UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Aus dem Institut für IT-Sicherheit
der Universität zu Lübeck
Direktor: Prof. Dr.-Ing. Thomas Eisenbarth

Security Analysis of Confidential VMs on Modern Server Architectures

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck

Aus der Sektion Informatik / Technik

vorgelegt von
Luca Wilke
aus Hildesheim

Lübeck, 2024

1. Berichterstatter: Prof. Dr. Thomas Eisenbarth
2. Berichterstatter: Prof. Dr. Frank Piessens
Tag der mündlichen Prüfung: 28.05.2025
Zum Druck genehmigt. Lübeck, den 24.06.2025

Abstract

Cloud computing has transformed data management and IT practices for organizations and individuals alike, offering unmatched scalability, flexibility, and cost-efficiency. However, it comes with privacy concerns, as the cloud service providers can access all processed data. Trusted Execution Environments (TEEs) are one potential solution, offering a new form of isolation that even locks out the infrastructure operator. Attacks from any software component outside the TEE are thwarted by novel access restrictions while physical attacks are prevented by memory encryption. Even the operating system or hypervisor cannot overcome these restrictions. With Intel SGX, Intel TDX, and AMD SEV-SNP, both major x86 CPU vendors offer TEEs on their server CPUs. This thesis scrutinizes the extent to which the current TEE generation delivers on their security promises.

We start this thesis by describing the isolation mechanisms implemented by SGX, TDX, and SEV-SNP. Building on these insights, we demonstrate that the trend to use deterministic memory encryption without integrity or freshness has several shortcomings. We show that monitoring deterministic ciphertexts for changes allows leaking information about the plaintext, which we exploit on SEV-SNP. SGX and TDX prevent straightforward exploitation by restricting software attackers from reading and writing the ciphertext, while SEV-SNP only restricts writing. Next, we challenge the security of such access restrictions by showing that an attacker with brief physical access to the memory modules can create aliases in the address space that bypass these safeguards. We exploit this on SEV-SNP to re-enable write access for software attackers, culminating in a devastating attack that forges attestation reports, undermining all trust in SEV-SNP. On SGX and TDX, such attacks are mitigated by a dedicated alias check at boot time.

Finally, we examine the security of VM-based TEEs against single-stepping attacks, which allow instruction-granular tracing and have led to numerous high-stakes attacks on SGX. We show that SEV-SNP is also vulnerable to single-stepping and provide a software framework enabling easy access to single-stepping on SEV for future research. Next, we analyze the single-stepping security of Intel TDX, which comes with a built-in mitigation comprising a detection heuristic and a prevention mode. We uncover a flaw in the heuristic that stops the activation of the prevention mode, thereby re-enabling single-stepping on TDX. Furthermore, we unveil an inherent flaw in the prevention mode that leaks fine-grained information about the control flow.

Kurzfassung

Cloud-Computing hat das Datenmanagement und die IT-Praktiken für Organisationen und Einzelpersonen gleichermaßen durch seine unvergleichliche Skalierbarkeit, Flexibilität und Kosteneffizienz transformiert. Es bringt jedoch Datenschutzbedenken mit sich, da die Cloud-Anbieter auf alle verarbeiteten Daten zugreifen können. Trusted Execution Environments (TEEs) sind eine mögliche Lösung für die Datenschutzbedenken. Sie bieten eine neuartige Form der Isolation, die sogar den Infrastrukturbetreiber umfasst. Angriffe von jedweder Software außerhalb der TEE werden durch neue Zugriffsbeschränkungen verhindert und physische Angriffe werden durch Speicherverschlüsselung verhindert. Selbst das Betriebssystem oder der Hypervisor können diese Restriktionen nicht überwinden. Mit Intel SGX, Intel TDX und AMD SEV-SNP bieten beide großen Hersteller von x86 CPUs TEEs auf ihren Server-CPUs an. Diese Doktorarbeit untersucht, inwieweit die aktuelle TEE-Generation ihre Sicherheitsversprechen einhält.

Wir beginnen mit einer detaillierten Erklärung, wie SGX, TDX und SEV-SNP ihre Isolationsgarantien umsetzen. Darauf aufbauend zeigen wir, dass der Trend deterministische Speicherverschlüsselung ohne Integrität oder Frische zu verwenden mehrere Schwächen aufweist. Wir zeigen, dass das Überwachen von deterministischen Chiffretexten auf Veränderungen, Informationen über den Klartext preisgibt, und demonstrieren Angriffe auf SEV-SNP. SGX und TDX verhindern eine einfache Ausnutzung solcher Angriffe, indem sie Lese- und Schreibzugriffe von Softwareangreifern auf den Chiffretext unterbinden, während SEV-SNP nur Schreibbeschränkungen implementiert.

Als Nächstes stellen wir die Sicherheit solcher Zugriffsrestriktionen infrage, indem wir zeigen, dass ein Angreifer mit kurzzeitigem physischem Zugang zu den Speichermodulen Aliase im Adressraum erzeugen kann, die diese Sicherheitsvorkehrungen umgehen. Wir nutzen dies auf SEV-SNP aus, um den Schreibzugriff für Softwareangreifer erneut zu ermöglichen, was zu einem verheerenden Angriff führt, der es erlaubt Attestierungsberichte zu fälschen und somit sämtliches Vertrauen in SEV-SNP untergräbt. SGX und TDX verhindern solche Angriffe durch eine dedizierte Suche nach solchen Aliasen während des Systemstarts.

Abschließend untersuchen wir die Sicherheit von VM-basierten TEEs gegen Single-Stepping-Angriffe, welche eine instruktionsgranulare Beobachtung der Ausführung

ermöglichen und bereits in mehreren gravierenden Angriffen auf SGX verwendet wurden. Wir zeigen, dass SEV-SNP ebenfalls anfällig für Single-Stepping ist, und stellen ein Software-Framework bereit, um Single-Stepping auf SEV für zukünftige Forschung zugänglich zu machen. Als Nächstes analysieren wir Intel TDX, dass über eine eingebaute Single-Stepping Gegenmaßnahme verfügt, die aus einer Erkennungsheuristik und einem Präventionsmodus besteht. Wir decken einen Fehler in der Heuristik auf, der die Aktivierung des Präventionsmodus verhindert und damit erneut Single-Stepping auf TDX ermöglicht. Darüber hinaus enthüllen wir einen inhärenten Fehler im Präventionsmodus, der feingranulare Informationen über den Programmablauf verrät.

Acknowledgements

First, I want to thank my supervisor, Thomas Eisenbarth, for his guidance and support and for always believing in me. You played a pivotal role in my decision to pursue a Ph.D. in the first place and I do not regret it one bit. Next, I want to thank Jan Wichelmann. You basically were a second supervisor during the early stages of my Ph.D. and taught me most of what I know about writing academic papers. Thank you to Jo Van Bulck for inviting me as a visiting scholar to KU Leuven. I had a fantastic time and deeply enjoy our ongoing collaboration. Thank you to all of my co-authors, without whom this Ph.D. would not have been possible: Diego F. Aranha, Sebastian Berndt, Jo Van Bulck, Mengyuan Li, Jesse De Meulemeester, Mathias Morbitzer, David Oswald, Anna Pätschke, Anja Rabich, Gianluca Scopelliti, Florian Sieck, Okan Seker, Akira Takahashi, Radu Teodorescu, Ingrid Verbauwhede, Jan Wichelmann, Greg Zaverucha, Yinqian Zhang. Finally, I want to thank my father and my sibling for always being there for me, especially during the darker times of my Ph.D. journey.

Luca Wilke
Lübeck, December 2024

Contents

Part I

Security Analysis of Confidential VMs on Modern Server Architectures

1

Introduction

The shift toward cloud computing has revolutionized how individuals and organizations manage their data and computation. As businesses strive for scalability, flexibility, and cost-efficiency, the cloud has become an indispensable infrastructure. An essential building block for cloud computing is sharing the same physical hardware between multiple tenants using compartments called *Virtual Machines (VMs)* for isolation. Initially, security efforts centered on safeguarding the cloud service provider from malicious tenants and ensuring isolation between tenants. However, soon it crystallized that data privacy is one major hindrance for moving computations to the cloud, as the cloud service provider can spy on the data processed on its infrastructure. Regular encryption schemes can only protect data at rest and in transit but not data that is being computed on. While, *Fully Homomorphic Encryption (FHE)* schemes [38] and *Garbled Circuits* [10] can solve this issue, they have not been broadly adopted due to their performance overhead when protecting general purpose computations. *Trusted Execution Environments (TEEs)* have emerged as a powerful alternative, providing hardware-based mechanisms to isolate and protect computations from the infrastructure operator. They aim to protect against attacks from all software on the host system and also against attackers with physical access. To achieve this, they build on a hardware root of trust that is part of the CPU itself. Currently, there are three major TEEs available on x86 server systems: Intel SGX [31], AMD SEV(-SNP) [4] and Intel TDX [55]. Intel SGX was released in 2015 and isolates individual processes, while AMD SEV (announced 2015, most recent update SEV-SNP in 2020) and Intel TDX (announced 2021) protect Virtual Machines (VMs). VMs protected by TEEs are also referred to as confidential VMs (cVM). All three TEEs are publicly available in commercial clouds [39, 88]. However, recent advancements focus more on AMD SEV and Intel TDX as they promise a lift-and-shift solution for securing existing VM deployments, which are the building block of most cloud-based infrastructure.

Since the release of Intel SGX, TEEs have been scrutinized by the academic research community. In this thesis, we group the resulting attacks into four categories: attacks on memory encryption, attacks on architectural isolation, attacks on microarchitectural isolation, and physical attacks.

All considered TEEs encrypt their data before writing it to memory via a hardware memory encryption unit built into the memory controller of the CPU. The main intention

is to safeguard data against physical attacks that try to read the information directly from the memory modules, such as cold boot attacks [128]. The initial Intel SGX design employed strong memory encryption with cryptographic integrity and freshness. While secure against adaptive attacks, this approach significantly constrained the ability of SGX to handle large amounts of memory, limiting its applicability for many industry use cases. AMD SEV, Intel TDX and the updated, scalable SGX thus opted for memory encryption based on lightweight tweaked block ciphers without integrity or freshness for their memory encryption. As we will learn throughout this thesis, on its own, such a design only offers sufficient protection in scenarios like a cold boot attack, where the attacker can only access a single snapshot of the encrypted data. However, without additional means, the way that TEEs integrate into the untrusted host system enables software-level attackers to read and write the encrypted data repeatedly. Several attacks, including contributions of this thesis, demonstrated that the ability to manipulate or repeatedly read the encrypted data is sufficient to subvert the isolation guarantees of SEV [14, 76, 79, 80, 123]. With SEV-SNP, AMD introduced write restrictions, while Intel SGX and TDX prohibit reading and writing.

After loading data into the CPU, TEEs must ensure proper architectural and microarchitectural isolation to guard against software attackers. Such isolation properties have been extensively studied in "classic" attacker models, where an unprivileged process or a VM tries to infer information about other parts of the system. Most attacks are based on timing variations when querying microarchitectural structures like data and instruction caches [7, 85, 97, 127], address translation caches [40, 115], branch predictors [34, 35] and execution ports [2]. The strong TEE attacker model, which includes privileged software components like the operating system or the hypervisor, helps to significantly reduce the noise for such attacks. In addition, it includes several side-channels that would be out of scope with the standard attacker model that assumes a trusted operating system and hypervisor. Most of these side-channels revolve around the fact that the untrusted host needs to remain in control of certain hardware components, such as hardware timers and resource management, like memory allocation. Using control over memory allocation, an attacker can infer the memory access patterns of TEEs with page granularity [126], which is known as the page fault controlled-channel. Later, it was shown that the host system's control over the APIC timer allows it to interrupt the TEE after every instruction, resulting in instruction-granular traces of its execution [20, 122, 124]. This type of attack is commonly known as single-stepping. Combined with well-established side-channels, such as cache attacks, single stepping provides attackers with a powerful toolkit for monitoring the code executing inside the TEE. Combined with partial knowledge about the executing code, such observations can be used to leak information about data that influences the control flow or memory accesses of the program. Most publications focus on cryptographic libraries and extract secret keys [12, 89, 91, 108]. However, Xu

et al. [126] show that general-purpose software, like image decoding, is also affected, leaking information about the processed data.

In principle, well-known implementation techniques called data oblivious constant time prevent such leakages. However, research has repeatedly shown that implementing data oblivious constant time correctly, especially against a single-stepping adversary, is error-prone [91, 108]. While rewriting small applications or cryptographic libraries to use this technique is feasible, rewriting all software is infeasible. Thus, data oblivious constant time is ill-suited for the VM-based TEE model, given its large and complex software stacks. Due to the severity of these issues, which led to numerous exploits against SGX and SEV, Intel TDX comes with a built-in countermeasure against single-stepping. In addition, Intel SGX recently [30] received an update to retrofit a countermeasure. However, the TDX countermeasure still allows for a slightly weaker subvariant of single-stepping [122] and neither of the countermeasures prevent the page fault side-channel.

1.1 Main Contributions

In this thesis, we analyze the security of TEEs on modern server architectures, focusing on the two VM-based solutions AMD SEV and Intel TDX. We show novel attacks and propose improved TEE designs to mitigate them.

In summary, we:

- **Show limits of deterministic memory encryption for TEEs** To be able to scale to large memory sizes, modern TEEs like AMD SEV-SNP, Intel TDX, scalable Intel SGX and ARM CCA all opted to use deterministic memory encryption. On their own, these ciphers leak whether the same plaintext value is encrypted more than once and whether a write operation changes the underlying plaintext. Many applications and algorithms that are commonly used inside TEEs leak secret data through their memory access behavior if the attacker can observe the aforementioned patterns. Thus, most TEEs implement access rights mechanisms to prevent attackers from observing the ciphertext. We show that the countermeasures implemented by SEV-SNP are insufficient. Based on this flaw, we show end-to-end attacks that leak encryption keys from state-of-the-art constant time implementations. Finally, we propose mechanisms to mitigate the issue in software and provide a proof-of-concept implementation to prevent the leakage of register values of userland applications when context switching to the kernel. Wichelmann et al. [118, 119] build on these ideas and provide automated tools to protect user space applications themselves. Deng et al. [32] show how to automatically detect vulnerable code locations.

- **Show security risks of memory encryption without integrity and freshness for TEEs** To support scaling to large memory sizes, modern TEEs such as AMD SEV-SNP, scalable Intel SGX, and ARM CCA have adopted block ciphers that lack robust cryptographic integrity or freshness guarantees for encrypting their memory content. Only Intel TDX offers integrity protection via a 28-bit MAC. However, the MAC is not backed by secure on-chip memory but resides on the DRAM together with the data it aims to protect and is only secure if protected against brute-force attacks. Without integrity and freshness, a memory encryption scheme cannot detect if the ciphertext has been manipulated or if an old ciphertext has been replayed. To prevent attacks based on manipulated ciphertexts, most TEEs implement access rights-based checks that prevent attackers from manipulating the raw ciphertext. We show an attack mechanism that creates aliases on the physical address layer, defeating the access rights checks of SEV-SNP and possibly ARM CCA. We demonstrate a powerful end-to-end attack against SEV-SNP that enables attackers to forge attestation reports and thus fully undermine the system's root of trust.

- **Show Single-Stepping Attacks on Confidential VMs** Prior to our work, single-stepping attacks have not been explored for VM-based TEEs. We show that both AMD SEV-SNP and Intel TDX are vulnerable to this class of attacks. On both platforms, the core mechanism of using the APIC timer as an interrupt source remains unchanged. However, the fact that VMs are interrupt-aware and run a fully-fledged operating system poses additional challenges that we need to overcome. For SEV-SNP, we provide an extensive, reusable software framework that already found some adoption [37, 131].

  In contrast to SEV-SNP, Intel TDX comes with a dedicated single-stepping countermeasure that tries to detect single-stepping attacks via a heuristic and subsequently activates a prevention mode to prevent the attacker from repeatedly interrupting after a deterministic number of instructions. We analyze its security guarantees in-depth and discover two flaws. The first one shows a flaw in the heuristic that allows circumventing the activation of the prevention mode completely and thus re-enables single-stepping attacks. In addition, we show a new technique for filtering zero-steps, as the previously used methods do not work with Intel TDX. The second flaw exploits a weakness in the design that still allows the attacker to infer the number of executed instructions. While weaker than single-stepping, such instruction counting attacks still reveal fine-grained information about the control flow, enabling exploitation of non-constant time behavior. We demonstrate the feasibility of our attack primitives by leaking cryptographic keys from wolfSSL and OpenSSL.

- **Identify New Attack Surfaces in Legacy VM Components** One of the main allures of confidential VMs is their promise to secure existing VM workloads without requiring extensive changes. We show that traditional software components expose major security risks when used in the context of confidential VMs. LUKS2, the de facto standard for disk encryption allows an attacker to trick Linux into using a decryption primitive that leaks the key via the cache side-channel. The OVMF UEFI version (when writing the undeSErVed trust paper) mapped all pages as executable, making it easy to inject code by exploiting flaws in the memory encryption.

1.1.1 Individual Publications

This section gives a summary of all papers that make up the main contributions of this thesis. The full text of all publications can be found in Part II.

A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP In this paper, we provide novel insights on the applicability of ciphertext side-channels to TEEs. This class of attacks was first demonstrated in the CipherLeaks paper by Li et al. [80], but they only analyzed the effect on SEV's VMSA data structure, which stores the VM's register values during context switches to the hypervisor. AMD released a firmware patch to secure the VMSA against ciphertext side-channels. In our paper, we demonstrate that ciphertext side-channel attacks affect the whole memory of confidential VMs and categorize them into two primitives: dictionary attacks and collision attacks. We demonstrate end-to-end attacks against OpenSSL and OpenSSH that leak encryption keys. Furthermore, we show that the way Linux stores user space register during context switches to the kernel is highly susceptible to ciphertext side-channel attacks. This enables the construction of a general-purpose primitive that can be used to leak partial register information from any user space process inside the confidential VM. These attacks clearly show that AMD's mitigation is insufficient. Furthermore, we analyze both hardware and software countermeasures to secure TEEs against the presented attacks. We provide a proof-of-concept implementation to secure the context switch mechanism of Linux. Follow-up work by Wichelmann et al. [118, 119] builds on our ideas to develop automated tooling to protect applications against ciphertext side-channel attacks.

This paper was published at the IEEE S&P 2022 conference and is joint work with Mengyuan Li, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, Yinqian Zhang [76]. The full text is in Chapter 9.

SEV-Step: A Single-Stepping Framework for AMD-SEV In SEV-Step, we demonstrate that SEV-SNP virtual machines can be reliably single-stepped using the APIC timer. Previously, single-stepping attacks were demonstrated for process-based TEEs,

particularly for Intel SGX. In addition, we provide a reusable framework to facilitate future research, which has already gained some traction in the community [37, 131]. Besides single-stepping, we also show that AMD CPUs/SEV VMs leak information about the executed instruction via the interrupt latency, similar to the results in prior work for Intel SGX [19]. We demonstrate the capabilities of our framework in an attack case study against LUKS2 disk encryption. In this scenario, the attacker manipulates the unauthenticated metadata of the encrypted disk to trick the VM into decrypting the disk with an implementation that is vulnerable to cache-side channels.

SEV-Step was published at the *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2024 [124]. The full text is in Chapter 7. SEV-Step is joint work with Jan Wichelmann, Anja Rabich and Thomas Eisenbarth.

TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX In TDXdown, we analyze the built-in single-stepping countermeasure of Intel TDX and show two attacks: full single-stepping and *StumbleStepping*. Intel's countermeasure is based on a heuristic that tries to detect single-stepping attacks and a prevention mode that gets activated by the heuristic to prevent the attack. Our first attack exploits a flaw in the detection heuristic whose decision is partially based on a timing measurement. By throttling the frequency of the CPU core, the attacker can ensure that the heuristic never detects single-stepping attempts. As a result, the prevention mode is never activated, allowing to single-step TDX with techniques similar to prior work [20, 124]. For our second attack, *StumbleStepping*, we exploit an inherent flaw in the prevention mode that leaks the number of instructions executed by the TD, which is still sufficient to exploit secret-dependent control flow. As a separate contribution, we provide an in-depth analysis of secret dependent control flow in nonce truncation implementations in state-of-the-art crypto libraries. Using our single-stepping and StumbleStepping primitives, we exploit nonce truncation leakages in wolfSSL and OpenSSL. Finally, we discuss improved designs for single-stepping countermeasures. After our disclosure, Intel updated the detection heuristic but did not change the prevention mode. Instead, they refer developers to their implementation guidelines for preventing secret-dependent control flow.

TDXdown was published at the ACM CCS 2024 conference and is joint work with Florian Sieck and Thomas Eisenbarth [122]. The full text is in Chapter 8.

BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments In BadRAM, we show how manipulating the information on the SPD chip on DDR4 and DDR5 DRAM modules can be used to create aliases in the physical address space of the CPU. For most memory modules, the manipulation requires brief physical access, while some memory modules allow overwriting this information from software. The aliases can

be used to overcome physical address-based access rights checks, as, e.g., used by Intel SGX and SEV-SNP. Without the access rights check for write protection, SEV-SNP cannot uphold its integrity guarantees and again becomes susceptible to ciphertext manipulation, replay attacks, and manipulations of the address space layout. We show an end-to-end attack that breaks the remote attestation mechanism of SEV-SNP, undermining all trust in the system. For classic Intel SGX, the strong memory encryption which provides freshness and integrity, prevents such attacks but still allows for fine-grained write pattern leakage. Scalable Intel SGX and Intel TDX perform a dedicated alias check during system boot and deactivate themselves if aliases are found. AMD released firmware updates to mitigate our attacks.

