the frame-level counter required a more sophisticated approach and is fully modeled through gem5's memory system, sending memory requests first to the counter cache, and forwarded to the DRAM memory if they miss.

Figure 4 shows the overhead that the memory encryption entails (1) with only 30-bit random nonces/block counters, (2) with additional frame-level counters, and (3) with additional indicator bit. We observe that the overhead when only using the encryption with random nonces/block counters stays below 2% for most benchmarks. This is expected as it only adds a small static delay for every memory access. Nonces themselves do not add any latency. Memory-intensive workloads like canneal and vips report a stronger overhead with 4.75% and 3.81%, respectively. Overall, the geometric mean for this configuration is 1.06% overhead. Adding the frame-level counter adds significant overhead to those workloads frequently accessing many different memory pages and thus frames, clearly observable by canneal. It reports an overhead of nearly 50% when using frame-level counters. For other workloads, the small 1 kB cache hides a lot of the latency, but not for canneal. The cache reports a miss rate of around 80%, which strongly correlates with the high overhead. Every miss effectively doubles the latency for one memory access. The geometric mean also increases to 5.49% for this configuration with frame-counters.

The gem5 simulation does not model non-temporal writes, e.g., Direct Memory Access (DMA), bypassing the cache. These can cause additional read cycles because the counter values are unknown. However, in classic SEV-SNP for example, DMA is not allowed to write into the encrypted memory of a VM. Hence, we do not need to load a counter since

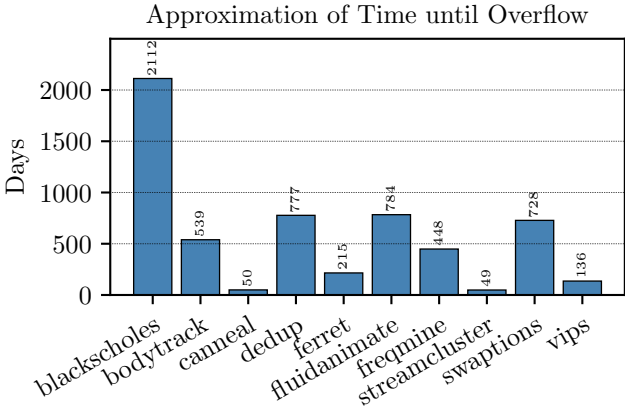


Figure 5: Calculated time the respective workload would need to run continuously for the first block counter to overflow and cause the indicator bit to flip.

writes only occur into unencrypted shared memory. In some use cases like trusted devices with AMD SEV-TIO [AMD23], writes into encrypted memory can occur and need an additional read to first load the counter. Examples are GPU-heavy workloads like machine learning and AI applications. For these workloads, using a counter would significantly slow down their execution.

Memory Integrity and Tagging. When adding memory integrity in the same way Juffinger et al. [JLK+23] did, we cannot use QARMA-64 when also adding memory tagging. Instead, we use SIMON 32/64 as a substitute, as it will allow us to encrypt/decrypt 32-bit plain-/ciphertexts required for the memory tagging to function correctly. As there are no latency values for SIMON 32/64 available for recent technologies, we used a reference hardware implementation[1], unrolled it, and synthesized it using the Synopsys Design Compiler and 45 nm Nangate OpenCell Library [Sil]. It reported a latency of 4.95 ns. The total latency of the MAC computation, as well as integrating the memory tag, is $2 \cdot 4.95$ ns $= 9.9$ ns, assuming 16 parallel and one final instance of SIMON 32/64.

Figure 4 also shows the additional performance overhead when MAC computation are enabled in addition to memory encryption. The relative distribution of the overheads is mostly identical to that without MAC computation. However, we see a global increase in overheads attributed to the added static latency on a memory access. For benchmarks with a smaller memory footprint, the increase is less noticeable, while those with a high memory footprint, namely canneal, almost double in overhead for random nonces/counters and configurations with an indicator bit. Without an indicator, the latency of the additional memory request almost fully hides the MAC computation, showing an overhead increase by only around 7%. The geometric means for all three tested configurations rise by 0.7%-1%.

6.5 Estimation of Counter Overflow Times

We want to give an intuition on when the indicator bit becomes useless, i.e., the performance degrades back to the configuration without the indicator. For each benchmark, we record the highest block counter per memory frame. Multiplied by the total runtime of the benchmarks, this gives us an indication of how long a similar workload will take to overflow a block counter and thus requires fetching the frame-level counter on each access. Figure 5 provides an overview of the time each benchmark or similar workload would need to run until the first block counter overflows. For benchmarks with small memory footprints,

[1] https://github.com/apsarras/simon-hw

values in the range from 700 to 2100 days of runtime are observed. Highly memory-intensive workloads, like canneal and streamcluster, take around 50 days for the first block counter to overflow. This clarifies why the indicator bit is so effective in reducing the overhead of frame-level counters. Until the first overflow, no frame-level counter needs to be fetched from DRAM and therefore does not incur any performance overhead. Even then, the additional cache would hide the additional latency for further accesses. There have to be enough overflows in different pages to saturate the cache and cause performance impacts for workloads accessing many different pages/frames. This would take even longer than overflowing a single block counter. At this point, a full change of the VM encryption keys could happen, which would also allow us to reset all counters. This would require the full memory to be re-encrypted once, but would afterward restore the performance.