BadRAM was published at the IEEE S&P 2025 conference and is joint work with Jesse De Meulemeester, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, Jo Van Bulck [87]. The full text is in Chapter 9.

undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation In undeSErVed trust, we show a flaw in the (remote) attestation of SEV and SEV-ES, allowing an attacker to generate the same attestation value for each 16-byte granular permutation of the initial VM image. Based on this flaw, we demonstrate how an attacker can reorder the code of the widely used OVMF UEFI, to set up a return oriented programming attack that eventually allows executing arbitrary code in the SEV VM. We demonstrate an end-to-end attack where the attacker leaks the keys of the VM's encrypted disk. While we propose countermeasures to increase the complexity of exploiting the found issues, we conclude that they cannot be fully mitigated for SEV-ES due to the lack of integrity protection for the VM's memory layout. Independent of our paper, AMD addressed these issues with SEV-SNP which is not affected by our attacks.

undeSErVed trust was published at the *15th IEEE Workshop on Offensive Technologies* in 2021 and won the best paper award [125]. The full text is in Chapter 5. undeSErVed trust is joint work with Jan Wichelmann, Florian Sieck and Thomas Eisenbarth.

1.2 Other Contributions

In addition to the previously discussed main contributions, I also worked on several publications that are not part of this thesis. In the following, I will briefly summarize the papers and my contributions.

SEVurity: No Security Without Integrity Breaking Integrity-Free Memory Encryption with Minimal Assumptions In SEVurity, we analyze the security of SEV-ES against ciphertext manipulation, showing code injection attacks simply by re-using existing ciphertext blocks at new memory locations. SEV-ES was the most recent version of SEV when the paper was published.

Prior research uncovered that SEV uses AES in the Xor-Encrypt mode, where a physical addressed-based tweak is XORed to the plaintext before encryption. We provide additional details on reverse engineering the exact tweak values and show that more recent AMD CPUs use the Xor-Encrypt-Xor mode, which offers better security guarantees.

Knowledge of the encryption mode and tweak values is a crucial prerequisite for our code injection attack. For the attack, we first build a dictionary of known plaintext ciphertext blocks. To this end, we assume that the kernel version is known and show how an attacker can locate the ciphertext of the kernel code in memory to build the dictionary. To inject code at a memory location, the attacker searches the dictionary for a ciphertext block that would decrypt to the desired instruction when copied to the new memory location. This step requires knowledge of the physical address-based tweak values. For the Xor-Encrypt-Xor mode, the attacker also needs to apply the XOR difference between the old and the new tweak value to the ciphertext before decryption. Otherwise, the plaintext would be randomized during decryption. Using our dictionary, we show that the attacker can gain control over 2 bytes out of a 16-byte ciphertext block. Using this primitive, we inject a code gadget into the SEV VM that allows the attacker to encrypt arbitrary plaintext, i.e., the attacker gains control over all 16 bytes. The injected values can be executed as code or used as data.

This paper was published at the IEEE S&P 2020 conference [123]. I am the first author, with Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth as co-authors. SEVurity is the paper version of my master thesis and thus is not included as a main contribution in this thesis.

SNI-in-the-head: Protecting MPC-in-the-head Protocols against Side-channel Analysis In this paper, we analyze the security of proof systems based on the MPC-in-the-head paradigm against differential power analysis side-channel attacks. We show that the MPC-in-the-head approach does not provide inherent side-channel security. In a case study, we demonstrate an attack on the ZKBoo protocol, which is for example used by the Picnic post-quantum signature scheme. Afterward, we propose $(n+1)$-ZKBoo, a refined version of the protocol, which is immune to the presented attacks by using gadgets with the "strong non-interference" property. We apply $(n+1)$-ZKBoo to Picnic and conduct practical experiments to verify its security against differential power analysis.

The paper was published at the *ACM SIGSAC Conference on Computer and Communications Security* in 2020 and is joint work with Okan Seker, Sebastian Berndt, and Thomas Eisenbarth. I worked on the implementation and performance evaluation of the proposed countermeasure in Picnic.

Side-Channel Protections for Picnic Signatures In the paper, we analyze masking countermeasures for signature schemes based on the MPC-in-the-head paradigm to protect them against side-channel attacks, focusing on physical side-channels like EM emanations. We show that the state-of-the-art masking scheme for MPC-in-the-head is vulnerable when used with the Picnic signature scheme. Next, we develop an improved masking scheme, that is also easier to integrate into other schemes and has a smaller overhead on the signature size. We implement our masking approach for the Picnic3 signature scheme and validate our implementation by performing extensive EM emanations experiments.

The paper was published at the *Conference on Cryptographic Hardware and Embedded Systems (CHES)* in 2021 and is joint work with Diego F. Aranha, Sebastian Berndt, Thomas Eisenbarth, Okan Seker, Akira Takahashi and Greg Zaverucha (alphabetical order) [8]. I performed the EM emanation experiments.

Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software In Cipherfix, we harden existing binaries to be secure, even if they are executed in a TEE that is vulnerable to ciphertext side-channel attacks. We focus on cryptographic implementations and use taint tracking to find all memory locations storing sensitive data. Afterward, we use static binary instrumentation on the original binary code to rewrite all accesses to secret data to use masking. The core masking/unmasking step only uses CPU registers, which we assume to be secure against ciphertext side-channels, as discussed in prior work. Since the masks change on every write, so are the corresponding ciphertexts, mitigating the ciphertext side-channel.

The paper was published at the *32nd USENIX Security Symposium* in 2023 and is joint work with Jan Wichelmann, Anna Pätschke and Thomas Eisenbarth [118]. I helped in the conception of the countermeasure and the performance evaluation.

SNPGuard: Remote Attestation of SEV-SNP VMs Using Open Source Tools In this SoK-style paper, we show which steps are required to properly attest SEV-SNP VMs for two different use cases. For the first use case, we ensure the confidentiality and integrity of the payload code executing in the VM, which requires to establish a secure channel to transmit a disk encryption key. In the second use case, we only ensure the integrity, making the startup process less involved. While the individual software components required for these workflows already exist, their documentation is scattered

and scarce. Furthermore, they are not trivial to integrate with each other. We provide additional documentation in the paper and combine all components in an easy-to-use and easy-to-extend software framework.

The paper was published at the *7th Workshop on System Software for Trusted Execution (SysTEX'24)* workshop in 2024 and is joint work with Gianluca Scopelliti [121]. I am the main author.

1.3 Outline

This thesis is structured into three parts. Part I contains background information and a detailed overview of the state of the art regarding attacks and defenses on TEEs. Part II contains the full text of the publications that form the main contribution of this thesis. **??** contains a short CV, providing an overview of my academic career. The first part is structured as follows.

In Chapter 2, we provide background information on the x86 architecture with a focus on its isolation mechanisms. We show how hardware-accelerated virtualization extends these mechanisms before closing the background chapter with an introduction to TEEs.

We start Chapter 3 with a detailed, technical description of how Intel SGX, AMD SEV, and Intel TDX implement their isolation guarantees, complementing the background information from the previous chapter. Afterward, we provide a detailed summary of the research landscape regarding TEE security, grouping the results into four categories: attacks on memory encryption (Section 3.2), attacks on architectural isolation (Section 3.3), attacks on microarchitectural isolation (Section 3.4) and physical access attacks (Section 3.5). Chapter 4, closes the first part with my conclusion of the current state of TEE security and an outlook to interesting challenges for future work.

2

Background

In this chapter, we provide essential background information on the x86 architecture and on *Trusted Execution Environments (TEEs)*. For the x86 architecture, we focus on its isolation mechanisms since these are the most relevant to understanding TEEs. We start with general concepts like CPU privilege levels and virtual memory and then proceed to hardware-accelerated virtualization on modern x86 CPUs. Afterward, we introduce the general concept of TEEs and refer the reader to Section 3.1 for a detailed technical description of the isolation mechanism used by the TEEs considered in this thesis.

2.1 x86 Systems Architecture

In this section, we provide a succinct description of the features of the x86 CPU architecture that are important for this thesis. Furthermore, we explain basic operating systems concepts and how they are implemented on x86. For a more detailed description, we refer the reader to the excellent books *Computer Architecture - A Quantitative Approach* [43] and *Operating Systems: Three Easy Pieces* [9] on which this section is based.

The x86 architecture defines a set of instructions, general purpose and system/configuration registers that software can use to interact with the CPU. Instructions are the building blocks for programs and range from simple arithmetic operations like an addition (`add`) to memory accesses (`mov`) or control flow transfer instructions (`call`, `jmp`). General purpose registers offer readily accessible storage and serve as operands for instructions. Finally, there are various system/configuration registers that allow configuring the behavior of the CPU, or track state, like the address of the currently executing instruction.

2.1.1 Isolation Mechanisms

An *operating system* is the fundamental software running on the CPU. Its task is to manage resources such as CPU time, memory, and attached hardware devices like hard drives or network cards. It offers an API with high-level abstractions for these resources that can be used by regular applications. The core part of the operating system is called *kernel*.

Another important task for modern operating systems is to implement strict security policies that prevent, e.g., a malicious or buggy program from interfering with other programs or the operating system itself. To this end, operating systems make use of the features of the x86 architecture to implement a privilege hierarchy and to isolate different programs, which are also referred to as processes, using the concept of privilege levels and virtual memory. In this thesis, we focus on the Linux operating system, which is the most popular operating system for severs. However, most concepts apply to all operating systems.

Privilege Levels The first mechanism that the operating system uses to implement its security and isolation guarantees is privilege levels. The x86 architecture supports four hierarchical privilege levels, which are referred to as *ring 0* to *ring 3*. Ring 0 is the most privileged level and is usually used for the operating system, while ring 3 has the least privileges and usually runs regular user programs. Ring 0 is also referred to as *kernel mode* or *kernel space* while ring 3 is commonly referred to as *userland* or *user space*. The current privilege level mainly restricts the kind of instructions that can be executed and which CPU configuration registers can be accessed. Instructions that can only be executed in ring 0 are also referred to as privileged instructions. Important examples are access to the `cr3` configuration register, which controls the current address space, and the `lidt` instruction for configuring interrupt handlers. In addition, certain configuration features, like the access rights configuration stored in page tables, may also reference the current privilege level to impose restrictions. After power on, the CPU starts in ring 0, and the operating system sets up its isolation mechanisms by preparing address spaces and configuring interrupts before eventually executing user space applications by dropping to ring 3.

Virtual Memory The second important isolation mechanism that operating systems use is virtual memory. Due to legacy reasons, the x86 architecture supports several addressing modes. We restrict ourselves to *64-bit long mode*, which is the addressing mode that modern operating systems set up. On x86, we distinguish between two kinds of addresses: *physical addresses* and *virtual addresses*. A physical address is of global scope and refers to a unique location in memory. It is used by the *Memory Management Unit (MMU)* of the CPU to load the requested location from the memory modules. A virtual address, on the other hand, is only valid in the scope of the current address space, which is defined by the page tables pointed to by the `cr3` register. Virtual addresses are translated to physical addresses via page tables. Section 2.1.1 shows an overview of the translation process for a three-level page table. The Linux kernel uses five-level page tables by default on most x86 systems.

When software performs a memory access, the CPU uses the page tables referenced by the `cr3` register to translate the virtual address to a physical address. Besides the

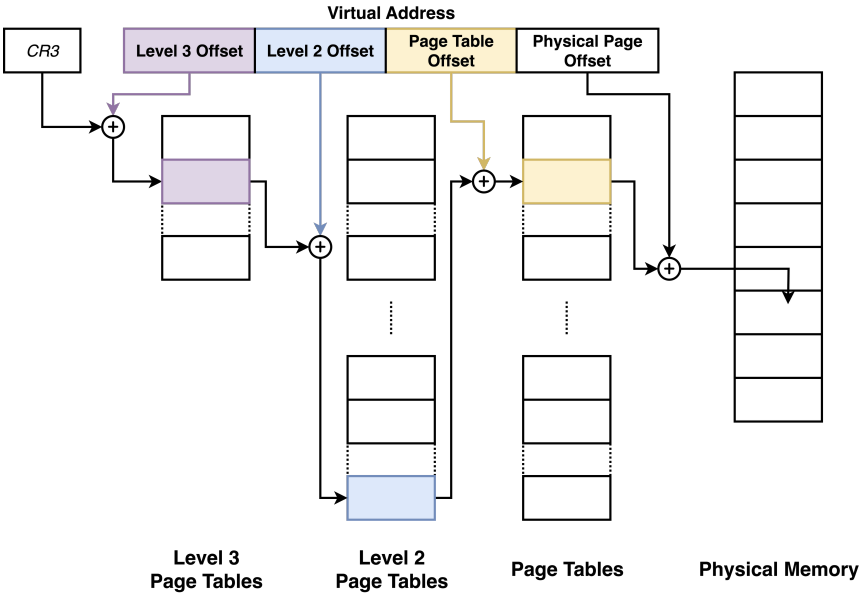


Figure 2.1: Example of a three-level page table. The `cr3` register points to the root of the page table. The entries of the intermediate page table stages point to the base address of the next stage, while the entries of the final page table stage point to 4096-byte aligned physical addresses. In addition to the address, page table entries also contain configuration metadata, like access rights. The virtual address is divided into multiple parts, which specify the offsets in the intermediate page table stages and the offset in the final physical address.

address translation, page tables allow specifying access rights for a virtual address, like "read-only", "read+write", and "execute". In addition, page tables can restrict access to a virtual address to ring 0. By default, page tables manage memory in chunks of 4096 bytes, which are referred to as pages. Thus, all access rights have at least 4096-byte granularity. The main advantage of page tables is that they allow the operating system to virtualize physical memory, providing each process with its own view on memory. The concept of virtual memory enables the operating system to isolate itself from userland but also userland processes from each other. In addition, it allows processes to use arbitrary addresses without the need to consider the memory usage of other processes.

Interrupts Operating systems isolate themselves from user space processes using the privilege levels and page table mechanisms of the CPU. Interrupts are used to allow the operating system to regain control if certain events occur. In more detail, the x86 architecture differentiates between interrupts, exceptions, and traps. Interrupts refer to events that are asynchronous to the currently executing code, like a package arriving at the network card. Exceptions and traps are, instead, related to the currently executing instruction. Exceptions refer to error conditions, like a division by zero or an invalid memory access, e.g, due to insufficient permissions (page fault exception). Finally, traps are caused by special instructions that consciously want to trigger a context switch to the operating system. The most common example is the `syscall` instruction, which user space application issue to use the API exposed by the operating system. To handle interrupts, exceptions and traps, which we also refer to as events, the operating system prepares an *Interrupt Descriptor Table (IDT)* and activates it via the privileged `lidt` instruction. For each event, the IDT includes a configuration entry that specifies a software routine to handle it. When an event occurs, the CPU automatically transitions to ring 0 and invokes the handler defined in the IDT. Additionally, the CPU saves a reference to the next instruction of the interrupted program, allowing the operating system to resume execution at that point using the *iret* instruction. However, the operating system is responsible for storing the remaining state, like the values of the general purpose registers.

One crucial interrupt on the x86 architecture is the APIC timer interrupt. This is a countdown timer that the operating system programs to ensure that it regains control after a fixed amount of time. The primary use case is to implement scheduling algorithms that run multiple user space processes in a time-sliced fashion.

2.1.2 Caches

Building storage that is large, delivers high throughput and maintains low random access latency presents a significant technical challenge. As a result, modern CPUs

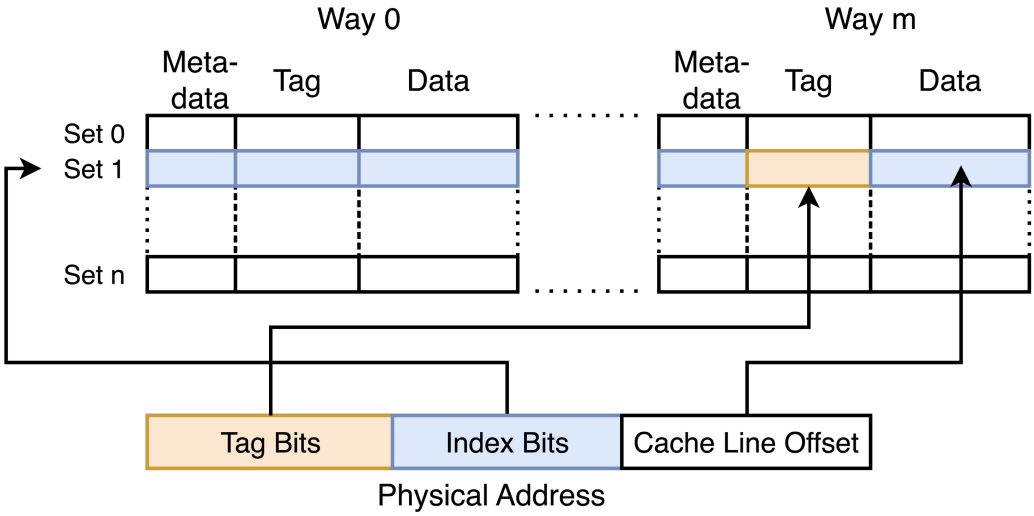


Figure 2.2: Cache with n sets and m ways. The index bits are used to select a cache set. Next, the cache tag and the valid bit in the metadata bits are used to select the correct entry from the m ways. Finally, the cache line offset bits determine the byte granular position in the cache line.

can execute much faster than the main memory is able to serve requests. The main memory is also referred to as *Dynamic Random Access Memory (DRAM)* and is located on external memory modules called *Dual Inline Memory Modules (DIMMs)* that the CPU communicates with over the memory bus using the *Double Data Rate (DDR)* protocol. To alleviate the performance impact of the relatively slow DRAM, the CPU comes with a set of small but very fast cache memory. The cache memory is organized in a hierarchy, usually ranging from level 1 (L1) to level 3 (L3), with L1 being the fastest and smallest cache. On L1, there is usually one dedicated cache for instructions and one dedicated cache for data. On all other levels, code and data are usually stored in the same cache. Accessing the L1 cache takes 3 to 5 CPU clock cycles, while the L2 cache takes 10 to 20 cycles, and the L3 cache takes 30 to 50 cycles. A DRAM access takes roughly 200 to 500 cycles. Usually, each CPU core has its own L1 cache and L2 cache, while the L3 cache is shared between multiple cores. Whenever a memory access is performed, the CPU first checks whether the data already resides in the cache before accessing the DRAM. Similarly, writes can also be cached, postponing the slow DRAM access. Caches are organized into *sets* and *ways*, as depicted in Figure 2.2. Each memory address is assigned to precisely one cache set via a hash function that is based on the virtual address (L1 cache) or parts of the physical address (L2,L3 cache). Each cache set consists of multiple ways, each of which can store a 64-byte chunk of memory, which is commonly referred to as a cache line. In addition, for each way, the cache stores a unique identifier for the cached address called *cache tag* as well as some metadata like a *valid* bit or *dirty* bit. The cache tag is derived from the physical address. When querying a cache for a given

address, all ways of the corresponding cache set are accessed in parallel, checking the cache tag and the valid bit to identify if there is a match. After an initial warm-up phase, all cache ways usually contain valid entries. Thus, subsequent memory accesses that are not stored in the cache need to evict an existing entry. While CPU vendors do not disclose their replacement policies, independent research revealed that most modern AMD and Intel CPUs use an approximation of the *Least Recently Used (LRU)* algorithms called *Pseudo LRU (PLRU)* [13]. A long line of research exploits the timing difference between a cache hit and a cache miss as a side-channel to leak information about the memory accesses of other security domains on the system [7, 85, 97, 127].

In addition to the caches for DRAM data, the CPU also has caches to store address translations from the page tables. This class of caches is called *Translation Lookaside Buffer (TLB)* and is also organized in a hierarchy.

2.2 x86 Virtualization

This section provides a concise introduction to hardware-accelerated virtualization on modern AMD and Intel CPUs. For additional details, we refer to the book *Modern Operating Systems* [113] and to AMD's and Intel's manuals [5, §15],[51, §24]. Since most publications in this thesis focus on AMD CPUs, certain details in the following description, especially the names of instructions, are specific to AMD CPUs. However, Intel's and AMD's approach to hardware-accelerated virtualization is very similar.

In recent years, industry use cases shifted heavily towards cloud computing, where cloud service providers offer computational resources as a service. For efficient resource usage, cloud service providers run workloads for multiple mutually distrusting customers on the same hardware. While the process abstraction provided by operating systems enables running applications from different users on shared hardware, the shared operating system kernel presents a large attack surface for potential exploits by malicious programs. In addition, customers might require a specific operating system or version. Virtualization addresses these issues by introducing a new kind of CPU compartmentalization, called *Virtual Machines (VMs)*. For the contained software, a VM simulates all interfaces of a bare metal x86 computer system, enabling to run whole operating systems with their own set of user space applications inside the VM. The software component that manages the VMs is referred to as the *hypervisor* or more general *host system*. With virtualization, the operating system does not need to be adapted to run in a VM. However, for efficiency reasons, some operating systems include paravirtualized drivers that can consciously communicate with the hypervisor, allowing the use of more efficient interfaces. In the

following, we describe the three main areas that require virtualization: CPU privilege levels, memory subsystem and interrupts.

2.2.1 CPU

As discussed in Section 2.1, x86 CPUs support four different privilege levels. To virtualize unmodified operating systems, the hypervisor needs to enable the VM to execute privileged instructions, which can only run on ring 0. At the same time, the hypervisor also needs to protect itself from the VM. To resolve this clash, hardware accelerated virtualization adds a new dimension to the privilege levels which we refer to as *host mode* and *guest mode*. Figure 2.3 shows an overview. In each mode, software has access to all CPU privilege levels. On AMD CPUs, the host mode can transfer the CPU into guest mode using the newly added `VMRUN` instruction, which takes a pointer to the *Virtual Machine Control Block (VMCB)* data structure as an argument. Using the VMCB, the host mode can configure the hardware such that certain instructions are *intercepted* and trap to the host mode, which is referred to as a #`VMEXIT`. There are two main use cases for interception. The first use case is to intercept the execution of certain instructions in order to emulate them. For example, the hypervisor might want to intercept the `cpuid` instruction to modify the information about specific hardware details. Another example are the `rdmsr` and `wrmsr` instructions, which are used to access configuration registers. The second use case is intercepting certain interrupts, exceptions, or traps. For example, the hypervisor usually wants to ensure that the APIC timer interrupt is intercepted to ensure that it always regains control after a fixed time slice. Another important example is handling page fault exceptions, which we will discuss in the next section.

In addition to event interception, hardware-accelerated virtualization also saves and restores parts of the CPU register state to facilitate context switching between the two modes. Among others, this includes the instruction pointer and the value of certain configuration registers for the address space (`cr3`), interrupt handlers (`idtr`), and interrupt enablement status (`rflags`).

2.2.2 Memory

Hardware-accelerated virtualization introduces a second level of page tables, known as *Nested Page Tables (NPTs)* on AMD systems, to facilitate efficient memory virtualization. The NPTs are managed by the hypervisor and inaccessible to the VM. When context switching into a VM, the hypervisor specifies the memory address of the root of the NPT via a field in the VMCB. In guest mode, the VM manages the guest page tables via the regular `cr3` register. From the VM's point of view, these are the only page tables.

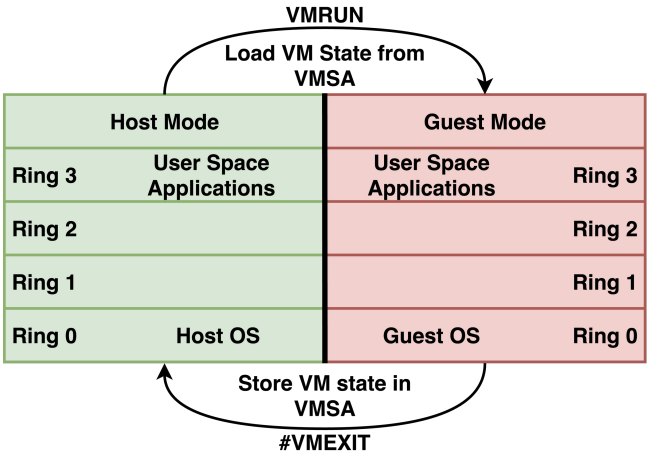


Figure 2.3: Schematic overview of host- and guest mode on CPUs with hardware-accelerated virtualization. Each mode allows access to all privilege levels, but the host mode can configure the hardware such that certain instructions and events trap to the host mode which is dubbed `#VMEXIT`. In addition, the hardware automatically loads and saves certain state when transitioning between host and guest mode.

However, upon performing a memory access in guest mode, the hardware will use the result of walking the guest's `cr3` page tables as input into the NPTs instead of directly performing the memory access. Thus, the addresses managed by the guest are referred to as *Guest Virtual Addresses (GVAs)* and *Guest Physical Addresses (GPAs)*. Translation faults while walking the guest page tables lead to a regular page fault exception that is usually handled by the VM. Translation faults while walking the NPTs lead to a newly added *Nested Page Fault (NPF)* exception that traps to the hypervisor. The NPTs contain the same permission and status bits as regular page tables. If the type of memory access performed by the VM is not allowed in the NPTs a NPF exception is generated. Building on this mechanism, the hypervisor can implement virtualization for memory reads and writes to certain memory pages. This mechanism is, for example, used to emulate memory-mapped device interfaces.

2.2.3 Interrupts and Devices

Interrupt virtualization enables the hypervisor to inject virtual interrupts, exceptions, and traps into the VM. To this end, the hypervisor specifies the event as part of the VCMB before calling `VMRUN`. After entering the VM, the hardware will deliver the corresponding interrupt to the VM. Recent Intel [51, §30.6] and AMD [5, §15.36.21] CPUs offer a more efficient mechanism for interrupt injection that does not require a `#VMEXIT`. To this end, the hardware has built-in support for virtualizing the interrupt controller itself. The virtualized interrupt controller offers the hypervisor an interface for interrupt injection

and presents them to the VM via the corresponding register in the VM's virtualized view on the interrupt controller.

Combined with interrupt interception, and triggering a nested page fault on specific memory accesses, interrupt injection can be used to virtualize internal and external devices. To this end the hypervisor advertises the device to the VM but ensures that all read and write accesses to the interfaces of the device trap to the hypervisor. There are two classes of such interfaces. The first one is configured via specific instructions like `rdmsr`/`wrmsr` or `in`/`out`. The hypervisor can use instruction intercepts to learn the requested state manipulation during writes and to replace the results of reads with the virtualized state. The second interface maps the configuration registers of a device directly into the virtual address space, allowing manipulation via the regular `mov` instruction for memory reads and writes. This is referred to as *Memory Mapped I/O (MMIO)*. To virtualize MMIO interfaces, the hypervisor uses the NPTs to ensure that memory accesses to those pages encounter an NPF exception. By decoding the faulting memory access the hypervisor can infer the data argument for memory writes or replace the results of memory reads.

A typical example of device emulation is the APIC timer, which operating systems use to generate a periodic "tick" for implementing tasks like process scheduling. As such, both the host system and the VM require the APIC timer. However, there is only one physical APIC timer on each CPU core. In addition, allowing the VM to handle the physical APIC timer interrupt would open up the hypervisor to denial of service attacks from the VM. After intercepting a state manipulation request by the VM, the hypervisor programs the physical APIC timer to interrupt at the required point in time, i.e., the minimum of both the requested interrupt from the host and the requested interrupt from the VM. After intercepting the physical interrupt, the hypervisor determines whether the VM's virtualized APIC timer has reached zero. If this is the case, it injects a virtual APIC timer interrupt into the VM.

2.3 Trusted Execution Environments

In this section, we provide general background on TEEs. For an in-depth technical description of the TEEs considered in this thesis, we refer to Section 3.1. This chapter is based on the technical documentation of Intel SGX [31], AMD SEV [4, 5, 64, 65] and Intel TDX [55, 57].

TEEs are a new form of compartmentalizing computer systems. Traditionally, the operating system, or more general software in ring 0 (in host mode), has full control over the hardware and all software. This mode of operation works well if the physical computer

system is owned by the same entity that wants to execute software on the system. With cloud computing, this is no longer the case. The cloud service provider operates a large cluster of servers and rents them to customers. For scalability and usability reasons, customers usually do not rent a whole physical server but only rent access to a VM or other virtualized compute environments that are running on the physical servers. Using traditional access rights management, the cloud service provider has full access to all code and data that is processed on its platform since it operates both the hardware and the hypervisor software. Current encryption techniques can only efficiently protect data at rest and in transit but cannot enable efficient computation on the data while it is encrypted. Thus, customers have to trust the cloud service provider not to spy on their data while they compute on it. TEEs are a CPU feature that promises to lock out the cloud service provider, or more generally privileged software, from accessing a compartment like a VM or a process. Depending on the implementation, the mechanism is enforced by firmware or a dedicated coprocessor that are under the control of the CPU vendor, whom the user still has to trust. This is also referred to as the *(hardware) root of trust*.

Early designs, like ARM TrustZone, only allow creating a single VM-like compartment that is protected from the rest of the system and thus dubbed *Secure World*. Thus, this design is not well suited for the cloud computing use case, which involves multiple, mutually distrusting customers who share the same hardware. Later, on x86, Intel SGX enabled to protect individual processes, which are called *enclaves*. While SGX allows multiple instances, software initially had to be rewritten to work with SGX, and early versions heavily restricted the amount of memory available to enclaves. Later versions greatly improved the memory limit and offered improved software frameworks, called *library OS (libOS)* to ease running existing software in enclaves without the need for modifications. However, SGX enclaves cannot use ring 0 or run a full-fledged OS. In recent years, the industry shifted more towards the VM-scoped TEEs AMD SEV and Intel TDX. Both only require minimal changes to the protected software and integrate very well into the cloud computing ecosystem, which already relies heavily on VMs. In the remainder of this section, we restrict ourselves to Intel SGX, AMD SEV and Intel TDX.

On a technical level, TEEs are based on two mechanisms: Attestation to prove that the initial state of the TEE instance is secure and Isolation at runtime to prevent an attacker from manipulating the TEE instance.

2.3.1 Isolation Mechanisms

For isolation, we differentiate between access rights-based isolation and cryptographic isolation. As depicted in Figure 2.4, the former protects the data against software-level

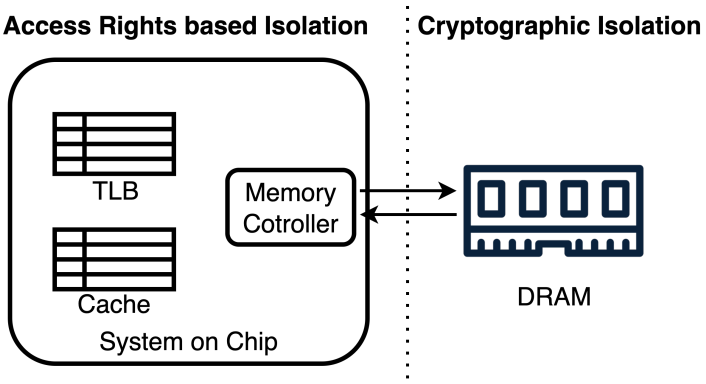


Figure 2.4: Overview of the protection mechanisms employed by TEEs.

attackers while it resides in plaintext on the system-on-chip, while cryptographic isolation protects the data against attackers that try to access the data on the DRAM, which includes both software-level and physical access.

TEEs use memory encryption to protect their data before it gets written to DRAM. To achieve this, they use dedicated hardware memory encryption engines that are part of the MMU. The main intention is to protect the data against physical attackers who have direct access to the DRAM modules. One well-known example of such an attack are cold boot attacks [128] where the attacker exploits that DRAM retains data even after power is cut, especially if cooled to low temperature. This enables attackers to move the DIMM to a different computer to read out its content. The initial, "classic" Intel SGX design used Merkle-Tree based memory encrypted schemes with both integrity and freshness. However, classic Intel SGX has strict limitations on the maximal memory size, as Merkle-Tree based encryption schemes do not scale well to large memory sizes. The more recent scalable Intel SGX, Intel TDX, and AMD SEV switched to lightweight, tweaked block cipher-based encryption schemes to meet the requirement to protect large amounts of memory. The memory encryption keys are managed through the root of trust.

In addition to memory encryption, all x86 TEEs also employ access rights restrictions to prevent software outside the TEE from writing to its memory. SGX and TDX also prohibit reading. As we will learn later in this thesis, implementing read and write restrictions is crucial for the security of TEEs if their memory encryption does not ensure integrity and freshness. Besides restricting direct access to the memory of the TEE, the data must also be protected while it resides in the various caches of the CPU. To this end, TEEs ensure that attackers cannot perform memory accesses that lead to the same cache tag by either ensuring that the attacker cannot access the physical address to begin with or by including additional TEE context identifiers into the cache tag.

Finally, the register content and CPU state of the TEE instance also needs to be protected. To this end, the root of trust is involved in the context switch from the TEE instance to the host, storing the register content and relevant CPU state into the private memory of the TEE. Similarly, when switching back to the TEE, the host has to use an API of the root of trust, which loads the previously stored execution context. For the initial context switch into the TEE, the host system has to use a well-defined entry point, which is ensured via the attestation process described in the next paragraph. As part of the context switch, the root of trust also loads/unloads the correct memory encryption key.

2.3.2 Attestation

The attestation process aims to prove to a (remote) party that their code is running inside a TEE instance and that the host system meets certain security requirements. This step is critical for, e.g., the cloud computing scenario as the customer would otherwise again need to trust the cloud service provider.

To create a new TEE instance the TEE owner (e.g. the customer of a cloud service provider) has to interact with the untrusted host system, which in turn has to use the API provided by the root of trust. First, the TEE owner specifies their initial code image in plaintext and sends it to the host system. The image also specifies the entry point(s) and, for VM-based TEEs, the initial register values. While creating a TEE instance, the root of trust grants the host access to a set of bootstrapping API functions that allow the host to load arbitrary code and data into the TEE instance. The loaded data is protected by the memory encryption and access restriction mechanisms of the TEE. When executing, the TEE performs its code and data accesses with its memory encryption key. Thus, the bootstrapping API is the only mechanism through which the host system can load content into the TEE. To ensure that the host system does not perform any malicious changes to the initial image, the bootstrapping API internally computes a hash value over the loaded data that also takes into account the memory layout of the loaded data. After loading the initial image, the host system informs the root of trust to mark the TEE instance as ready, at which point the host can no longer use the bootstrapping API for this TEE instance. Next, we explain how the TEE owner can verify the correct execution of the bootstrapping procedure through attestation.

There are two variants of attestation: pre- and post-attestation. With *pre-attestation*, the attestation report has to be generated and verified before the TEE instance can be launched. In contrast, with *post-attestation*, the code inside the TEE instance is responsible for requesting the attestation report. Intel SGX, AMD SEV, and Intel TDX all use post-attestation because it provides more flexibility. Thus, we restrict ourselves to this scenario. Since the initial code image must be sent to the untrusted host in plaintext, it should

not contain any secret data like cryptographic keys anyway, eliminating most scenarios where the TEE instance must not be started before attestation has been completed.

With post-attestation, the host starts the TEE at a well-defined entry point. A common use case is that the code in the TEE waits for the TEE owner to engage in the attestation process. To this end, the TEE owner sends a request to the TEE, which may include custom data like a nonce or a public key. Upon receiving this request, the TEE instance calls the root of trust to generate an attestation report. The *attestation report* is a data structure that contains the hash over the initially loaded data, security-relevant system configuration, and software version numbers, as well as the custom data provided by the TEE owner and the TEE instance. In addition, it may also contain custom data from the TEE instance. The attestation report is signed by a key that is private to the root of trust and can be verified by the TEE owner using a public key infrastructure operated by the CPU vendor. Thus, the TEE can send the report to the TEE owner without additional transport encryption. The CPU vendor's public key infrastructure is also what proves to the TEE owner that their code is indeed protected by TEE technology.

A common use case is that the initial code image makes use of the custom data fields in the attestation report to exchange public key pairs with the TEE owner in order to build an encrypted channel. The public/private key pair of the TEE instance is not part of the initial image but generated on-the-fly, to prevent extraction by the untrusted host system. After verifying the attestation report, the TEE owner can use the channel to send additional secrets to the TEE instance, like an encryption key that the TEE can use to load additional code and data or to access to some external service.

3

State of the Art

In this section, we provide a detailed overview of the security research on x86 confidential VMs. To provide a more holistic view, we also discuss the design and certain research results for the process-scoped Intel SGX TEE. We start by explaining the core isolation mechanism of Intel SGX, Intel TDX, and AMD SEV. Next, we present research results categorized by attacks on the memory encryption system (Section 3.2), attacks on the architectural isolation (Section 3.3), and attacks on the microarchitectural isolation (Section 3.4)

3.1 TEE Designs

In this section, we discuss how Intel SGX, Intel TDX, and AMD SEV implement their isolation guarantees. We categorize the isolation mechanisms into the following areas: host access restrictions, page tables management, encryption key management, memory encryption, and interrupt handling. For a general introduction to TEEs we refer to Section 2.3.

With "host access restrictions", we refer to architectural means to prevent privileged software-level attackers from reading and/or writing to the memory of TEEs. The main intention is to counteract certain weaknesses of lightweight but scalable memory encryption schemes.

Another key isolation area is the management of page tables. On the one hand, the untrusted host system should remain in control of memory management. On the other hand, exposing the page tables of the TEE to the attacker undermines the integrity of its memory layout and enables coarse-grained tracking of its memory accesses.

To protect against physical attackers, TEEs use memory encryption. In addition, most TEEs allow the use of a unique memory encryption key for each TEE instance, providing additional protection against cross-TEE data leakage.

Finally, the handling of exceptions and interrupts also plays a vital role for the security of TEEs. Traditional VMs require the virtualization of several interrupts and assume a

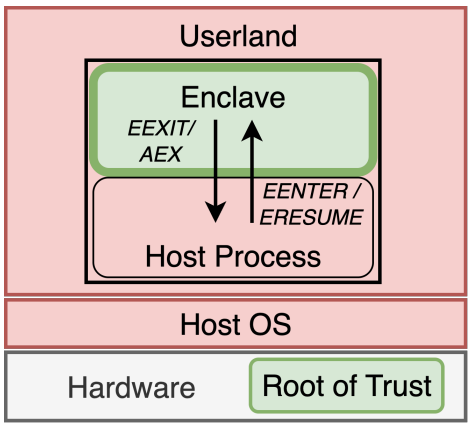


Figure 3.1: Intel SGX is a process-scoped TEE where each instance, called enclave, is tied to a regular process. An instruction set extension allows interacting with enclaves in a remote procedure call manner.

trusted hypervisor. VM-based TEEs still require interrupt virtualization but also need to guard against malicious interrupt injections from the now untrusted hypervisor.

3.1.1 Classic Intel SGX

Intel Software Guard Extensions (SGX) [31, 41] is a process-scoped TEE that was introduced in 2015. To differentiate it from later versions, which have a drastically different design in some key areas, we refer to it as *classic SGX* if we need to distinguish between the two versions. Each Intel SGX instance is a process that consists of an untrusted host part and a trusted enclave part, as shown in Figure 3.1. The host part can enter the enclave at well-defined entry points by using the `EENTER` and `ERESUME` instructions that are part of the instruction set extension that comes with SGX. Similarly, an enclave can either synchronously switch back to the host part via the `EEXIT` instruction or is automatically switched back due to an asynchronous event, like an interrupt. From the host's point of view, interacting with the enclave is similar to performing a remote procedure call. Since SGX is process-based, enclaves cannot use ring 0.

Host Access Restrictions All enclave memory must reside in a memory area named *Enclave Page Cache (EPC)*, which is part of the *Processor Reserved Memory (PRM)*. The PRM is a physically contiguous memory area, which the host has to reserve early during boot by writing its physical address range to two MSRs. Afterward, the host is prevented from writing to this memory range. Reading will return a fixed value instead of the actual content.

Page Table Management For SGX enclaves, the untrusted operating system manages the regular page tables [31, §5.2.3]. However, SGX maintains a kind of shadow page table

called *Enclave Page Cache Metadata (EPCM)* that resides inside the EPC. For each page that should be useable by an enclave, the EPCM stores the identity of the enclave, the expected virtual address, and the expected read, write, and execute permission status of the page. Violations lead to a page fault or general protection fault [51, §35.3] During enclave construction, the operating system has access to instructions that allow it to specify the expected values as mandated by the enclave owners enclave image. In addition, the SGX firmware ensures that each physical page can only be assigned to one enclave. To filter if a given memory access needs to be validated against the information in the EPCM, the hardware/firmware checks if the physical address resides in the protected EPC range.

Encryption Key Management For classic SGX, memory encryption is exclusive to enclaves. There is only one memory encryption key that is used for all EPC memory.

Memory Encryption Classic Intel SGX uses a custom encryption scheme based on *AES Counter Mode (AES-CTR)* in combination with a *Merkle-Tree* to achieve cryptographic integrity and freshness. On each write, a fresh nonce is generated using a 56-bit counter value combined with a value derived from the physical address where the ciphertext is stored. Thus, the ciphertext is bound to a memory location, and the same plaintext never encrypts to the same ciphertext. The integrity and freshness properties are achieved by computing a *Message Authentication Code (MAC)* for each ciphertext block, including the expected nonce. Finally, a Merkle-Tree protects the integrity and freshness of the nonces and MACs. The Merkle-Tree itself is protected by storing its root in secure memory inside the CPU package, which is assumed to be inaccessible to physical attackers.

While providing strong cryptographic guarantees, the Merkle-Tree severely limits the ability to protect large amounts of memory. For each memory read/write, the relevant path in the Merkle-Tree needs to be checked/updated, introducing additional latency as well as computational and memory bandwidth overhead. Since the height of the Merkle-Tree increases with the amount of protected memory, so does the overhead. Finally, storing the Merkle-Tree and MAC values introduces roughly 25% memory overhead. Intel SGX limits the size of the protected memory to 256 MB, which are shared among all enclaves [49]. While there are academic papers [36, 112] that propose design improvements, they were not adopted by industry. Instead, recent TEE designs favor performance over cryptographic guarantees, abandoning Merkle-Tree based designs. Starting with its 12th generation Intel Core Processors, Intel no longer supports classic SGX [50].