6.6 Memory & Area Overheads

Fully random nonces add no memory overhead, as they can be completely stored in the additional ECC memory. 30-bit counters in DDR4 add an overhead of 5.85% due to the need to store the counters together with their cache lines. For DDR5, this would result in a 11.7% memory overhead. Adding frame-level counters requires reserving 0.1% of the available DRAM memory to store them, as well as a static 1 kB separate cache. For our memory encryption, we need four ZIP-AES instances in total. As there is no hardware implementation of ZIP-AES available at this point, we have to estimate the approximate size by dividing the size of a regular AES-128 by 2. To be consistent, we use the same AES used as comparison in Avanzi [Ava17], from where we already extracted the latency. They reported an area of 71164 GE for a full AES-128. This would result in an approximate area of 35582 GE for one instance of ZIP-AES. For all four instances, we arrive at 142328 GE in total. As we replace the existing encryption hardware with our implementation, the actual overhead will be less. Unfortunately, there is no area data available for the current implementation of AMD CPUs, which would allow an exact calculation of the overhead. For the MAC and memory tagging hardware, we need 17 instances of SIMON 32/64 in DDR4 and 9 instances of SIMON 64/96 in DDR5. Again, we used the same design of an unrolled implementation as for the latency. Our design compiler reported an area of 3627 GE, giving us a total of $17 \cdot 3627 = 61659$ GE in the case of DDR4 and 32643 GE in the case of DDR5.

7 Conclusion

In this paper, we looked at multiple ways to mitigate ciphertext side channels inside the hardware rather than via software. We used non-deterministic memory encryption with random nonces or counters to add enough entropy to the ciphertext to make attacks from software and directly via the memory bus infeasible. By storing counters or nonces in the additional memory of ECC memory, we eliminate further costly accesses to the DRAM to load those values. While nonces are bound by the birthday paradox, limiting their security to half of their bit size, they are more lightweight and easier to implement than the naturally more secure counters. Counters come with the downside of requiring additional storage after their initial load to avoid the need to load them a second time when incrementing and writing them back to memory. Furthermore, they slow down DMA-heavy workloads since those accesses are usually not cached, and thus neither is the counter. For such workloads, nonces are preferred, especially when ECC bits are plentiful like in DDR5 or future architectures, balancing out their naturally weaker security. Our analysis shows that a good level of security can already be reached with small random nonces in ECC-sparse environments like DDR4. Furthermore, we showed that our implementations can be combined with other security mechanisms from prior research, like memory integrity

and tagging, by sharing the space available in the ECC memory. Finally, we evaluated the performance of our implementations as well as when running in combination with memory integrity and tagging, showing that the overall overhead stays manageable for most workloads. Overall, more ECC bits in DDR5 or future architectures will not only help strengthen their original purpose of protecting memory integrity but also strengthen resistance against ciphertext side channels when used to store freshness.

Acknowledgements

We would like to thank Florian Stolz for the discussion, advice, and support.

This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, through project number 560392681, and under Priority Program SPP 2253 Nano Security through the project RAINCOAT II (440059533); an Australian Research Council (ARC) Discovery Project number DP210102670; and as an ERC project 101097056 (SYMTRUST).

References

[AMD20] AMD. Advanced memory device correction (amdc) for servers, 2020. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/advanced-memory-device-correction.pdf.

[AMD21] AMD. Amd secure encryption virtualization (SEV) information disclosure (bulletin ID: AMD-SB-1013), 2021. URL: https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1013.

[AMD23] AMD. AMD SEV-TIO: trusted i/o for secure encrypted virtualization. *Whitepaper*, 2023.

[AMD25] AMD. SEV secure nested paging firmware ABI specification. (Rev 1.58), 2025.

[ARM25] ARM. Arm confidential compute architecture software stack, 2025. URL: https://developer.arm.com/documentation/den0127/latest.

[Ava17] Roberto Avanzi. The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency S-boxes. *IACR Trans. Symmetric Cryptol.*, 2017(1):4–44, 2017. DOI: 10.13154/TOSC.V2017.I1.4-44.

[BBB+11] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 2011. DOI: 10.1145/2024716.2024718.