Interrupts With SGX, interrupts and exceptions always lead to a context switch to the untrusted host. SGX enclaves do not have the notion of an interrupt or exception handler in the sense of the x86 architecture. However, similar behavior can be implemented via software conventions.

In more detail, when the processor executes in enclave mode, interrupts and exceptions always cause an *Asynchronous Enclave Exit (AEX)*. The AEX allocates a new frame from the *State Save Area (SSA)* of the current enclave thread to securely store the execution state of the enclave before context switching to the host system. The SSA is organized as a stack and resides in secure enclave memory that is inaccessible to the untrusted host. After handling the interrupt/exception, the host system may use the `ERESUME` instruction to transparently reenter the enclave, consuming the topmost SSA frame. Alternatively, the untrusted host system can also make use of the regular *EENTER* instruction to enter the enclave with a fresh state at one of its entry points. This mechanism can be used to run a handler inside the enclave that may inspect and modify the SSA frame saved during the AEX. Afterward, the handler code can cause a synchronous exit to request the untrusted host to resume the enclave via `ERESUME`, consuming the topmost SSA frame, which now has been potentially modified by the handler in the enclave. While conceptually similar, this is not an interrupt handler in the sense of the x86 architecture but rather a software-defined convention.

With the introduction of AEX-Notify [30] in 2023, there exists a configuration option to prevent the untrusted host from transparently resuming the enclave via *ERESUME* after an AEX. This mechanism is used to implement a software-based countermeasure against single-stepping attacks. We refer to Section 3.3.3 for additional details.

3.1.2 Scalable Intel SGX

Starting with the 3rd Generation of Intel Xeon Scalable Processors in 2021, Intel introduced an overhauled version of SGX [63], targeting the protection of large, existing applications in the cloud computing use-case. We refer to this version as *scalable SGX* if we need to differentiate it from the previous version.

The **page table management** and **interrupt** behavior of scalable SGX is the same as for classic SGX.

Memory Encryption Depending on the exact CPU model, scalable SGX uses the newly introduced *Total Memory Encryption (TME)* or *Total Memory Encryption Multi Key (TME-MK)* engine and allows for enclaves sizes up to 512 GB per CPU socket. For memory encryption, both use 128-bit AES with in the *XEX encryption mode with tweak and ciphertext stealing (XTS)*. The tweak values are boot-time static tweak values and derived from the physical address [53], i.e. each 16-byte memory block uses a different tweak. Thus, they do not provide integrity or freshness guarantees.

Host Access Restrictions Like classic SGX, scalable SGX uses the concept of a physically contiguous, fixed-size EPC and prevents code outside of enclaves from accessing EPC

memory. In addition, scalable SGX repurposes one bit of the *Error correction code memory (ECC)*, which is required for using scalable SGX, for usage as a 64-byte granular *ownership bit* [63]. The ownership bit is managed by the hardware as follows: If the CPU operates in enclave mode and writes to a memory location inside the EPC, the ownership bit is set. If the CPU writes to memory while not operating in enclave mode, the ownership bit of the accessed memory location is cleared. Similarly, the CPU enforces the following checks whenever the CPU reads from memory: If the CPU is inside enclave mode and reads from an EPC memory location with the ownership bit cleared, it will terminate, and the SGX feature will be disabled. If the CPU is not in enclave mode and reads from memory with the ownership bit set, it will read a fixed value instead of the actual content. In contrast to the existing physical address-based EPC read/write restrictions, the ownership bit-based approach cannot be circumvented by physical aliasing attacks, which we discuss in Section 3.5.1.

Encryption Key Management Scalable SGX is available on CPUs that only support TME but also on systems that support TME-MK. The main difference between these two memory encryption technologies is that TME only supports one encryption key. We did not find literature that describes the key management. We assume, that with TME, there is an additional key, that is exclusive to SGX. Similarly, we assume that with TME-MK, one of the keys is reserved for use with SGX.

3.1.3 AMD SEV-SNP

AMD Secure Encrypted Virtualization (SEV) [65] is a VM-scoped TEE that was introduced in 2016. TEE-protected VMs are also referred to as *confidential VMs (cVMs)*. In 2017, AMD released the iterative update *SEV Encrypted State (SEV-ES)* [64] and in 2020 the latest version *SEV Secure Nested Paging (SEV-SNP)* [4] was released. If not indicated otherwise, we refer to SEV-SNP in this thesis. As shown in Figure 3.2, SEV builds on AMD's existing virtualization infrastructure. In contrast to Intel SGX and Intel TDX, SEV uses a dedicated coprocessor as its hardware root of trust, which is referred to as the *AMD Secure Processor (SP)*. The SP exposes an API to the hypervisor for creating and managing SEV VMs. Entering and exiting SEV VMs does not require an explicit interaction with the SP but repurposes the existing `VMRUN` instruction and the `#VMEXIT` event from the regular virtualization feature, although enhanced with additional functionality on the microcode level. The strong isolation guarantees require several changes to the hypervisor, especially for handling `#VMEXIT` events, since the VM's register file is no longer accessible to the hypervisor.

Page Table Management Since SEV is VM-scoped, we have to consider two classes of page tables, as explained in Section 2.2.2. The guest page tables are managed by the SEV

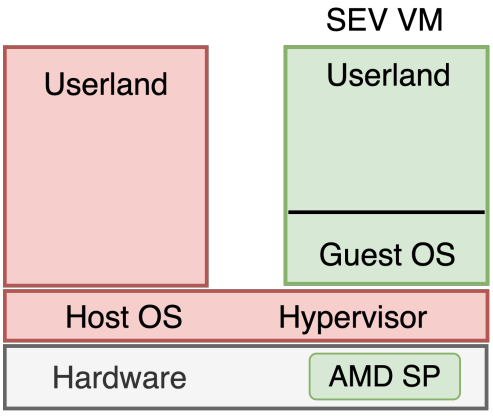


Figure 3.2: AMD SEV VMs integrate into the system architecture similar to regular VMs.

VM and are inaccessible to the untrusted hypervisor. However, the *Nested Page Tables (NPTs)* that translate *Guest Physical Addresses (GPAs)* to *Host Physical Addresses (HPAs)* are under the control of the hypervisor. With SEV-SNP, an additional mechanism called *Reverse Map Table (RMP)* was introduced to ensure that the hypervisor cannot perform arbitrary changes to the GPA to HPA mappings in the NPT [4]. In essence, the RMP acts as a trusted shadow page table that is used to validate the translations provided by the NPT. During boot, the hypervisor has to donate a physically contiguous memory range for the RMP by writing its start- and end address to the `RMP_BASE` and `RMP_END` MSRs[5, §15.36.3]. After initializing the SEV-SNP subsystem, the hypervisor can no longer directly access this memory range.

The RMP is a linear table that is indexed by the HPA. For each HPA, the RMP stores several attributes. The most important one is the *Assigned* attribute, which indicates if this page belongs to a SEV VM, the host system, or firmware. We will refer to this as *guest-owned*, *hypervisor-owned* and *firmware-owned*. For guest-owned pages, the RMP additionally stores the expected GPA, the validation status, and the ASID, which maps the entry to a concrete VM instance. For firmware-owned pages, the hypervisor also has to set the *Immutable* bit in the RMP entry. Once set, this bit prevents the hypervisor from updating the entry in the future without coordinating with the AMD SP.

The basic workflow for managing memory for VMs via the RMP works as follows: To assign a memory page to a VM, the hypervisor uses the `rmpupdate` instruction to change the following fields: *Assigned*, expected GPA, and the ASID of the assigned VM. However, `rmpupdate` will always reset the *Validated* field. Whenever the VM triggers a page table walk, the hardware page table walker uses the result of the GPA to HPA translation from the untrusted NPT to consult the RMP. If the memory page is not assigned to the currently executing VM, the hardware aborts the access and generates a RMP page fault. Otherwise, the *Validated* bit is consulted. If the bit is unset, the hardware injects a `#VC`

into the VM, informing the VM about the address for which the validation failed. If the VM considers the GPA benign, it can use the `pvalidate` instruction to set the *Validated* bit and continue with the memory access. Whenever the *Validated* bit is set upon a RMP check initiated by a memory access from a SEV-VM, the hardware will verify that the GPA used as input to the RMP matches the GPA stored in the RMP. Otherwise, the access is aborted with a RMP page fault. In summary, the hypervisor can still make arbitrary changes to the mapping in the NPT, but the RMP check ensures that the VM is notified about changed mappings. Note, that the access rights and status bits in the NPT are not validated by the RMP.

Memory Encryption On SEV-SNP enabled CPUs, 128-bit AES with the *Xor-Encrypt-Xor (XEX)* mode is used for memory encryption. The tweak values use boot-time generated randomness and are derived from the physical address, i.e., each 16-byte memory block uses a different tweak. Each SEV VM, has its own encryption key. AES-XEX does not offer any integrity or freshness guarantees. The only exception to this is how the *Virtual Machine Save Area (VMSA)* is handled. This data structure stores the VM's state, e.g., its register file, during context switches from the VM to the hypervisor. Since SEV-ES, the VMSA is protected with an integrity check value during context switches. For SEV-SNP, since the "MilanPI-SP3 1.0.0.5" firmware version, the data in the VMSA gets masked with a nonce before encryption/after decryption to prevent ciphertext side-channel attacks [3] which we discuss in Section 3.2.2. In contrast to the two Intel SGX variants, SEV does not impose any artificial limitations on the amount of memory that can be used for the TEE instances, and the memory pages do not have to reside in a physically contiguous pool. One bit of the physical address space is repurposed to determine if a memory access should get routed through the memory encryption engine. This mechanism allows software to access the raw ciphertexts of encrypted memory locations.

Host Access Restrictions Prior to SEV-SNP, privileged software attackers had read and write access to the encrypted memory of SEV VMs. With SEV-SNP, the RMP prevents software-level attackers from writing the memory of the SEV VM. To this end, the hardware performs a RMP check during any page table walks related to write accesses by the host. The RMP verifies that the accessed physical address is marked as hypervisor-owned in the RMP. Otherwise, the access is aborted with a page fault.

Encryption Key Management SEV uses a different memory encryption key for each VM and also supports encrypting the memory of the host system. To associate memory encryption keys to a VM, SEV repurposes the *Address Space Identifier (ASID)* [6, 65]. The ASID is an identifier that is part of all memory accesses on the system (VM and host). It extends the TLB tag and thus alleviates the need to flush the whole TLB when context switching to the hypervisor or another VM. The ASID of the hypervisor/host system is

hardwired to 0. The hypervisors can arbitrarily change the ASIDs of VMs by changing the corresponding field in the *Virtual Machine Control Block (VMCB)*.

With SEV, the ASID is also used to tag instruction and data cache entries to secure the data of SEV VMs while it resides unencrypted in these caches. When creating a SEV VM, the hypervisor must activate an unused ASID by interacting with the SP via the `SNP_ACTIVATE` command. Afterward, the SP creates a new memory encryption key and associates it with the given ASID. If the hypervisor wants to recycle an ASID, it has to first deregister it via another call to the SP, which will ensure any potential cache entries with that ASID are flushed. After allocating an ASID, the hypervisor can use the launch API of the SP to set up the encrypted initial image of the new SEV VM. Finally, when launching the VM, the hypervisor specifies the VM's ASID as part of its VMCB. At this point, the hypervisor can arbitrarily manipulate the ASID value in the VMCB, e.g., swap the ASID of an attacker-controlled VM with the ASID of a victim VM. However, AMD argues [65] that after such a swap, both VMs would immediately crash since they would decrypt their code with a different key, which is extremely unlikely to result in valid instructions. Li et al. [78], show that this assumption only partially holds prior to SEV-SNP. We refer to Section 3.4.3 for more details.

Interrupts SEV VMs inherit the existing interrupt and exception virtualization features from AMD's implementation of hardware-accelerated virtualization, which is called *Secure Virtual Machine (SVM)* [5, §15]. To virtualize these events, SVM allows configuring that certain interrupts or exceptions transparently lead to a `#VMEXIT`. Furthermore, SVM has a feature to allow the hypervisor to inject virtual interrupts or exceptions into the VM.

With SEV-SNP, AMD introduced additional security features for interrupt injections [5, §15.36.16], [4]. *Restricted Injection* limits the legacy SVM interrupt injection interface and only allows the injection of one fixed "doorbell" interrupt. After processing the doorbell interrupt, the VM has to engage in a communication protocol with the hypervisor to receive the actual payload interrupt. *Alternate Injection* is supposed to work in tandem with Restricted Injection and allows hiding the communication step of Restricted Injection from the OS inside the VM by making use of AMD's *Virtual Machine Privilege Levels (VMPLs)* feature. Among other things, VMPLs enable the implementation of a trusted shim inside the SEV VM that engages in the communication protocol with the hypervisor required for Restricted Injection. The Alternate Injection feature enables the trusted shim to inject the fetched interrupts into the VM's OS, that is running in a lower VMPL, using the legacy interface. Thus, the VM's OS does not need to be aware of the changes introduced by the Restricted Injection feature.

In order to emulate certain interrupts or exceptions, the hypervisor needs access to specific register values of the VM. However, since SEV-ES, the register state of the VM is

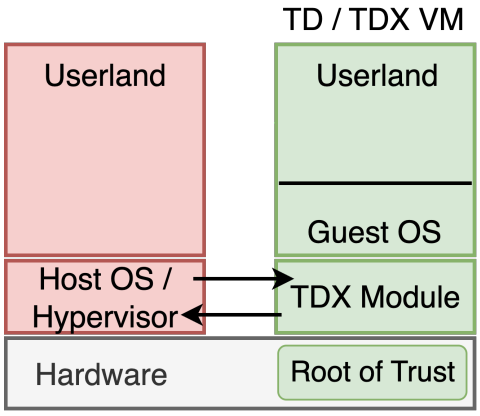


Figure 3.3: Overview of the Intel TDX architecture. The TDX module acts as a trusted layer between the TD and the untrusted hypervisor. Architecturally, it is isolated from the hypervisor via the newly added SEAM CPU mode.

encrypted during context switches and thus inaccessible to the hypervisor. To resolve this issue, SEV uses a shared memory protocol between the VM and the hypervisor to share information in such instances.

3.1.4 Intel TDX

Intel Trust Domain Extensions (TDX) [55] is the most recent TEE by Intel that offers the creation of confidential VMs that Intel also refers to as *Trust Domains (TDs)*. It was announced in 2021, and CPUs with TDX became publicly available with the 5th generation Xeon Scalable CPU series in December 2023. TDX builds on a newly added CPU mode called *Secure Arbitration Mode (SEAM)*. Figure 3.3 shows an overview of the architecture. The SEAM mode is split into two sub-modes: *VMX root* and *VMX non-root*, which mirror the existing *Virtual Machine Extensions (VMX)* modes used for virtualization. The TDX module runs in SEAM VMX root mode, and creating TDs is exclusive to this mode. It acts as a trusted software layer between the TDs and the untrusted hypervisor. However, the TDX module is not intended to replace the hypervisor and delegates most resource management tasks to it. The TDs themselves run in VMX non-root mode. These dedicated CPU modes act as an anchor to enforce several policies.

Page Table Management Since TDX is a VM-based TEE, memory accesses by a TD require both *Guest Virtual Address (GVA)* to *Guest Physical Address (GPA)* and GPA to *Host Physical Address (HPA)* translation. The former is performed by the regular page tables, which are managed and located inside the TD. The latter is performed by the *Extended Page Tables (EPT)*, which are Intel's pendant to the NPTs on AMD systems. With TDX, the EPTs are split into two sub-tables: One for shared pages and one for private pages.

Whether a page is treated as shared or private depends on a repurposed bit of the GPA, i.e., the software inside the TD can control this status [57, §9]. For shared pages, the hardware page table walker will use the shared EPTs, which are directly managed by the hypervisor. For private pages, the hardware page table walker will use the private EPTs, which are managed by the TDX module and thus protected from direct access by the hypervisor. Instead, the hypervisor has to use an API exposed by the TDX module to modify the private EPTs. The TDX module ensures that the hypervisor cannot add the same host physical address as a private page to two TDs. If private pages are added while the TD is running, the TD is informed about the change. The TDX modules' API for the private EPTs does not allow direct modifications of the access rights and status bits after creating the EPT entry. However, it still offers the functionality to temporarily block an address translation, causing a regular page fault when the TD tries to access it.

Memory Encryption TDX uses *Total Memory Encryption-Multi-Key (TME-MK)*, the successor of TME. This feature was also briefly advertised as *Multi Key Total Memory Encryption (MKTME)*. Like TME, TME-MK uses 128-bit AES-XTS with a tweak value that is derived from boot-time randomness and the physical address and is thus unique for each 16-byte memory chunk. Each TD uses its own encryption key and AES-XTS does not offer integrity protection or freshness. However, TDX implements two additional mechanisms to protect the integrity: logical integrity protection and cryptographic integrity protection. For both, TDX uses the ECC bits in the DRAM to store security-critical metadata. The *logical integrity mode* is only an access prevention mechanism and is implemented via a single status bit stored in the repurposed ECC bits. It tracks if the untrusted host tried to write to a memory location that is used by a TD and will subsequently prevent the memory location from being used by a TD. In addition, if the hypervisor tries to read a memory location used by a TD, it will always read a fixed pattern. The upcoming paragraph on access restrictions explains this in more detail. The *cryptographic integrity mode* additionally protects the data via a SHA-3 based MAC with 28-bit tags that are also stored in repurposed ECC bits. MAC verification errors permanently disable a memory location to prevent brute force attacks on the short MAC. Like SEV, TDX does not artificially limit the memory size of TDs, nor does it require the memory pages to reside in a physically contiguous, boot-time allocated memory pool.

Key Management TME-MK repurposes parts of the upper physical address bits as a *KeyID*. The `PCONFIG` instruction is used to associate a KeyID with an AES key. With TDX, the KeyID range is split into a private and a public range at boot time. Only the SEAM CPU mode, exclusive to the TDX module and TDs, is allowed to use `PCONFIG` with private KeyIDs. In addition, trying to perform a memory access with a private KeyID outside of SEAM mode will result in a page fault.

Access Restrictions TDX imposes restrictions on read and write operations [57, §9]. To implement the read and write restrictions, TDX repurposes one ECC bit to store additional metadata for each 64-byte memory block. This feature is also referred to as logical integrity protection. For each block, it stores the *TD-bit* to indicate whether this block belongs to a TD Initially, the TD-bit is zero. When the CPU performs a memory access using a private KeyID, i.e., it is operating in SEAM mode, the TD-bit is set for the accessed memory location. Otherwise, the TD-bit gets cleared. When the CPU performs a read with a private KeyID and the memory location does not have the TD-bit set, a fixed value is returned, and an exception is generated. The TDX module handles the exception and will eventually terminate the TD. When the CPU operates outside of SEAM mode and tries to read a memory location that has the TD-bit set, a fixed value is returned, but no exception is generated. As the ECC bits are fetched on each memory access anyway, this mechanism does not reduce the memory bandwidth of the system. While the host can write to memory locations with the TD bit set, this will trigger a "poison" mechanism that leads to a fatal exception the next time a TD tries to access it.

Interrupts Similar to AMD SEV, Intel TDX is by default interrupt aware, making use of Intel's hardware accelerated *Virtual Machine Extensions (VMX)* feature [55],[57, §11.10]. We refer to the interrupt paragraph of the SEV description in Section 3.1.3 for a short introduction to the concept of interrupt virtualization. Contrary to SEV, the Intel hardware suppresses the injection of interrupt vectors 0 to 31 into TDs. This range includes the *Non Maskable Interrupt (NMI)* and most exceptions. To inject NMIs, the hypervisor has to use the API of the TDX module. For maskable interrupts, which are not in the range of 0 to 31, TDX uses the concept of posted interrupts, which builds on Intel's APIC virtualization. The hypervisor has access to a data structure called *Posted Interrupt Descriptor* where it can queue interrupts that it wants to inject into the TD. To notify the TD about pending interrupts, the hypervisor can either use an API function of the TDX module or send an inter-processor interrupt with a dedicated vector to the core that executes the TD. Both methods act as a doorbell. Upon receiving the doorbell, the hardware takes the interrupts queued in the posted interrupt descriptor and injects them into the TD by manipulating the corresponding registers of the virtualized APIC.

3.2 Attacks on Memory Encryption

In this section, we discuss attacks that exploit weaknesses in the memory encryption system of TEEs. First, we summarize attacks that actively manipulate ciphertexts or copy them to new memory locations, exploiting the lack of integrity protection and freshness observed in recent TEE designs. Such attacks were primarily shown against SEV versions prior to SEV-SNP, which did not prevent software-level attackers from writing to the

memory of the TEE. Afterward, we discuss passive attacks that only require read access to leak information from ciphertext patterns exhibited by deterministic memory encryption modes. From the considered TEEs, only SEV-SNP allows software-level attackers to read the encrypted memory of the TEE. However, the software-based Rowhammer attack or an advanced physical attacker could potentially overcome software access rights restrictions, expanding the scope of these attacks to scalable Intel SGX and Intel TDX.

3.2.1 Ciphertext Manipulation

Before SEV-SNP, software-level attackers could modify the ciphertext of the protected VMs. Such manipulations have been explored in two forms. Buhren et al. [14] use ciphertext manipulation as a primitive to introduce faults into computations. Another line of work [33, 79, 123], introduces *ciphertext moving attacks*, where the attacker copies ciphertext from one location to another to build encryption/decryption oracles or to cause unexpected behavior in software using the modified data. In contrast to fault attacks, the ciphertext does not decrypt to randomized plaintext with ciphertext moving attacks

Exploiting Software Write Access Buhren et al. [14] show that attackers can exploit memory encryption without integrity protection to perform classical data fault attacks. Simply flipping a bit in the ciphertext will lead to a randomized plaintext. However, without integrity protection, the encryption system cannot detect this manipulation. They demonstrate a well-known fault attack against RSA CRT. The authors conduct their experiments on *AMD Secure Memory Encryption (SME)*, which uses the same encryption engine as SEV but focuses on protecting the host memory and does not provide any TEE capabilities. However, their attack would also be possible with SEV. With SEV-SNP, software-level attackers can no longer write to the memory of protected VMs, mitigating the attack. The same is true for scalable Intel SGX and Intel TDX. In principle, Rowhammer attacks could be used to manipulate the ciphertext without the need for architectural software-level access. While scalable SGX, Intel TDX, and AMD SEV-SNP are only available on server CPUs, that require ECC memory, Cojocar et al. [29] show that ECC memory is not sufficient to prevent Rowhammer. Nonetheless, we are not aware of any published Rowhammer attacks on these systems. In contrast to scalable SGX and SEV-SNP, classic Intel SGX and Intel TDX could detect the manipulated ciphertext due to their integrity features. For TDX, this is an optional feature.

Ciphertext Moving Attacks Ciphertext moving attacks have been used in two scenarios to build encryption/decryption oracles on AMD SEV versions prior to SEV-SNP. Both require the attacker to know the tweak values of the encryption system. Du et al. [33]

and Wilke et al. [123] analyzed the encryption mode of SEV in detail and show that for each physical address bit $i \geq 4$ there is a 16-byte tweak vector t_i. The tweak value for a physical address p is computed as

$$T(p) \coloneqq \bigoplus_{i=4}^{n-1} \mathrm{bit}(p,i) \cdot t_i,$$

where $\mathrm{bit}(p,i)$ represents its i-th least significant bit. For first-generation EPYC CPUs, they reverse engineer the exact tweak values, exploiting that the tweaks are static and have low entropy. However, starting from the second-generation EPYC CPUs, the tweak values have high entropy and are regenerated at boot-time, making the brute-force-based reverse engineering approach infeasible. Finally, the AES-XEX encryption of a message m at address p is defined as $\mathrm{Enc}_K(m,p) \coloneqq \mathrm{AES}_K\left(m \oplus T(p)\right) \oplus T(p)$. Some first-generation EPYC CPUs used AES-XE instead of AES-XEX. For brevity, we only describe the attacks for the AES-XEX mode.

For the first attack scenario, demonstrated by Du et al. [33] and by Li et al. [79], the attacker requires access to a mechanism that copies data into/out of the VM to build encryption/decryption oracles. Let us assume the VM runs a network reachable service that accepts some form of payload data that temporarily gets stored at memory address A inside the VM. The attacker's goal is to store the encryption of plaintext m at address B inside the VM. First, the attacker sends the payload $m \oplus T(B) \oplus T(A)$, which the service running inside the VM will store at address A. Thus, the ciphertext C_A at address A is $\mathrm{AES}_K\left(m \oplus T(B) \oplus T(A) \oplus T(A)\right) \oplus T(A)$ Using the page fault controlled-channel, the authors infer when the data has been written to A. Next, they copy the ciphertext from A to B and XOR $T(A) \oplus T(B)$ to the ciphertext C_A. Let C'_A be the manipulated ciphertext. When the VM accesses address B, the ciphertext will decrypt to

$$\begin{aligned}
&\mathrm{Dec}_K(C'_A, B) \\
&= \mathrm{AES_K}^{-1}\left(m \oplus T(B) \oplus T(A) \oplus T(A) \oplus T(B)\right) \oplus T(A) \oplus T(A) \oplus T(B) \oplus T(B) \\
&= \mathrm{AES_K}^{-1}\left(m\right) \\
&= m
\end{aligned}$$

Using a service that copies data out of the VM, an attacker can build a decryption oracle in a similar manner. Li et al. [79] show that DMA bounce buffers can be used as a mechanism to copy data into/out of the VM, which only requires that the VM uses some (virtual) device that performs DMA operations.

The second attack scenario, demonstrated by Wilke et al. [123], does not require any

network reachable service inside the VM nor DMA bounce buffers. Instead, they use a known plaintext to ciphertext dictionary as their source for ciphertexts. To manipulate the plaintext at a given memory location, the attacker searches through their dictionary to find a ciphertext that decrypts to the desired plaintext at the targeted memory location. The authors show that about 8 MB of known plaintext is sufficient to reliably control 2 bytes of a 16-byte ciphertext block. To obtain the dictionary, they assume that the Linux kernel binary used by the VM is known to the attacker and show how to locate it in memory at runtime. Using this primitive, they demonstrate how to inject small code gadgets into the VM. They show how to manipulate the control flow by inserting `ret` instructions into functions before their actual end. Furthermore, they show how to construct a more complex code gadget, which allows to bootstrap a primitive that grants control over all 16 plaintext bytes of a ciphertext block. The core idea is that the injected program constructs the desired value inside a CPU register and then pushes the register to its stack memory to encrypt it with the VM's key.

In summary, ciphertext moving attacks require that the attacker can read and write the encrypted memory of the TEE and that the encryption system does not have freshness. More specifically, they require that a ciphertext created at one memory location can be decrypted at another memory location without yielding randomized plaintext. From the TEEs considered in this thesis, the encryption systems of scalable Intel SGX, AMD SEV, Intel TDX, and potentially ARM CCA all have this property. However, only SEV allows privileged software-level attackers to read and write the raw ciphertext of protected VMs. Starting with SEV-SNP, ciphertext moving attacks are mitigated by two mechanisms. First, SEV-SNP disallows writing to the VM's memory, preventing copying ciphertexts from one location to another. Second, CPUs with SEV-SNP use high entropy tweak values that cannot be reverse-engineered. In combination with the AES-XEX encryption, copying a ciphertext to a new memory location would result in randomized plaintext.

3.2.2 Ciphertext Side-Channels

Ciphertext side-channels are a class of attacks against TEEs that exploit deterministic memory encryption. They were introduced by Li et al. [80] in 2021 and generalized by Li et al. [76]. In [80], Li et al. introduce ciphertext side-channel attacks in the context of the *VM Save Area (VMSA)* of AMD SEV. The VMSA is part of the hardware-accelerated virtualization interface. Among other information, it stores the register values of the VM upon a context switch to the hypervisor. Throughout the lifetime of a VM, the VMSA is stored at a fixed physical address. Thus, the encryption system always uses the same tweak value, resulting in a one-to-one mapping between plaintexts and ciphertexts. Li et al. build a dictionary of ciphertext to plaintext mappings to leak the value of certain registers of the VM from the ciphertext of the VMSA. To obtain such a mapping, the

authors assume that the attacker knows the code of the UEFI firmware executed by the VM. For confidential VMs, this is usually the case as it is part of the public initial code image. They locate several points in the code of the UEFI where *(i)* a context switch to the hypervisor is triggered and *(ii)* the value of certain registers is known to the attacker. For *(ii)*, they use hard-coded values as well as values that follow a simple logic, i.e., are incremented in a loop. They apply pattern matching to the VM's exit behavior to detect which exits belong to the previously analyzed instances. As a register value only requires 8 bytes, the VMSA stores two register values in one 16-byte ciphertext block for most registers, making dictionary building challenging. However, specific registers, like `rax`, are stored in isolation. Nonetheless, Li et al. show that they can learn enough mappings to build exploits. For example, they show how to learn the mappings for all values from 0 to 128 for `rax`. Based on this, they demonstrate end-to-end attacks against the RSA and ECDSA implementation in OpenSSL. In both cases, they exploit that there is a loop over the key that uses a function call to fetch small chunks of the key during each iteration. The function returns the chunk of the key in the `rax` register. Using the page fault side-channel, Li et al. trigger a VM exit when this function returns and infer the value of `rax` by using the dictionary to leak the secret key. AMD released a firmware patch to mitigate ciphertext side-channel attacks against the VMSA [3]. The patch retrofits nondeterministic encryption to the VMSA page by applying a randomly generated mask to the plaintext before encryption/after decryption.

In their follow-up paper, Li et al. [76] show that the Linux kernel itself, as well as commonly used cryptographic libraries, also exhibit memory access patterns that are exploitable via ciphertext side-channel attacks. Even worse, the memory addresses used by these applications are not known in advance. Thus, AMD's mitigation is incomplete and cannot easily be extended to solve all instances of ciphertext side-channel leakages. The authors show that attacks similar to the one on the VMSA can also be applied in the context of user space to kernel space context switches in Linux, during which the kernel has to store the register values of the user space program. Immediately after the context switch, Linux stores the register values on the aptly named entry stack, whose location is fixed per CPU. Before scheduling another user space thread, Linux stores the register values to a data structure fixed during the lifetime of the user space thread. In addition, they show that the stack memory usage of the ECDSA implementation of OpenSSL is also vulnerable to dictionary attacks. Furthermore, they show another leakage pattern dubbed collision attacks, where it is sufficient to observe whether or not the ciphertext changes without the need to recover the actual plaintext. This pattern is commonly exhibited by the *constant time swap* operation used in constant time implementations of cryptographic algorithms. The authors also discuss the applicability of ciphertext side-channel attacks to other TEEs. They state that classic Intel SGX is not vulnerable because it uses nondeterministic memory encryption and that scalable Intel SGX and

Intel TDX are, in principle also vulnerable to ciphertext side-channel attacks. However, their access rights management prevents software-level attackers from directly accessing the ciphertext, preventing straightforward exploitation.

Deng et al. [32] developed an automated tool that scans cryptographic code for access patterns vulnerable to ciphertext side-channel attacks. Their approach is based on dynamic taint tracking and static symbolic execution. They report over 200 vulnerable code locations in the RSA, ECDSA, and ECDH implementations of OpenSSL, MbedTLS, and WolfSSL. However, ciphertext side-channel attacks have applications beyond cryptographic code. Yuan et al. [130] show that ciphertext side-channels can leak the weights of deep neural network models. Their attack only needs to observe a single inference to reconstruct models that achieve between 77% and 97% accuracy, depending on the software stack used for the observed inference. In another paper, Yuan et al. [129] explore the recovery of image and video inputs processed by neural networks, including transformers. They successfully reconstruct images and videos that are visually indistinguishable from the original inputs. Moreover, they use the recovered inputs to train their own model and use it to generate adversarial inputs for the original model.

Countermeasures Nondeterministic memory encryption completely eliminates the ciphertext side-channel. However, the most recent x86 TEEs have abandoned nondeterministic memory encryption technologies due to the considerable overhead when protecting large amounts of memory. While the architectural access restrictions implemented by Intel's TEEs prevent straightforward exploitation, they could be overcome by the Rambleed [71] subvariant of Rowhammer or by advanced physical attackers. Starting from revision 1.55, the SEV ABI specification mentions a "ciphertext hiding" feature, indicating that AMD is working on a mitigation [6]. However, no additional details are provided. Several works instead explore countermeasures that retrofit nondeterministic encryption on the software level. Li et al. discuss 3 basic approaches: interleaving data with a nonce, masking data before writing it to memory, and changing the memory location of the data on each write to ensure that a different tweak value is used. They provide a proof of concept mitigation against the attack on the userland to kernel context switch mechanism by changing the memory location where the register file is stored on each context switch. The following works assume such a protection for register values. Wichelmann et al. [118] explore masking-based countermeasures for user space applications and provide an automated toolchain to protect the secret key and all derived values in cryptographic code. To this end, they use dynamic taint tracking and instrumentation to identify all memory locations containing secret data and static binary instrumentation to rewrite the corresponding memory accesses to use a secure masking gadget. Their evaluation of several cryptographic implementations shows an average overhead of factor 2.4 up to 17.5, depending on the security parameters. Wichelmann

et al. [119] aim to provide holistic protection against various leakage sources, including the ciphertext side-channel. Their approach divides the code into uniform chunks and requires the current code block and the currently processed data to be loaded into a scratchpad. Both code and data are loaded through a software *Oblivious RAM (ORAM)* construction, and the location of the scratchpads is changed after each load to prevent ciphertext side-channels. In addition, data is protected using the interleaving strategy. They divide each 16-byte data chunk into 8 bytes of payload data and an 8-byte nonce that is updated on each write. Due to the holistic protection that includes cache attacks and single-stepping, they observe tremendous overheads ranging from factor 169 for a small example up to factor 100,019 the ECDH implementation in mbedTLS.

3.3 Attacks on Architectural Isolation

Trusted Execution Environments need to uphold their security guarantees against a privileged attacker that can make full use of the vast amount of architectural features and mechanisms of modern x86 CPUs. A long line of work shows how (partial) access to page tables, control over high-frequency interrupt sources, interrupt virtualization, and power management features can be used to subvert the security guarantees of TEEs. In this section, we discuss the resulting attacks, focusing on single-stepping and the recent single-stepping mitigation attempts.

3.3.1 Page Fault Controlled-Channel

Page fault controlled-channel attacks enable the attacker to monitor the memory accesses of TEEs with 4 KB granularity. They were introduced by Xu et al. [126] and since then been used by nearly all attack papers on SGX [17, 83, 91, 95], SEV [44, 80, 93, 104, 123] and TDX [122]. While Xu et al. [126] focus on showing that certain applications leak text documents or image outlines via the page fault controlled-channel, most of the attacks cited above use them to synchronize with the state of the application before applying a subsequent attack.

For SGX and SEV-SNP, the attacker leverages their direct access to the page tables to reduce the access permissions to a page. On the next access of the given type by the TEE, a page fault exception is generated, which leads to a context switch to the attacker-controlled operating system/hypervisor. Both SGX and SEV-SNP report the access type but mask the lowest 12 bits of the faulting address.

On TDX, the hypervisor does not have direct access to the EPTs that perform the GPA to HPA translation for private TD memory. Thus, it can no longer force page faults by

manipulating the access rights bits. However, the TDX module, which mediates the hypervisor's access to the EPTs, still exposes two dedicated API functions to temporarily prevent the TD from accessing a page. The first function [56, §3.6.2.1] is a generic block that triggers a page fault exception on any type of access. However, the TDX module sanitizes the information passed to the hypervisor, hiding the access type and the lowest 12 bits of the faulting address. Wilke et al. [122], show that even without the access type, the information is still sufficient to track the execution state of applications inside TDX. The second function [56, §5.3.3] allows only blocking write accesses and is intended to reduce the downtime phase of live migration.

Disabling or severely restricting the attacker's ability to trigger page faults would drastically improve the security of TEEs. For TDX, write blocking for migration could be made an optional feature. The primary use case for the regular block function cited in the TDX manuals are TLB shootdowns [56, §9.7]. TLB shootdowns are required when removing or changing an existing mapping in the EPTs. The hypervisor needs to ensure that all vCPUs of a TD atomically switch to the updated EPT entry and do not use stale TLB entries. The ability to block a translation alleviates the need to stop the execution on all vCPUs synchronously. For TDs, updating EPT entries during runtime is already a complex operation due to the need for the TD to accept mapping changes in order to prevent malicious address layout manipulations. Thus, switching back to a less efficient synchronous mechanism might be a worthwhile tradeoff if it removes the need for a page block mechanism.

For AMD SEV, the use cases for page blocking are similar. To prevent an attacker from manipulating the access rights in the NPT, the *Reverse Map Table (RMP)* mechanism could be extended to additionally verify the value of the access rights bits during page table walks and terminate the VM upon mismatches. In addition, the RMP would need to be consulted on every page table walk, not just for write operations.

For both Intel SGX variants, the EPCM already stores the expected read, write, and execute permissions for each page. However, currently, violations trigger page fault exceptions and thus leak the memory access to the untrusted operating system. Instead, for high-security demands, the enclave could simply terminate upon page faults.

3.3.2 Page Remapping

Morbitzer et al. [92, 93, 94] show that, prior to SEV-SNP, a privileged attacker can use their control over the *Nested Page Tables (NPT)*, to reorder the address space of a SEV VM with page granularity. Thus, they can, e.g., change which data an application reads at a certain address.

As explained in Section 3.1, the NPT performs the *Guest Physical Address (GPA)* to *Host Physical Address (HPA)* translation and is under the control of the untrusted hypervisor. In the *SEVered* attack [92, 93], Morbitzer et al. show how to construct a decryption oracle. To this end, they assume a network-reachable service in the VM that returns a resource to the attacker. While serving the request, the service copies the resource into a network package. For the attack, they change the GPA to HPA mapping of the GPA that contains the resource and thereby change the data that the network service will read and eventually send to the attacker. Since the NPT can only be used to remap with 4 KB page granularity, the alignment and size of the resource dictate which parts of a page can be leaked. They demonstrate the attack for Apache, Nginx, and OpenSSH. To infer the GPA of the targeted resource, they start by repeatedly querying the service and use the page fault controlled-channel to observe the memory accesses performed by the VM. Using the resulting GPAs as a candidate set, they query different resources of the service to reveal which GPA belongs to the targeted resource. In their follow-up work [92], Morbitzer et al. show how to use the page fault controlled-channel to infer the GPA of cryptographic secrets like TLS or disk encryption keys and subsequently leak them using the SEVered attack. In [94] Morbitzer et al.show that the basic mechanism of the SEVered attack can also be used to inject arbitrary code into the VM.

With SEV-SNP, NPT remapping attacks are mitigated by the RMP mechanism, which can detect if the hypervisor changed the GPA to HPA mapping and subsequently prohibits the memory access. For Intel TDX, the hypervisor has to use an API exposed by the TDX module to access the EPT, Intel's pendant to the NPT. The TDX module enforces certain restrictions and notifies the TD if a mapping gets changed. For both SGX variants, the attacker has direct access to the page tables, but the trusted EPCM is used to verify that the virtual to physical mapping has not been changed since the enclave was created.

3.3.3 Single-Stepping

Single-stepping attacks try to interrupt the targeted TEE as frequently as possible, ideally after every instruction. This enables the attacker to obtain fine-granular information about the execution state and control flow of the TEE. Several works [42, 75, 89] explored the idea on Intel SGX, but failed to achieve single instruction granularity. In SGX-Step, Bulck et al. [20] demonstrate reliable, instruction-granular single-stepping. Similarly, Li et al. [80] first applied the idea to SEV-SNP and Wilke et al. [124] demonstrate reliable single-stepping. Intel TDX was released with a built-in single-stepping countermeasure, but Wilke et al. [122] reveal flaws that again enable them to single-step TDX. Combined with the page fault controlled-channel, single-stepping enables fine granular tracking of the execution state of the TEE. At the time of writing, there are countermeasures for SGX [30] and TDX [57, §17.3].

Mechanism The core primitive for single-stepping attacks is the same on SGX, SEV-SNP, and TDX. Constable et al. [30], provide a detailed root-cause analysis. The attacker exploits that external interrupts lead to an exit from the TEE, returning control to the attacker. The attacker programs the APIC timer to trigger an external interrupt shortly after the TEE was entered. To achieve single-stepping, the interrupt must arrive during the execution of the first instruction inside the TEE. The exact timing does not matter, as interrupts are only processed at instruction boundaries. However, without additional measures, the interrupt sometimes arrives too early or too late, causing zero-steps or multi-steps. To avoid multi-stepping, the attacker needs to artificially prolong the time required to execute the first instruction. To this end, they flush the page that contains the first instruction from the CPU caches, including the TLB. For SGX, it also has been shown that resetting the *Accessed* bit in the page tables prolongs the time required for the page table walk. For TDX, the attacker additionally needs to overcome the detection heuristic of the built-in countermeasure. We explain this in more detail in the countermeasures paragraph.

Prolonging the execution time of the first instruction does not help to avoid zero-steps. However, in contrast to multi-steps, zero-steps do not lead to a loss in precision. Thus, it is sufficient if they can be detected and filtered. Section 3.3.4 discusses attacks based on zero-steps. To filter zero-steps on SGX, Bulck et al. [20] check if the *Accessed* bit in the page tables is again set to one after being reset before the stepping attempt. On SEV, Wilke et al. [124] exploit the lack of isolation in the performance counters, using the "retired instructions" event to infer the exact step size. For TDX, neither of the two methods work, as the attacker does not have access to the GPA to HPA page tables, and, according to the manuals, the performance counters are isolated. Instead, Wilke et al. [122] use a cache attack to check if the code page of the first instruction has been accessed.

Attacks Single-stepping has been used for various attacks. **Instruction counting** attacks use single-stepping together with the page fault controlled-channel to reveal the control flow of the TEE with instruction granularity [20, 91, 122, 124]. This allows to, e.g., exploit non-constant time behavior in cryptographic libraries in order to leak encryption keys [91, 108]. Gast et al. [37] exploit the lack of performance counter isolation on SEV-SNP in combination with single-stepping to create even more fine-grained traces of the VM's execution. **Interrupt Latency** attacks analyze the time between entering and exiting the TEE while single-stepping. Bulcket al. [19] demonstrated that the timing reveals information about the executed instruction. For example, generating a random number with `rdrand` takes significantly more cycles than an `add` instruction. Distinguishing, e.g., different ALU instructions from each other proves difficult due to the similar timings. Furthermore, the authors show that data-dependent timing of, e.g., the *div* instructions can be distinguished. However, in most instances, distinguishing instruction types or

data operands requires a high amount of measurements. Finally, single-stepping is also used as a generic **Amplifier** by simply improving the synchronization of the attacker with the victim code to reduce noise [11, 18, 48, 61, 98, 102, 108, 131].