[BCG+23] Majed Valad Beigi, Yi Cao, Sudhanva Gurumurthi, Charles Recchia, Andrew C. Walton, and Vilas Sridharan. A systematic study of DDR4 DRAM faults in the field. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*, pages 991–1002. IEEE, 2023. DOI: 10.1109/HPCA56546.2023.10071066.

[BCG+24] Majed Valad Beigi, Yi Cao, Sudhanva Gurumurthi, Charles Recchia, Andrew C. Walton, and Vilas Sridharan. Corrigendum to: A systematic study of DDR4 DRAM faults in the field. *CoRR*, abs/2408.15302, 2024. DOI: 10.48550/ARXIV.2408.15302. arXiv: 2408.15302.

[BCT+25] Majed Valad Beigi, Yi Cao, Grace Tsai, Sudhanva Gurumurthi, and Vilas Sridharan. DDR5 DRAM faults in the field. In *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2025 - Supplemental Volume, Naples, Italy, June 23-26, 2025*, pages 36–41. IEEE, 2025. DOI: 10.1109/DSN-S65789.2025.00039.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[CRGB19] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: on the effectiveness of ECC memory against Rowhammer attacks. In *IEEE SP*, pages 55–71, 2019.

[CSB+26] Jalen Chuang, Alex Seto, Nicolas Berrios, Stephan van Schaik, Christina Garman, and Daniel Genkin. Tee.fail: breaking trusted execution environments via ddr5 memory bus interposition. In *47th IEEE Symposium on Security and Privacy (IEEE S&P '26)*. IEEE Computer Society, 2026. URL: https://tee.fail.

[DKPT17] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. Prime+Abort: a timer-free high-precision L3 cache attack using Intel TSX. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security*, pages 51–67. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen.

[DYM+17] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. *CoRR*, abs/1712.05090, 2017. arXiv: 1712.05090. URL: http://arxiv.org/abs/1712.05090.

[FGL+24] Antonio Flórez-Gutiérrez, Lorenzo Grassi, Gregor Leander, Ferdinand Sibleyras, and Yosuke Todo. General practical cryptanalysis of the sum of round-reduced block ciphers and ZIP-AES. In *Asiacrypt*, pages 280–311, 2024. DOI: 10.1007/978-981-96-0947-5_10.

[FVH+20] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: exploiting the many sides of target row refresh. In *IEEE SP*, pages 747–762, 2020.

[GJR25] Stefan Gloor, Patrick Jattke, and Kaveh Razavi. REFault: a fault injection platform for Rowhammer research on DDR5 memory. In *Proceedings of the Microarchitecture Security Conference*, 2025.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: a remote software-induced fault attack in JavaScript. In *DIMVA*, pages 300–321, 2016.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *DIMVA*, pages 279–299, 2016. DOI: 10.1007/978-3-319-40667-1_14.

[GSC+03] Blaise Gassend, G. Edward Suh, Dwaine E. Clarke, Marten van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity verification. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA'03), Anaheim, California, USA, February 8-12, 2003*, pages 295–306. IEEE Computer Society, 2003. DOI: 10.1109/HPCA.2003.1183547. URL: https://doi.org/10.1109/HPCA.2003.1183547.

[Gue16] Shay Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptol. ePrint Arch.*:204, 2016. URL: http://eprint.iacr.org/2016/204.

[HB17] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In *VEE*, pages 129–142, 2017. DOI: 10.1145/3050748.3050763.

[HRY24] Gal Horowitz, Eyal Ronen, and Yuval Yarom. Spec-o-Scope: cache probing at cache speed. In *CCS*, pages 109–123, 2024. DOI: 10.1145/3658644.3690313.

[HS10] Ruirui C. Huang and G. Edward Suh. IVEC: off-chip memory integrity protection for both security and reliability. In André Seznec, Uri C. Weiser, and Ronny Ronen, editors, *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, pages 395–406. ACM, 2010. DOI: 10.1145/1815961.1816015. URL: https://doi.org/10.1145/1815961.1816015.

[Int22a] Intel. Intel trust domain extensions. *Whitepaper*, 2022.

[Int22b] Intel. Intel® hardware shield - intel® total memory encryption, 2022. URL: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/white-paper-intel-tme.pdf.

[JLK+23] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. CSI:Rowhammer - cryptographic security and integrity against Rowhammer. In *IEEE SP*, pages 1702–1718, 2023. DOI: 10.1109/SP46215.2023.10179390.

[JNH+24] Jonas Juffinger, Sudheendra Raghav Neela, Martin Heckel, Lukas Schwarz, Florian Adamsky, and Daniel Gruss. Presshammer: Rowhammer and rowpress without physical address information. In *DIMVA*, pages 460–479, 2024.