Countermeasures Since the release of SGX-Step in 2017, single-stepping has been used in the vast majority of attacks on SGX. Several academic papers try to detect single-stepping heuristically by monitoring interrupt rates or performance monitoring counters [25, 73, 96]. By design, these heuristics can generate false positives in noisy real-world settings, which have significant consequences, as the proposed solutions typically terminate the enclave when violations are detected.

Shih et al. [107] use the deprecated *Transactional Synchronization Extensions (TSX)* to redirect all enclave interruptions to a handler inside the SGX enclave, hiding the actual location at which the interrupt occurred from the OS. While they focus on the page fault controlled-channel, their use of TSX should also prevent single-stepping. Lang et al. [72] dynamically relocate data at runtime to obscure the signal of any memory access-related side-channel, including single-stepping. They use the deprecated TSX extension to enforce that their relocation primitives are executed without interruption, especially without single-stepping.

Chen et al. [24] propose OS modifications to execute an enclave without any interruption and use a checking mechanism inside the enclave to verify this invariant. To this end, they place a canary in the *State Save Area (SSA)* inside the enclave, which would get overwritten by the *Asynchronous Enclave Exit (AEX)* mechanism, which handles context switches due to interrupts. The enclave software periodically checks the canary and terminates execution if a violation is detected. The requirement for completely uninterrupted execution of the enclave restricts the ability of the OS to manage resources efficiently. However, it could be an interesting tradeoff for enclaves with high-security demands.

Wichelmann et al. [119] ensure that applications do not leak information while being single-stepped. To this end, they apply a code transformation that slices the program into uniform code blocks and also ensures that code fetches and data accesses are routed through a software-based *Oblivious RAM (ORAM)* construction, which obscures all page access patterns. While very generic, their approach comes with very high overhead. None of these countermeasures have found widespread adoption.

With *AEX-Notify* [30], Constable et al., in 2023, released the first countermeasure for Intel SGX, that targets the root cause of single-stepping. The paper is joint work between Intel and academic researchers. The core idea of the countermeasure is to prevent the attacker from slowing down the execution of the first instruction. They show that without this slowdown, the APIC timer and other interrupt sources are not precise enough to cause

reliable singe-stepping. Undoing the slowdown is performed by the AEX-Notify handler, a software routine running inside the enclave that needs to be executed every time the enclave is interrupted, i.e., after every AEX. The AEX-Notify handler prefetches the code and data of the next payload instruction to undo any artificial slowdowns. It is split into a regular section that mainly computes the addresses that need to be prefetched and a small critical section that does the actual prefetching. If the critical section is interrupted, the AEX-Notify handler will get re-executed from the start. Otherwise, the AEX-Notify handler continues to execute at the interrupted location since it does not leak any information via its control flow.

Using the existing SGX ABI, it is not possible to ensure that the AEX-notify handler is executed after every AEX because the untrusted host can resume the enclave via the `ERESUME`, which makes the interruption transparent for the enclave. In more detail, the AEX stores the current state of the enclave, and the `ERESEUME` instruction transparently reloads the state as part of the context switch into the enclave. We refer to Section 3.1.1 for more details.

The AEX-Notify firmware update introduces the `AEXNOTIFY` flag, which is under the control of the enclave. If the flag is set at the time of the AEX, a subsequent `ERESUME` does no longer resume the enclave from the stored state but instead behaves like the regular `EENTER` instruction, allowing the enclave to enforce the execution of the AEX-Notify handler. In addition, the firmware update adds the `EDECCSSA` instruction that allows the enclave to consume the state stored during the AEX. This enables the AEX-Notify handler to eventually resume execution at the point of interruption without relying on the untrusted host.

Intel's most recent TEE, Intel TDX, was released with a built-in countermeasure against single-stepping attacks. The countermeasure builds on the fact that with TDX, the hypervisor has to go through the trusted TDX module to run a TD. Similarly, the CPU returns to the TDX module instead of the untrusted hypervisor for all TD exits. For most exit reasons, the TDX module eventually returns control to the hypervisor. To prevent single-stepping, the TDX module applies a heuristic to decide whether the current TD exit is due to a single-stepping attack. To this end, the number of retired instructions and the time elapsed since the preceding TD entry is measured. If either more than two instructions have been executed or at least 4096 cycles have passed since TD entry, the exit is classified as benign. Otherwise, the single-stepping prevention mode gets activated. The prevention mode ensures that the TD executes an additional, randomized number of instructions before the TDX module hands back control to the hypervisor. To this end, the TDX module invokes the TD multiple times, executing a single instruction on each invocation.

Wilke et al. [122] show that by downclocking the core of the TD and TDX module, they can ensure that the detection heuristic always classifies the time since the last TD entry as sufficient. As a result, the prevention mode is never activated. Without the prevention mode, TDX can be single-stepped using techniques similar to SGX-Step [20] and SEV-Step [124]. The downclocking itself can be performed from software, e.g., via the cpufreq driver of the Linux kernel. Besides single-stepping, Wilke et al. also show a second attack primitive, which they call *StumbleStepping*. For StumbleStepping, they exploit that the prevention mode inherently leaks the number of instructions it executes in the TD via a cache side-channel, thus still allowing instruction counting attacks.

Intel addressed the single-stepping attack from Wilke et al. with TDX module version 1.5.06 [54] by changing the detection heuristic so that it no longer relies on the time between TD entry and exit. However, there still is a fallback to the old heuristic for certain TD configurations. Intel will not provide mitigations against the StumbleStepping instruction counting primitive. Instead, Intel requires developers to ensure their code is not vulnerable to instruction counting, by strictly adhering to the constant-time programming paradigm.

For AMD SEV, there are no countermeasures against single-stepping attacks. Due to the lack of a trusted layer between the SEV VM and the hypervisor the TDX countermeasure cannot easily be adopted. However, it might be possible to implement countermeasures similar to AEX-Notify via a slight firmware change that ensures that the SEV VM cannot be transparently resumed but always executes an adapted version of the AEX handler after each interruption.

3.3.4 Zero-Stepping

Zero-stepping attacks try to repeatedly measure the effect of an instruction without requiring to execute the targeted TEE instance multiple times. They can be used against instructions inside the TEE as well as against the context switch into the TEE itself. In both cases, the attacker ensures that no instruction inside the TEE retires, enabling infinite repeated measurements. This allows the exploitation of side-channels with a very low signal-to-noise ratio. There are two main approaches for zero-stepping attacks: timer-based and page fault-based.

Timer-based zero-stepping is closely related to single-stepping attacks. However, instead of trying to avoid zero-steps, the attacker actively chooses a short APIC timer interval to ensure that they only zero-step, i.e., only the instruction for the context switch is retired. For SGX, Chowdhuryy et al. [28] show that the `ERESUME` instruction, which is usually used to context switch into the enclave, has a serializing effect. Thus, no payload instruction of the enclave is in-flight if it is already interrupted during the `ERESUME` instruction.

However, as ERESUME restores the register state of the enclave by loading it from the SSA frame, the instruction is an interesting target in itself. Schwarz et al. [106] use timer-based zero-stepping to leak register values of SGX enclaves via an MDS attack. On AMD SEV, Wang et al. [116] explore timer-based zero stepping against the VMRUN instruction, which performs the context switch into SEV VMs. However, they only performed experiments for the plain SEV version. They use the MSR_CORE_ENERGY_STAT MSR to observe the current power consumption of the CPU. Surprisingly, they are able to differentiate not only between data operands but also between different instructions inside the SEV VM. This indicates that although only the VMRUN instruction is executed architecturally, at least one instruction inside the SEV VM is already in-flight. This is especially interesting because intuitively, there should be a dependency between the VMRUN instructions and the first instruction inside the VM, as VMRUN restores the VM's register file and address space. The authors do not further analyze the root-cause of the observed power leakage.

For page fault-based zero-stepping, the attacker manipulates the page tables (c.f. Section 3.3.1) to ensure that the targeted instruction inside the TEE always experiences a page fault and is thus never retired. In contrast to timer-based zero-stepping, the context switch into the TEE completes. Although the targeted instruction never retires, other instructions that follow the targeted instruction (in program order) may also be already in-flight due to out-of-order execution unless they have a dependency on the targeted instruction. For SGX, several works [17, 90, 106, 109] use this mechanism to repeatedly observe the context switch or the first instructions in the enclave via various side-channels.

Lipp et al. [83] use zero-stepping against Intel SGX in combination with the RAPL interface, which allows measuring the power consumption of the CPU. They show that an attacker can differentiate between different instructions and instruction operands. The paper does not state which zero-stepping variant is used, but given the results from [28], only page fault-based zero stepping should allow observing effects of instructions inside the enclave. Chowdhuryy et al. [28] also explore using the deprecated Intel TSX instruction set extension to repeatedly observe the effect of an instruction without retiring it.

Countermeasures For Intel SGX, the AEX-Notify single-stepping countermeasure from [30] et al. provides partial protection against zero-stepping. We refer to Section 3.3.3 for an introduction to AEX-Notify. Since the interrupt handler provided by AEX-Notify prefetches the code and data operands of the first payload instruction, it should not be possible to use the first payload instruction as an anchor for page fault-based zero-stepping. The AEX-Notify handler is split into a regular part and a small, critical part. If the handler is interrupted during the critical part, it restarts from the beginning on the next entry, whereas the regular part continues at the interrupted location. Thus, an

attacker could zero-step individual instructions in the regular part or repeatedly trigger the re-execution of the whole interrupt handler. Similar to the *ERESUME* instruction, the interrupt handler must also restore the register file. This could potentially be exploited in zero-stepping attacks. However, further research is required to determine how the AEX-Notify handler interacts with zero-stepping attacks.

Intel TDX was released with a built-in countermeasure against page fault-based zero-stepping. The TDX module tracks the number of page faults without architectural progress. After reaching a certain threshold, it refuses to enter the TD until the hypervisor corrects the mappings. The single-stepping countermeasure of TDX also prevents zero-stepping, as the triggered prevention mode executes at least one instruction in the TD. SEV does not have countermeasures against any zero-stepping variant.

3.3.5 Interrupt Injection

VM-based TEEs like SEV or TDX allow the untrusted hypervisor to inject interrupts and exceptions into the VM to facilitate virtualization. For SGX, popular SDK and libOS implementations enable a similar concept. We refer to Section 3.1 for additional background information on interrupt injection mechanisms. Schlüter et al. [104, 105] and Sridhara et al. [111] show that by injecting unexpected interrupts, the attacker can use side effects of the interrupt/exception handlers to manipulate the execution state of the TEE or to trigger malicious behavior in an application.

In *Heckler*, Schlüter et al. [105], analyze the security of AMD SEV-SNP, Intel TDX, and ARM CCA against interrupt/exception injection attacks, finding vulnerabilities on SEV-SNP and TDX. Both TEEs allow the hypervisor to inject the `0x80` syscall interrupt. As the syscall handler stores the exit code in the `rax` register, triggering a syscall at arbitrary code locations inside the VM allows the attacker to manipulate the register state of the TEE. Based on this primitive, they show how to bypass authentication checks in both OpenSSH and the sudo program, enabling an attacker to open a shell and gain root privileges.

Furthermore, Schlüter et al. show that on SEV-SNP, the hypervisor can also inject exceptions like "Divide by 0" (`0x0`) into SEV VMs, while TDX blocks direct injection of vectors `0x0` to `0x31`, which covers standard exceptions and NMIs. Injecting exceptions enables an attacker to trigger signal handlers of user space applications, again allowing to manipulate the execution state through side effects of these handlers.

For SEV-SNP, combining the recently introduced *Restricted Injection* and *Alternate Injection* modes, in principle, allows to implement a trusted shim layer inside the VM that could filter and sanity check injected interrupts. However, the authors state that this is not yet

implemented. Intel TDX prevents the hypervisor from directly injecting interrupt/exception vectors in the range of `0x0` to `0x31`. Instead, the hypervisor has to use an API of the trusted TDX module, which applies filtering. The authors state that none of the vectors allowed by TDX have handlers with exploitable side effects. To mitigate their attack, the authors suggest adding syscalls (`0x80`) to the list of filtered interrupt vectors for TDX. Orthogonal to the proposed changes the authors also show how the `0x80` handler of the guest kernel can implement filtering by checking whether the interrupt was externally injected. However, they state that prohibiting externally injected `0x80` breaks some legacy use cases.

In *WeSee*, Schlüter et al. [104] show similar attacks but focus on the SEV-SNP specific `#VC` interrupt. This interrupt is part of the paravirtualized communication interface between the hypervisor and the VM that is used to, e.g., share required register state to enable the emulation of instructions, like `cpuid`. It is usually triggered by the hardware upon execution of said instructions, to instruct the guest kernel to share state with the hypervisor. As such, the authors argue that there is no valid use case requiring the hypervisor to inject a `#VC` externally. Thus, they suggest blocking the injection at the firmware/hardware level. As a workaround, the authors also discuss strategies for the guest Linux kernel to validate the reason for the `#VC` exception but ultimately deem this approach as too error-prone due to the code complexity of the `#VC` handler.

Sridhara et al. [111] analyze the security of Intel SGX against interrupt/exception injection attacks. As discussed in Section 3.1.1, SGX is process-scoped and does not have a hardware-based mechanism for injecting events like interrupts or exceptions. However, many SGX SDKs and libOS implementations provide a software interface to facilitate the injection of exception-like events to, e.g., handle a division by zero inside the enclave or to implement inter-thread or inter-enclave communication. Sridhara et al. find multiple instances where malicious injections can manipulate the behavior of SGX enclaves. Similar to the previously discussed attacks, most of the found vulnerabilities can be mitigated via additional filtering logic in the corresponding handler to verify that the injected exception is benign.

3.3.6 Software-based Undervolting

It is well-known that computations on a CPU can be faulted by changing its operating voltage or frequency and by exposing it to light impulses or electromagnetic impulses [47]. For many cryptographic algorithms, like RSA or AES, faults allow key-recovery attacks. However, fault attacks have mostly been explored in the context of a physical attacker that uses external gear, like a laser, to induce faults. To be more energy efficient, modern CPUs allow *Dynamic Voltage and Frequency Scaling (DVFS)* in order to save power and prevent

overheating. Several works explore using (undocumented) software interfaces related to DVFS to induce faults without the need for physical access by either overclocking [114] or undervolting [26, 66, 95, 99, 100] the CPU.

Tang et al. [114] explore software-induced fault attacks on a Nexus 6 phone that uses the ARMv7 architecture. They show that a privileged attacker can induce faults by overclocking individual CPU cores via memory-mapped configuration registers. The authors demonstrate attacks against the ARM TrustZone TEE, extracting cryptographic keys from protected applications and loading self-signed applications into the TEE. Qiu et al. [99] show similar attacks against ARM TrustZone by undervolting the CPU.

On Intel CPUs, several works [66, 95, 99, 100] use an undocumented MSR that is part of the DVFS interface to undervolt the CPU in order to induce faults. They demonstrate fault attacks against classic Intel SGX enclaves, including a fault attack on AES-NI, Intel's hardware-accelerated AES implementation. To mitigate these attacks, Intel deployed a microcode update that disables the exploited undervolting interface and includes the disablement status in the SGX attestation report. In *PMFault*, Chen et al. [26] exploit software vulnerabilities in the *Baseboard Management Controller (BMC)* of a Supermicro X11SSL-CF board, to undervolt the CPU by sending commands directly to the voltage regulators via the *Power Management Bus (PMBus)*. The BMC provides remote parties with management capabilities similar to those achieved by physical access and is commonly integrated into server mainboards. They demonstrate similar attacks against classic Intel SGX as the previously discussed papers that used an MSR for undervolting. However, Intel's microcode update does not prevent their attack primitive. Furthermore, the software version of the remote management interface is not part of the Intel SGX attestation, making it hard for a remote party to assess the security of the system. While recent Supermicro mainboards mitigated the vulnerabilities in the BMC that the authors exploited to get direct access to the PMBus, the authors argue that malicious PCI-E devices might also be used to carry out their attack since they are also connected to the PMBus. Finally, their attack challenges the security architecture of Intel SGX, which claims not to place any assumptions on the BIOS or the remote management interface.

Orthogonal to Intel's mitigations that disable the interface required for undervolting, Kogler et al. [69] explore software countermeasures to detect faults. While they prototype their solution with Intel SGX, their technique could potentially transfer to other TEEs. Their solution is based on two insights. The first insight is that MSR-based undervolting on Intel CPUs requires at least 57.7 µs and thus affects a large slide of instructions. The second insight is that the multiplication instruction (`mul`) is the most susceptible to faults if the CPU is undervolted. Based on this, they construct a `mul` based gadget that checks if it has encountered a fault. Next, they develop an extension for the LLVM compiler to automatically insert their gadget into the code during compilation using the "density"

of the gadget as a security parameter. Their analysis shows that placing the gadget after every one to two payload instructions detects 99% of the attacks and causes an overhead of 148.8%. If their gadget detects a fault, they suggest terminating the enclave. To prevent an attacker from repeatedly restarting the enclave, which could eventually allow them to overcome the probabilistic detection mechanism, they suggest using the EPID attestation mode of SGX to throttle the restart rate of an enclave on a given CPU. This approach requires a trusted third party. In principle, the mitigation from Kogler et al. could also apply to other undervolting or faulting mechanisms like *PMFault* or the hardware-based undervolting mechanisms that will be presented in Section 3.5.2. However, further analysis regarding the assumptions on the temporal resolution that an attacker can achieve is required.

3.3.7 Other

In this section, we briefly summarize relevant architectural attacks that do not fit into the previous categories.

Zhang et al. [131], show that on AMD systems, the attacker can use the `invd` instruction to drop memory writes of SEV VMs. On x86, writing a value usually only changes the entry in the cache instead of immediately writing to DRAM due to performance reasons. The corresponding cache line is marked as "dirty". Under normal operation, flushing or evicting a dirty cache line will write the value to DRAM to persist the change. The privileged `invd` instruction allows marking all cache entries as invalid/free without triggering a write-back to DRAM, effectively discarding the value of all dirty cache lines. The authors show that this instruction can be used while SEV VMs run on the system. An attacker can interrupt the VM after a security-critical write operation and issue an `invd` instruction to discard the newly written value, causing subsequent read operations to return the old value instead. However, `invd` does not allow for a fine-grained discard as it marks the whole L1 and L2 cache of the executing core (including the sibling core) and the last level cache of the current core complex invalid. The authors work around this issue by using eviction sets to trigger a write-back for all but the targeted cache set before issuing the `invd` instruction. AMD mitigated the issue with a microcode update. On Intel CPUs, the `invd` instruction leads to a general protection fault if either SGX or TDX is enabled.

Buhren et al. [16] uncover a flaw in the boot procedure of the *AMD Secure Processor (SP)* that allows them load and execute arbitrary code on the SP. The SP is a security coprocessor that is part of the system-on-chip of an AMD EPYC CPU and forms the hardware root of trust for SEV. Thus, executing attacker-controlled code on the SP completely undermines all security guarantees of SEV. The authors successfully extract

a CPU-specific key that is used to prove the authenticity in, e.g., remote attestation, allowing them to set up a "fake" SEV environment while still providing valid attestation reports. They demonstrate their attack on Zen 1 EPYC CPUs. Furthermore, they show that there is no rollback prevention for the SP firmware on Zen 1 CPUs, preventing AMD from mitigating the exploited flaw via a simple software update. They state that the only way to prevent a rollback is to revoke the keys that signed the vulnerable firmware versions. AMD EPYC CPUs from Zen 2 onward are not affected by this issue.

Several works [44, 117] exploit that before SEV-ES, the register state of SEV VMs was not encrypted during context switches to the hypervisor. This allows attackers to easily read and manipulate the register values of the VM.

Wilke et al. [125] show that prior to SEV-SNP, the attestation mechanism is permutation agnostic. An attacker can reorder the initial VM image with 16-byte granularity without changing the attestation measurement. This undermines all trust in the attestation of SEV and SEV-ES VMs. They exploit this ability to create malicious code gadgets and show how they can be used to gain arbitrary code execution capabilities. Their exploit is partly enabled by the fact the VM's UEFI image is not hardened against attacks since these are infeasible or out of scope when running on physical hardware or a regular VM. This again highlights the importance of thoroughly examining legacy VM software components before deploying them in a confidential VM environment, which operates under a significantly different threat model. Independent of their research, AMD mitigated the reordering issues in the attestation with SEV-SNP.

Radev et al. [101] and Hetzelt et al. [45] investigate the security of interfaces between the VM and the hypervisor, highlighting that numerous commonly used Linux software components and drivers fail to implement adequate input sanitization, which is critical when dealing with an untrusted hypervisor. Hetzelt et al. [45] show a fuzzer, to automatically discover such vulnerabilities in Linux device drivers.

3.4 Attacks on Microarchitectural Isolation

After a TEE loads data from DRAM, it resides in plaintext inside CPU. To prevent unauthorized access, TEEs introduce additional tagging mechanisms to enable isolation in microarchitectural structures like CPU caches or the TLB. In this section, we discuss the impact of cache attacks and Meltdown/Spectre/MDS attacks on TEEs.

3.4.1 Cache Attacks

Cache attacks are a well-known side-channel that can leak memory access patterns. They have been used to leak information across processes [97, 127] or even co-located VMs [7, 85] in the context of an unprivileged attacker.

Several works demonstrate cache attacks against classic Intel SGX[12, 89, 108]. The strong attacker model allows minimizing system noise and using single-stepping to achieve the optimal temporal resolution. As a mitigation to [17] Intel added a mechanism to flush the L1 data cache during context switches [58, 108].

On AMD SEV and Intel TDX, the integration of multi-key memory encryption into the system architecture has strong implications for cache attacks that require shared memory, like Flush+Reload. Both AMD's and Intel's memory encryption systems embed metadata into the physical address, which alters the cache tag. On Intel TDX, each TD has its own *Total Memory Encryption Multi Key (TME-MK)* KeyID which is encoded in the upper bits of the physical address and thus also part of the cache tag [52]. On AMD SEV, each memory access is associated with an *Address Space Identifier (ASID)*. In addition, one bit of the (guest) physical address, dubbed *C-bit*, is repurposed to select whether or not the memory access is routed through the memory encryption engine. Both, ASID and C-bit are part of the cache tag [78].

For both TDX and SEV, architectural isolation mechanisms prevent a privileged attacker from performing memory accesses with a KeyID/ASID used by a *confidential VM (cVM)* instance. This implies that the attacker cannot get a cache hit on data accessed by the cVM, nor can they use the `clflush` instruction to flush it. While SEV offers an additional mechanism that allows the hypervisor to flush the memory of the cVM with page granularity [5, §15.34.9], the inability to get a cache hit remains. Thus, the attacker cannot perform *Flush+Reload* cache attacks.

The TME-MK memory encryption engine used by TDX comes with an additional cache coherency mechanism that re-enables the hypervisor to flush the memory of TDs with cache line granularity [59]. Since the KeyID is part of the cache tag, it would allow the same memory location to reside in the cache multiple times under different KeyIDs. The coherency mechanism prevents such aliasing by flushing the existing entry. Using this mechanism, an attacker can perform Flush+Reload-style attacks against TDX by using an aliased memory access as both, the flush and the reload primitive. Wilke et al. [122] observe that the access times after flushing via an aliased memory access are even higher than regular DRAM access times. This is most likely because reloading again causes a flush. Scalable SGX is available on processors that support only TME but also on newer processors that support TME-MK. We conjecture that on TME-MK-enabled systems, one KeyID is reserved for SGX, leading to the same side effects as for TDX.

More recent AMD CPUs support a similar cache coherency mechanism [5, §15.34.9]. Wilke et al. [124], tried implementing the Load+Reload [82] cache attack on such a CPU and observed RAM access times instead of the expected L2 access times, indicating that there could be a similar, implicit flush as with TDX. However, in their experiments all cache lines of the accessed page show high access times, not only the targeted one.

Cache attack primitives that do not rely on shared memory, like *Prime+Probe*, are unaffected by the KeyID/ASID side effects described in the preceding paragraphs. Wilke et al. [124], show a Prime+Probe cache attack against AMD SEV.

3.4.2 Meltdown, MDS and Spectre Attacks

In this section, we briefly introduce Meltdown, *Microarchitectural Data Sampling (MDS)*, and Spectre attacks and discuss their applicability to TEEs.

Meltdown [84] mainly affects Intel CPUs and exploits that their microarchitecture already performs memory accesses before all access rights checks have been completed. While illegal memory accesses are eventually detected and aborted, the microarchitectural state partly persists and can be used to leak information, e.g., via the cache. Meltdown has also been shown on certain ARM CPUs [84], while AMD CPUs are reported to be immune [22]. *MDS* attacks [21, 90, 103, 106] leak data from microarchitectural buffers, exposing data handled within the CPU, especially in the context of *Simultaneous Multithreading (SMT)*.

Spectre [68] attacks exploit the speculative execution capabilities of modern CPU designs. Speculative execution enhances performance by allowing the CPU to predict the outcome of control flow transfers, enabling the CPU to speculatively continue executing rather than waiting until the target of the control flow transfer has been resolved. With Spectre, the attacker mistrains the corresponding prediction units of the CPU to influence the path of speculative execution. While instructions and memory accesses executed due to misspeculation are never architecturally committed, they often leave a footprint in the microarchitectural state, allowing to leak data. Cache state is the most common leakage mechanism. There are several Spectre variants. In *Spectre v1* [68], mistrained speculation for conditional direct branches is used to perform out-of-bounds memory accesses. Attackers can use this to speculatively access arbitrary values in the victim's address space. To leak speculatively accessed values, most attacks require code locations that perform a memory access based on the result of the speculative out-of-bounds read shortly after the branch. Afterward, the attacker can probe the cache to infer information about the accessed memory location. *Spectre v2* [68] targets indirect branches and mistrains the corresponding *Branch Target Buffer (BTB)* to speculatively execute attacker-controlled code paths. This equips the attacker with capabilities similar to return-oriented programming or jump-oriented programming, enabling them to stitch together several code snippets to

induce malicious behavior. *Spectre-RSB* [70, 86] is similar to Spectre v2 but targets return instructions, which use the *Return Stack Buffer (RSB)* for predicting the target address. On Intel CPUs from the Skylake microarchitecture upwards, the prediction for return instructions may use the BTB as a fallback for deeply nested function call chains, as the RSB has a limited size. In the following, we will treat Spectre-RSB as a subvariant of Spectre v2. *Spectre v4* [110] exploits speculation in the context of store-to-load forwarding such that a misspredicted dependency between a load and a prior store leads the CPU to access stale data.

Intel SGX For Spectre v1, SGX requires the enclave developer to deploy software countermeasures like inserting `lfence` instructions or using USLH [132]. For Spectre v2, Intel CPUs with the *Indirect Branch Restricted Speculation (IBRS)* feature implicitly ensure that the prediction of indirect branches for code running inside an enclave is isolated from the untrusted part of the system. This behavior is independent of the software-controlled enablement of IBRS [60]. Chen et al. [23] demonstrate that before IBRS, Spectre v2 attacks against SGX were possible. Several MDS attacks have been demonstrated against SGX [90, 103, 106]. Intel addressed these issues with microcode updates and added silicon changes to subsequent CPU generations [103, 106]. Recent Intel CPUs are not vulnerable to Meltdown and MDS attacks. The mitigation status of a CPU can be queried via the *IA32_ARCH_CAPABILITIES* MSR.

Intel TDX The TDX module employs several countermeasures to protect TDs against Spectre attacks [1]. To prevent Spectre v1 attacks, the TDX module uses the `lfence` instruction as a barrier before processing attacker-controlled input. For Spectre v2, the TDX module ensures that *Enhanced Indirect Branch Restricted Speculation (eIBRS)* is enabled, which isolates the SEAM non-root CPU mode, at which TDs execute, from the remaining system in terms of indirect branch prediction. To mitigate cross TD Spectre attacks, the TDX module issues an *Indirect Branch Predictor Barrier (IBPB)* whenever a TD is moved to a new logical processor. IBPB acts as a speculation barrier, preventing any preceding instructions from influencing the branch prediction of subsequent instructions. Furthermore, the TDX module also employs software countermeasures to protect against Spectre-BHB. To protect against Spectre v4, the TDX module ensures that *Speculative Store Bypass Disable (SSBD)* is activated. Finally, the TDX module checks the *IA32_ARCH_CAPABILITIES* MSR to ensure that it executes on hardware that is not vulnerable to Meltdown and MDS attacks.

AMD SEV With SEV-SNP, AMD offers additional protection against Spectre v2 for the VM [5, §15.36.17]. The *Branch Target Buffer Isolation* feature ensures that no branch target entries originating from code running outside the SEV VM are used while the SEV VM executes. In addition, the *Indirect Branch Prediction Barrier (IBPB) on Entry* feature

of SEV-SNP can be used to ensure that the hardware always inserts an IBPB barrier before entering the VM. Finally, *SMT Protection* can be enabled to ensure that while the vCPU thread of a SEV VM is actively running, the sibling SMT thread cannot be used. For Spectre v1, the SEV VM has to use software countermeasures like inserting `lfence` instructions or USLH [132]. Similar to Intel CPUs, AMD CPUs also support *Speculative Store Bypass Disable (SSBD)* to prevent Spectre v4. However, we did not find documentation on whether SEV can ensure the enablement. As previously mentioned, AMD processors are reported to be immune against Meltdown. Similarly, most MDS attacks do not affect AMD but the SEV documentation [4, 5, 6] does not state any dedicated checks or countermeasures.

3.4.3 Other

In *CROSSLINE*, Liet al. [78] analyze the ASID based isolation of AMD SEV and show two attacks for plain SEV and SEV-ES. With SEV, the ASID selects the VM's memory encryption key and tags the cache and the TLB to ensure proper isolation. We refer to Section 3.1.3 for more background information. The key idea of the two attacks is to use the hypervisor's control over the ASIDs to swap the ASIDs of a victim VM and an attacker-controlled VM. AMD states that this should not lead to security issues, as the VMs should crash when executing their next instruction since they decrypt their code with a different key, leading them to execute randomized values as instruction [65]. The first CROSSLINE attack uses the hardware page table walker to build a decryption primitive. To his end, they remap the GPA of the VM's root page table in the NPT and mark all other pages as not present. Afterward, they change the ASID of the attacker VM to the victim VM's ASID. On the next memory access of the attacker VM, the hardware page walker still tries to interpret the content of the remapped page as a page table entry. If the content of the page adheres to the page table format, the page table walker will generate a page fault since all pages previously have been marked as inaccessible in the NPT. The page fault reveals the decrypted content of the supposed page table entry, using the victim VM's key due to the ASID manipulation. No information is revealed if the content of the page does not adhere to the page table entry format. The author's analysis shows that, excluding actual page tables, only a small amount of memory satisfies the page table entry format. With SEV-SNP, the required NPT remapping is prevented by the *Reverse Map Table (RMP)*. While the authors describe that the attack requires a victim and an attacker VM, whose ASIDs are swapped, I suspect that this attack should also work by simply modifying the NPT of the victim VM. The second CROSSLINE attack shows that the attacker VM can reuse TLB entries of the victim to execute a few instructions. However, to leak data to the attacker, they require access to the VM's register file, which is prevented with SEV-ES.

In a subsequent publication, Li et al. [81] perform further analysis on the TLB management of SEV prior to SNP. According to their experiments, the TLB is tagged with the ASID, the logical core partition (isolating the two logical cores of a physical core), the C-bit, and the "VM mode", i.e., "plain", "SEV", "SEV-ES" and "SEV-SNP". They discover that it is the untrusted hypervisor's responsibility to flush the TLB when vCPUs are moved to a new logical core, allowing an attacker to suppress TLB flushes. Li et al. show two attack scenarios, which both assume that the attacker controls a process running in the VM. In the first scenario, they assume that the VM has at least two vCPUs, one executing the victim process and one the attacker process. The hypervisor time-slices both on the same logical core. For the attack, the victim is interrupted at virtual address VA_a using, e.g., the page fault controlled-channel. Next, the hypervisor flushes the TLB and runs the attacker process, using its own vCPU, to load a malicious gadget at VA_a, which, e.g., overwrites a crucial register value. Afterward, the victim vCPU is again scheduled but without flushing the TLB. Thus, it reuses the poisoned TLB entry, executing the attacker-controlled instruction. In the second scenario, the authors assume only one vCPU, where the TLB state would usually be destroyed by the `cr3` update inside the VM that is part of context switching between the victim and the attacker process. Li et al. demonstrate that this can be circumvented by temporarily moving the vCPU to a new logical core before performing the `cr3` update. They also highlight that applications like Nginx, Apache, and the lesser-known Dropbear SSH server spawn a new process for each connection, that reuses the same address space layout as the parent. OpenSSH also spawns such child processes but explicitly rerandomizes the address space. They argue that such a child process can act as the attacker-controlled process assumed in the previous attacks. In an attack case study, they demonstrate that an attacker can bypass the password check of Dropbear SSH, using the TLB entries of a concurrently executing benign login. With SEV-SNP, the hardware enforces correct TLB flushing, preventing this class of attacks [77].

3.5 Physical Access Attacks

In addition to protecting against privileged software-level attackers, most TEEs also consider physical attackers, at least for the DRAM interface. The TDX white paper states that the attacker model includes "CSP insiders (e.g., technician) that can attempt a HW attack to extract Cloud Tenant's secrets or spoof Tenants data/memory by hooking into system interfaces (DDR bus)." [55]. SGX considers the "...DRAM and the bus connecting it to the CPU chip to be untrusted." [31]. The SEV-SNP white paper states that "While certain physical attacks such as DRAM cold boot attacks (where DRAM chips are analyzed off-line) can be blocked by these technologies, on-line DRAM integrity attacks,

such as attacking the DDR bus while the VM is actively running, are out of scope. These attacks are very complex and require a significant level of local access and resources to perform" [4]. In this section, we discuss attacks on the DRAM interface as well as hardware-based mechanisms for undervolting the CPU.

3.5.1 Memory Bus

All x86 TEEs employ memory encryption to protect data before writing it to DRAM. However, the DDR protocol used on the memory bus remains unprotected. In addition, the physical sockets in which the memory modules are placed make it easy to access the individual physical pins of the bus. The memory modules are also referred to as DIMMs. On a high level, the DDR protocol specifies a location on the DIMM and whether data should be read or written. A DRAM address reflects the physical structure of the DIMM module and is defined in terms of ranks, banks, rows, and columns, while the CPU uses the concept of a one-dimensional physical address space. During system initialization, the BIOS discovers the installed DIMMs and sets up the physical address to DRAM mapping inside the *Memory Management Unit (MMU)* of the CPU. Similar to page tables for virtual memory, the MMU transparently translates physical addresses to DRAM addresses whenever the CPU performs a memory access. Meulemeester et al. [87] and Hopkins et al. [46] show that manipulating the DDR read/write request can be used to overcome memory access restrictions implemented by the CPU while Lee et al. [74] passively observe the memory bus to leak metadata of the memory accesses.

In *BadRAM*, Meulemeester et al. [87] show how an attacker with brief physical access can manipulate the physical address to DRAM address mapping to overcome physical address-based access restrictions, including the write access prevention introduced with SEV-SNP. They demonstrate the attack for both DDR4 and DDR5. To achieve this, they manipulate the data on the *Serial Presence Detect (SPD)* chip of the DIMM, which stores metadata like the memory size of the DIMM and does not use any form of authentication or data integrity mechanism. For most DIMMs, the manipulation requires one-time physical access, while some DIMMs even allow such changes from software. For their attack, Meulemeester et al. double the advertised size of the memory module. As a result, the physical address to DRAM address mapping created by the BIOS during system boot uses one additional "ghost" row-bit for the DRAM address. However, the DIMM itself silently ignores the "ghost" row-bit, leading to aliasing where each location on the DIMM is accessible by two physical addresses. This aliasing is completely transparent for the CPU. SEV-SNP blocks access to a contiguous physical range of memory to bootstrap its RMP mechanism for protecting the integrity of SEV VMs. However, by using aliased physical addresses, an attacker can again manipulate the data, rendering the mechanism ineffective. Based on this, Meulemeester et al. demonstrate an attack that completely

breaks the attestation mechanism of SEV-SNP. They also show that aliasing allows an attacker to overcome the EPC read/write protection of classic Intel SGX. However, due to the strong memory encryption, the attacker still cannot manipulate or replay data but only observe memory writes at a 64-byte granularity. Meulemeester et al. state that scalable Intel SGX and Intel TDX are not affected by these attacks, since they explicitly search for aliases during the boot process and disable the TEE features if any are found.

Hopkins et al. [46] show an FPGA-based DRAM interposer for DDR3 memory. In this context, an interposer is a physical man-in-the-middle device that is located between the DIMM and the actual physical memory socket on the mainboard of the computer. Their interposer allows redirecting memory reads/writes to a different location on the DIMM and can be dynamically controlled by the attacker. Leveraging this primitive, they show that an attacker can overcome any memory isolation mechanism that is based on physical addresses. The authors evaluate their attack on a CPU without TEE capabilities and show VM breakouts and privilege escalation attacks. On a TEE capable CPU, their interposer would enable the attacks described in the previously discussed BadRAM paper. Even worse, since the access redirection/aliasing is not static, like in the BadRAM paper, boot time alias checking cannot prevent the attack this time. However, modern, TEE-capable CPUs require at least DDR4 ECC memory, which requires much faster bus speeds than DDR3, for which Hopkins et al. build their interposer. Whether an FPGA-based design can scale up to such speeds remains an open problem.

Lee et al. [74] use a passive interposer to capture the DRAM addresses transmitted over the memory bus. They perform their experiments on CPUs with classic Intel SGX support and DDR4 memory. In contrast to the interposer by Hopkins et al., their interposer is commercially available but costs roughly $170,000. The captured DRAM addresses reveal the memory accesses of the CPU with cache line granularity. In contrast to regular cache attacks, their approach has no noise or runtime overhead. They demonstrate attacks that exploit secret-dependent memory accesses in SGX enclaves.

As we have explained in Section 3.1, scalable Intel SGX and Intel TDX both repurpose a single ECC bit as the “TD-bit”. The TD-bit marks memory as inaccessible for the untrusted host system. TDX also uses the ECC bits to store a 28-bit MAC that is only secure because mechanisms on the CPU prevent brute force attacks. Thus, an attack mechanism that grants direct access to the data, e.g., by manipulating the data transmitted over the memory bus, would have severe security implications for SGX and TDX.

3.5.2 HW-based Undervolting

In this section, we describe attacks that use physical access to manipulate the voltage of the CPU in order to cause data faults or glitches. In contrast to software-based attacks (c.f.

Section 3.3.6) that rely on certain MSRs for configuring *Dynamic Voltage and Frequency Scaling (DVFS)*, mitigating these hardware-based attacks is more challenging.

In *VoltPillager*, Chen et al. [27] undervolt the CPU by sending manipulated commands to the physical *Voltage Regulator (VR)* on the mainboard. The VR is connected to the CPU via the *Serial Voltage Identification (SVID)* bus which allows the CPU to dynamically control the supplied voltage. For their attacks, Chen et al. attach a microcontroller to the SVID bus to inject malicious requests onto the bus that temporarily drop the supplied voltage. Using this primitive, they show attacks similar to the ones presented in the software-based undervolting research [26, 66, 95, 99, 100]. The authors, e.g., show fault attacks against an RSA encryption running inside an Intel SGX enclave that allows to recover the secret key. While the authors only demonstrate their attacks on Intel CPUs, they state that the exploited mechanism should also affect other CPU vendors, like AMD.

In *One Glitch to Rule Them All*, Buhren et al. [15] use similar methods as Chen et al. [27] to undervolt AMD EPYC CPUs. Instead of targeting code executing on the main CPU, they show a fault attack against the *AMD Secure Processor (SP)*, an ARM-based coprocessor that is part of the system-on-chip. Crucially, the SP is the hardware root of trust for AMD SEV. Early during the boot process, the SP loads a public key from external memory and verifies its authenticity and integrity by comparing it to a hash value stored in on-chip memory. Afterward, the public key is used to verify the authenticity and integrity of subsequent code stages. Buhren et al. perform a glitch attack on the verification of the public key itself, allowing them to load their own key. This enables them to load manipulated versions of the subsequent code stages, gaining full control over the SP, which constitutes the hardware root of trust of SEV. They load a manipulated SEV firmware that allows the hypervisor to decrypt the memory of all SEV VMs while still issuing valid attestation reports. They demonstrate the attack on Zen 1, Zen 2, and Zen 3 CPUs. Thus, their attack affects SEV-SNP.

Both Chen et al. and Buhren et al. conclude that countermeasures against hardware-based undervolting are difficult. They argue that protecting the messages on the bus to the VR is insufficient as an attacker could easily swap out the VR itself due to its exposed position on the mainboard. Integrating the VR directly into the system-on-chip could make manipulations much harder. Another suggested approach is adding voltage monitoring circuitry to the CPU that tries to detect unexpected fluctuations in the supplied voltage and subsequently terminates the execution. Orthogonal to securing the hardware interface, the software-based "detect&abort" countermeasure from Kogler et al. [69] (.c.f Section 3.3.6) could potentially be applied here as well.

4

Conclusion

TEEs are a promising solution for ensuring data privacy in the context of cloud computing. As we have seen throughout this thesis, the strong TEE attacker model exposes several attack vectors that have led to numerous exploits. However, we have also seen continuous advancements in security mechanisms and mitigations that prevent many of the presented attacks. In my opinion, two areas are of particular interest for future TEE designs.

A recent trend in TEE design is partially shifting the trust anchor from the CPU to the DRAM modules. Classic SGX was the first widely available x86 TEE and focussed on protecting small, dedicated, security-critical applications. It uses a Merkle-Tree based memory encryption scheme with cryptographic integrity protection and freshness. The root of the tree is protected by secure, on-chip memory. As a result, classic SGX can treat the DRAM as entirely untrusted. However, the encryption scheme has a high time, space, and memory bandwidth overhead and does not scale well to large memory sizes. This is in contrast to many industry use cases for TEEs that demand protecting large legacy applications and complex software stacks. These needs were addressed by AMD SEV and later by Intel TDX. Both are VM-scoped, enabling a more straightforward adoption of existing software deployments. Scalable Intel SGX only partially addresses these needs. While it has a drastically increased memory size, it is still limited to processes. To efficiently scale to large memory sizes, SEV, scalable SGX, and TDX use tweaked block ciphers for memory encryption instead of a Merkle-Tree based design. The downside of this approach is that these schemes neither provide cryptographic integrity protection nor freshness. Intel TDX offers a separate mechanism to provide integrity protection via a SHA-3-based MAC with 28-bit tags but stores them together with the payload data and not on secure on-chip memory. As we have seen in Section 3.2 such encryption schemes are only secure in combination with access rights restrictions. Without integrity protection the attacker must be prevented from manipulating the ciphertexts and without freshness or nondeterministic encryption the attacker must not repeatedly read the ciphertexts.