[Kap17] David Kaplan. Protecting VM register state with SEV-ES. *Whitepaper*, 2017.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[KJQ+22] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: hammering from the next row over. In *USENIX Security*, pages 3807–3824, 2022.

[KPW16] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption, 2016.

[KWS+25] Nureddin Kamadan, Walter Wang, Stephan van Schaik, Christina Garman, Daniel Genkin, and Yuval Yarom. Ecc.fail: mounting rowhammer attacks on DDR4 servers with ECC memory. In Lujo Bauer and Giancarlo Pellegrino, editors, *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, pages 5679–5698. USENIX Association, 2025. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/kamadan.

[LJF+20] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 487–504. USENIX Association, 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lee-dayeol.

[LUSM24] Lukas Lamster, Martin Unterguggenberger, David Schrammel, and Stefan Mangard. Voodoo: memory tagging, authenticated encryption, and error correction through MAGIC. In *USENIX Security*, 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/lamster.

[LWW+22] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A systematic look at ciphertext side channels on AMD SEV-SNP. In *IEEE SP*, pages 337–351, 2022. DOI: 10.1109/SP46214.2022.9833768.

[LZW+21] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In *USENIX Security*, pages 717–732, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan.

[MHHW18] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: subverting AMD's virtual machine encryption. In *EuroSec*, 1:1–1:6, 2018. DOI: 10.1145/3193111.3193112.

[MWO+25] Jesse De Meulemeester, Luca Wilke, David F. Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. BadRAM: practical memory aliasing attacks on trusted execution environments. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, pages 4117–4135. IEEE, 2025. DOI: 10.1109/SP61157.2025.00104. URL: https://doi.org/10.1109/SP61157.2025.00104.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In David Pointcheval, editor, *CT-RSA*, pages 1–20, 2006. DOI: 10.1007/11605805_1.

[PTV21] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: overcoming the observer effect for high-precision cache contention attacks. In *CCS*, pages 2906–2920, 2021. DOI: 10.1145/3460120.3484816.

[Rog04] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *Asiacrypt*, pages 16–31, 2004. DOI: 10.1007/978-3-540-30539-2_2.

[SCG+03] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*, pages 339–350. IEEE Computer Society, 2003. DOI: 10.1109/MICRO.2003.1253207. URL: https://doi.org/10.1109/MICRO.2003.1253207.

[SD15] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer bug to gain kernel privileges. *Black Hat*, 15(71):2, 2015.

[SDA+25] Alexander Seto, Oytun Kuday Duran, Samy Amer, Jalen Chuang, Stephan van Schaik, Daniel Genkin, and Christina Garman. Wiretap: breaking server SGX via DRAM bus interposition. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*, pages 708–722. ACM, 2025. DOI: 10.1145/3719027.3765204. URL: https://doi.org/10.1145/3719027.3765204.

[Sil] Silvaco. 15nm Open-cell library and 45nm FreePDK. https://si2.org/open-cell-library/. URL: https://si2.org/open-cell-library/. Last accessed: 08/26/2025.

[SNR+18] Gururaj Saileshwar, Prashant J. Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K. Qureshi. SYNERGY: rethinking secure-memory design for error-correcting memories. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pages 454–465. IEEE Computer Society, 2018. DOI: 10.1109/HPCA.2018.00046.

[SSB20] David Suggs, Mahesh Subramony, and Dan Bouvier. The AMD "Zen 2" processor. *IEEE Micro*, 40(2):45–52, 2020. DOI: 10.1109/MM.2020.2974217.

[SWS25] Benedict Schlüter, Christoph Wech, and Shweta Shinde. Heracles: chosen plaintext attack on AMD SEV-SNP. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*, pages 3810–3824. ACM, 2025. DOI: 10.1145/3719027.3765209. URL: https://doi.org/10.1145/3719027.3765209.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 23(1):37–71, 2010. DOI: 10.1007/S00145-009-9049-Y.

[VDVFL+16] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: deterministic Rowhammer attacks on mobile platforms. In *CCS*, pages 1675–1689, 2016.

[WMA+19] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt of them all: inference attacks against secure virtual enclaves. In *AsiaCCS*, pages 73–85, 2019. DOI: 10.1145/3321705.3329820.

[WPWE23] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. Cipherfix: mitigating ciphertext side-channel attacks in software. In *USENIX Security*, pages 6789–6806, 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/wichelmann.

[WWME20] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: no security without integrity: breaking integrity-free memory encryption with minimal assumptions. In *IEEE SP*, pages 1483–1496, 2020. DOI: 10.1109/SP40000.2020.00080.

[YEP+06] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving cost, performance, and security of memory encryption and authentication. In *33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA*, pages 179–190. IEEE Computer Society, 2006. DOI: 10.1109/ISCA.2006.22.

[YF14] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.