To this end, AMD SEV, Intel TDX, and scalable Intel SGX employ additional access rights mechanisms. However, by design, such mechanisms cannot enforce any restriction for physical attackers. Thus, these TEEs have to trust that the physical connection to

the DRAM is secure and that the DRAM module itself is benign. As we have seen in Section 3.5.1, the memory bus and the DRAM modules are highly exposed to physical attacks, due to their external placement on the mainboard. However, the high bus speed of modern DDR4 and DDR5 DRAM makes memory bus interposer attacks technically challenging, requiring further research. Another attack avenue could be the software-only Rowhammer attack [67], which allows indirectly accessing memory locations, overcoming architectural access restrictions.

Implementing efficient access restrictions on the CPU that include privileged software components is non-trivial. To ensure optimal memory management, both SEV and TDX allow using arbitrary memory pages for their VMs, rendering a simple range check, as used by SGX, insufficient. AMD chose to implement a mechanism based on a flat page table called RMP, which itself resides in a small physically contiguous memory area protected by a range check. However, AMD only performs checks for write accesses, still leaving SEV-SNP exposed to the read-based ciphertext side-channel attack. I suspect this is due to performance considerations, as consulting the RMP comes with a time and memory bandwidth overhead. Starting from revision 1.55, the SEV ABI specification mentions a "ciphertext hiding" feature, indicating that AMD is working on a mitigation [6]. Intel instead repurposes one bit of the ECC memory to store whether a memory location can be accessed by the host. The ECC bits are also used to store the previously mentioned MAC. Thus, no additional memory access is required to perform the access rights check. While very efficient, repurposing ECC bits again places trust assumptions on the physical memory bus and the DRAM modules, which are hard to verify.

To what extent TEEs with tweaked block cipher-based memory encryption can withstand (advanced) physical attackers is an interesting future research direction. A hybrid system that allows storing critical secrets in a classic SGX-like environment could be an interesting approach but would require extensive changes to existing software architectures. On a physical level, verifying the current trust assumptions by authenticating the DRAM modules or integrating them more closely with the CPU would help to reduce the risk of physical attacks.

Another interesting area of TEE security, which has seen many recent developments, is side-channel security. In Section 3.3, we have seen that Intel SGX, AMD SEV, and Intel TDX all leak information about the control flow and memory accesses of the protected code. While the page fault side-channel only provides coarse-grained information, single-stepping enables the attacker to build instruction-granular traces. However, this information can only be weaponized if the control flow or memory accesses of the code are tied to "secret" data. In principle, it is well understood how to avoid both issues for cryptographic libraries by applying the data oblivious constant time programming

paradigm. However, academic research over and over found subtle control flow variations in state-of-the-art cryptographic libraries claiming to be constant time [91, 108, 122]. While the tooling for automatically detecting constant-time violations has improved as well [62, 120], this still raises questions about the practicality of consistently achieving constant-time behavior in the face of a single-stepping adversary. Even worse, both SEV and TDX are intended to protect entire VMs containing vast amounts of general-purpose software, like databases, key-value stores, or image processing. In many scenarios, the processed data needs to be considered secret in the sense of constant time programming. For certain applications, it has been shown that even the coarse-grained page fault controlled-channel is sufficient to leak data [126]. Rewriting large amounts of general-purpose software to adhere to the constant time programming paradigm is infeasible. Thus, TEEs need built-in, principled mitigations against this attack class, to enable the deployment of software components without the need for TEE-specific hardening. Intel started to address the issue and released a single-stepping countermeasure for SGX in 2023. In addition, Intel TDX was released with a built-in countermeasure against single-stepping. However, the latter still allows for a slightly weaker subvariant of single-stepping, and it seems unlikely that Intel will change the design [122]. None of the countermeasures mitigates the page fault controlled-channel. Finding robust countermeasures that can protect VM-based TEEs against both attacks is an interesting direction for future research.

While the previous paragraphs highlight shortcomings of current TEE designs, I want to stress that exploiting these shortcomings, as well as most other attacks described in this thesis, is non-trivial. In contrast to plain VMs or containers, TEEs significantly raise the security level of cloud computing. I am excited to continue working in this area, helping to shape more secure future TEE designs.

References

[1] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. *Intel Trust Domain Extensions (TDX) Security Review*. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf. Accessed on 07.10.2023. 2023-04.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port Contention for Fun and Profit". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 2019. DOI: 10.1109/SP.2019.00066. URL: https://doi.org/10.1109/SP.2019.00066.

[3] AMD. *AMD Secure Encryption Virtualization (SEV) Information Disclosure (Bulletin ID: AMD-SB-1013)*. https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1013. 2021.

[4] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. 2020-01.

[5] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Manual 24593, Rev. 3.42. AMD, 2024-03.

[6] AMD. *SEV Secure Nested Paging Firmware ABI Specification*. Tech. rep. 56860, Rev. 1.55. AMD, 2023-09.

[7] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. "S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015. DOI: 10.1109/SP.2015.42. URL: https://doi.org/10.1109/SP.2015.42.

[8] Diego F. Aranha, Sebastian Berndt, Thomas Eisenbarth, Okan Seker, Akira Takahashi, Luca Wilke, and Greg Zaverucha. "Side-Channel Protections for Picnic Signatures". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.4 (2021), pp. 239–282. DOI: 10.46586/TCHES.V2021.I4.239-282. URL: https://doi.org/10.46586/tches.v2021.i4.239-282.

[9] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.10. Arpaci-Dusseau Books, 2023-11.

[10] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. "Foundations of garbled circuits". In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 2012. DOI: 10.1145/2382196.2382279. URL: https://doi.org/10.1145/2382196.2382279.

[11] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. "ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture". In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/borrello.

[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. 2017. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

[13] Samira Briongos, Pedro Malagón, José Manuel Moya, and Thomas Eisenbarth. "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/briongos.

[14] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. "Fault Attacks on Encrypted General Purpose Compute Platforms". In: *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ , USA, March 22-24, 2017*. 2017. DOI: 10.1145/3029806.3029836. URL: https://doi.org/10.1145/3029806.3029836.

[15] Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean -Pierre Seifert. "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3484779. URL: https://doi.org/10.1145/3460120.3484779.

[16] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. "Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019. DOI: 10.1145/3319535.3354216. URL: https://doi.org/10.1145/3319535.3354216.

[17] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[18] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00089. URL: https://doi.org/10.1109/SP40000.2020.00089.

[19] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018. DOI: 10.1145/3243734.3243822. URL: https://doi.org/10.1145/3243734.3243822.

[20] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. 2017. DOI: 10.1145/3152701.3152706. URL: https://doi.org/10.1145/3152701.3152706.

[21] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. "Fallout: Leaking Data on Meltdown-resistant CPUs". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019. DOI: 10.1145/3319535.3363219. URL: https://doi.org/10.1145/3319535.3363219.

[22] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. "KASLR: Break It, Fix It, Repeat". In: *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*. 2020. DOI: 10.1145/3320269.3384747. URL: https://doi.org/10.1145/3320269.3384747.

[23] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. "SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution". In: *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. 2019. DOI: 10.1109/EUROSP.2019.00020. URL: https://doi.org/10.1109/EuroSP.2019.00020.

[24] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. "Defeating Speculative-Execution Attacks on SGX with HyperRace". In: *2019 IEEE Conference on Dependable and Secure Computing, DSC 2019, Hangzhou, China, November 18-20, 2019*. 2019. DOI: 10.1109/DSC47296.2019.8937682. URL: https://doi.org/10.1109/DSC47296.2019.8937682.

[25] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. "Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*. 2017. DOI: 10.1145/3052973.3053007. URL: https://doi.org/10.1145/3052973.3053007.

[26] Zitai Chen and David F. Oswald. "PMFault: Faulting and Bricking Server CPUs through Management Interfaces Or: A Modern Example of Halt and Catch Fire". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.2 (2023), pp. 1–23. DOI: 10.46586/TCHES.V2023.I2.1-23. URL: https://doi.org/10.46586/tches.v2023.i2.1-23.

[27] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David F. Oswald, and Flavio D. Garcia. "VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai.

[28] Md Hafizul Islam Chowdhuryy, Zhenkai Zhang, and Fan Yao. "PowSpectre: Powering Up Speculation Attacks with TSX-based Replay". In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. 2024. DOI: 10.1145/3634737.3661139. URL: https://doi.org/10.1145/3634737.3661139.

[29] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 2019. DOI: 10.1109/SP.2019.00089. URL: https://doi.org/10.1109/SP.2019.00089.

[30] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. "AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves". In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/constable.

[31] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptol. ePrint Arch.* (2016), p. 86. URL: http://eprint.iacr.org/2016/086.

[32] Sen Deng, Mengyuan Li, Yining Tang, Shuai Wang, Shoumeng Yan, and Yinqian Zhang. “CipherH: Automated Detection of Ciphertext Side-channel Vulnerabilities in Cryptographic Implementations”. In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/deng-sen.

[33] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. “Secure Encrypted Virtualization is Unsecure”. In: *CoRR* abs/1712.05090 (2017). arXiv: 1712.05090. URL: http://arxiv.org/abs/1712.05090.

[34] Dmitry Evtyushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. In: *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. 2016. DOI: 10.1109/MICRO.2016.7783743. URL: https://doi.org/10.1109/MICRO.2016.7783743.

[35] Dmitry Evtyushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. “BranchScope: A New Side-Channel Attack on Directional Branch Predictor”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. 2018. DOI: 10.1145/3173162.3173204. URL: https://doi.org/10.1145/3173162.3173204.

[36] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. “Scalable Memory Protection in the PENGLAI Enclave”. In: *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. 2021. URL: https://www.usenix.org/conference/osdi21/presentation/feng.

[37] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. “CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP”. In: *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. 2025-02.

[38] Craig Gentry. “A fully homomorphic encryption scheme”. PhD thesis. Stanford University, USA, 2009. URL: https://searchworks.stanford.edu/view/8493082.

[39] Google. *Google Cloud Confidential Computing / Confidential VMs*. https://cloud.google.com/compute/confidential-vm/docs/about-cvm. Accessed: 2021-01-26.

[40] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/gras.

[41] Shay Gueron. "A Memory Encryption Engine Suitable for General Purpose Processors". In: *IACR Cryptol. ePrint Arch.* (2016), p. 204. URL: http://eprint.iacr.org/2016/204.

[42] Marcus Hähnel, Weidong Cui, and Marcus Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel.

[43] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 6th Edition*. Morgan Kaufmann, 2018-01. ISBN: 978-0-12-811905-1.

[44] Felicitas Hetzelt and Robert Buhren. "Security Analysis of Encrypted Virtual Machines". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*. 2017. DOI: 10.1145/3050748.3050763. URL: https://doi.org/10.1145/3050748.3050763.

[45] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. "VIA: Analyzing Device Interfaces of Protected Virtual Machines". In: *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. 2021. DOI: 10.1145/3485832.3488011. URL: https://doi.org/10.1145/3485832.3488011.

[46] Bradley D. Hopkins, John Shield, and Chris North. "Redirecting DRAM memory pages: Examining the threat of system memory Hardware Trojans". In: *2016 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016, McLean, VA, USA, May 3-5, 2016*. 2016. DOI: 10.1109/HST.2016.7495582. URL: https://doi.org/10.1109/HST.2016.7495582.

[47] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. "Fault Injection Techniques and Tools". In: *Computer* 30.4 (1997), pp. 75–82. DOI: 10.1109/2.585157. URL: https://doi.org/10.1109/2.585157.

[48] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. "Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.1 (2020), pp. 321–347. DOI: 10.13154/TCHES.V2020.I1.321-347. URL: https://doi.org/10.13154/tches.v2020.i1.321-347.

[49] Intel. *11th Generation Intel Core Processor*. Volume1, Document Number 631121-012. 2023-05.

[50] Intel. *12th Generation Intel Core Processor*. Volume Revision 012, Document Number 655258-010. 2023-02.

[51] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1 to 4*. Revision 325462-084US. 2024-06.

[52] Intel. *Intel Architecture Memory Encryption Technologies*. Revision 336907-004US. 2022-10.

[53] Intel. *Intel Hardware Shield - Intel Total Memory Encryption*. White Paper. 2021-05.

[54] Intel. *Intel Security Announcement 2024-10-08-001*. https://www.intel.com/content/www/us/en/security-center/announcement/intel-security-announcement-2024-10-08-001.html. Accessed on 14.11.2024. 2024.

[55] Intel. *Intel Trust Domain Extensions*. https://cdrdv2.intel.com/v1/dl/getContent/690419. Accessed on 19.09.2024. 2023-02.

[56] Intel. *Intel Trust Domain Extensions (Intel TDX) Module Architecture Application Binary Interface (ABI) Reference Specification*. Revision 348551-004US. 2024-03.

[57] Intel. *Intel Trust Domain Extensions (Intel TDX) Module Base Architecture Specification*. Revision 348549-004US. 2024-03.

[58] Intel. *INTEL-SA-00161*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/l1-terminal-fault.html. Accessed on 18.11.2024. 2018.

[59] Intel. *MKTME Side Channel Impact on Intel TDX*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/mktme-side-channel-impact-on-intel-tdx.html. Accessed on 07.10.2023. 2023.

[60] Intel. *Speculative Execution Side Channel Mitigations*. Revision 2.0, Document Number 336996-002. 2018-05.

[61] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/islam.

[62] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. “"They’re not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks”. In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. 2022. DOI: 10.1109/SP46214.2022.9833713. URL: https://doi.org/10.1109/SP46214.2022.9833713.

[63] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. *Supporting Intel SGX on Multi-Socket Platforms*. White Paper. Intel, 2021.

[64] David Kaplan. *Protecting VM Register state with SEV-ES*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf. 2017-02.

[65] David Kaplan, Jeremy Powell, and Wolle. *AMD Memory Encryption*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf. 2021-10.

[66] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. “V0LTpwn: Attacking x86 Processor Integrity from Software”. In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/kenjar.

[67] Yoongu Kim, Ross Daly, Jeremie S. 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”. In: *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. 2014. DOI: 10.1109/ISCA.2014.6853210. URL: https://doi.org/10.1109/ISCA.2014.6853210.

[68] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 2019. DOI: 10.1109/SP.2019.00002. URL: https://doi.org/10.1109/SP.2019.00002.

[69] Andreas Kogler, Daniel Gruss, and Michael Schwarz. “Minefield: A Software-only Protection for SGX Enclaves against DVFS Attacks”. In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-minefield.

[70] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer". In: *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*. 2018. URL: https://www.usenix.org/conference/woot18/presentation/koruyeh.

[71] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. "RAMBleed: Reading Bits in Memory Without Accessing Them". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00020. URL: https://doi.org/10.1109/SP40000.2020.00020.

[72] Fan Lang, Wei Wang, Lingjia Meng, Jingqiang Lin, Qiongxiao Wang, and Linli Lu. "MoLE: Mitigation of Side-channel Attacks against SGX via Dynamic Data Location Escape". In: *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*. 2022. DOI: 10.1145/3564625.3568002. URL: https://doi.org/10.1145/3564625.3568002.

[73] David Lantz, Felipe Boeira, and Mikael Asplund. "Towards Self-monitoring Enclaves: Side-Channel Detection Using Performance Counters". In: *Secure IT Systems - 27th Nordic Conference, NordSec 2022, Reykjavic, Iceland, November 30-December 2, 2022, Proceedings*. 2022. DOI: 10.1007/978-3-031-22295-5_7. URL: https://doi.org/10.1007/978-3-031-22295-5%5C_7.

[74] 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: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lee-dayeol.

[75] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho.

[76] 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: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. 2022. DOI: 10.1109/SP46214.2022.9833768. URL: https://doi.org/10.1109/SP46214.2022.9833768.

[77] Mengyuan Li, Yuheng Yang, Guoxing Chen, Mengjia Yan, and Yinqian Zhang. “SoK: Understanding Design Choices and Pitfalls of Trusted Execution Environments”. In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. 2024. DOI: 10.1145/3634737.3644993. URL: https://doi.org/10.1145/3634737.3644993.

[78] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. “CrossLine: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV”. In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3485253. URL: https://doi.org/10.1145/3460120.3485253.

[79] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. “Exploiting Unprotected I/O Operations in AMD’s Secure Encrypted Virtualization”. In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan.

[80] 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: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan.

[81] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. “TLB Poisoning Attacks on AMD Secure Encrypted Virtualization”. In: *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. 2021. DOI: 10.1145/3485832.3485876. URL: https://doi.org/10.1145/3485832.3485876.

[82] Moritz Lipp, Vedad Hadzic, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. “Take A Way: Exploring the Security Implications of AMD’s Cache Way Predictors”. In: *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*. 2020. DOI: 10.1145/3320269.3384746. URL: https://doi.org/10.1145/3320269.3384746.

[83] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. “PLATYPUS: Software-based Power Side-Channel Attacks on x86”. In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. 2021. DOI: 10.1109/SP40001.2021.00063. URL: https://doi.org/10.1109/SP40001.2021.00063.

[84] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[85] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015. DOI: 10.1109/SP.2015.43. URL: https://doi.org/10.1109/SP.2015.43.

[86] Giorgi Maisuradze and Christian Rossow. "ret2spec: Speculative Execution Using Return Stack Buffers". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018. DOI: 10.1145/3243734.3243761. URL: https://doi.org/10.1145/3243734.3243761.

[87] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. "BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments". In: *to appear at the 2025 IEEE Symposium on Security and Privacy*. 2025.

[88] Microsoft. *Azure confidential computing*. https://docs.microsoft.com/en-us/azure/confidential-computing. Accessed: 2021-01-26.

[89] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. 2017. DOI: 10.1007/978-3-319-66787-4_4. URL: https://doi.org/10.1007/978-3-319-66787-4%5C_4.

[90] Daniel Moghimi. "Downfall: Exploiting Speculative Data Gathering". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA, 2023-08. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/moghimi.

[91] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat.

[92] Mathias Morbitzer, Manuel Huber, and Julian Horsch. "Extracting Secrets from Encrypted Virtual Machines". In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX , USA, March 25-27, 2019*. 2019. DOI: 10.1145/3292006.3300022. URL: https://doi.org/10.1145/3292006.3300022.

[93] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. "SEVered: Subverting AMD's Virtual Machine Encryption". In: *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*. 2018. DOI: 10.1145/3193111.3193112. URL: https://doi.org/10.1145/3193111.3193112.

[94] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. "SEVerity: Code Injection Attacks against Encrypted Virtual Machines". In: *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*. 2021. DOI: 10.1109/SPW53761.2021.00063. URL: https://doi.org/10.1109/SPW53761.2021.00063.

[95] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00057. URL: https://doi.org/10.1109/SP40000.2020.00057.

[96] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. 2018. URL: https://www.usenix.org/conference/atc18/presentation/oleksenko.

[97] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*. 2006. DOI: 10.1007/11605805_1. URL: https://doi.org/10.1007/11605805%5C_1.

[98] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/puddu.

[99] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. "VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019.

DOI: 10.1145/3319535.3354201. URL: https://doi.org/10.1145/3319535.3354201.

[100] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. "VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults". In: *Asian Hardware Oriented Security and Trust Symposium, AsianHOST 2019, Xi'an, China, December 16-17, 2019*. 2019. DOI: 10.1109/ASIANHOST47458.2019.9006701. URL: https://doi.org/10.1109/AsianHOST47458.2019.9006701.

[101] Martin Radev and Mathias Morbitzer. "Exploiting Interfaces of Secure Encrypted Virtual Machines". In: *Reversing and Offensive-oriented Trends Symposium (ROOTS)*. 2020.

[102] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CrossTalk: Speculative Data Leaks Across Cores Are Real". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. 2021. DOI: 10.1109/SP40001.2021.00020. URL: https://doi.org/10.1109/SP40001.2021.00020.

[103] Stephan van Schaik, Alyssa Milburn, Sebastian Ö sterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-Flight Data Load". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 2019. DOI: 10.1109/SP.2019.00087. URL: https://doi.org/10.1109/SP.2019.00087.

[104] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. "WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP". In: *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. 2024. DOI: 10.1109/SP54263.2024.00262. URL: https://doi.org/10.1109/SP54263.2024.00262.

[105] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. "HECKLER: Breaking Confidential VMs with Malicious Interrupts". In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/schl%20%5C%C3%5C%BCter.

[106] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019. DOI: 10.1145/3319535.3354252. URL: https://doi.org/10.1145/3319535.3354252.

[107] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/.

[108] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. "Util: : Lookup: Exploiting Key Decoding in Cryptographic Libraries". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3484783. URL: https://doi.org/10.1145/3460120.3484783.

[109] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. "MicroScope: Enabling Microarchitectural Replay Attacks". In: *IEEE Micro* 40.3 (2020), pp. 91–98. DOI: 10.1109/MM.2020.2986204. URL: https://doi.org/10.1109/MM.2020.2986204.

[110] *Spectre Variant 4*. https://project-zero.issues.chromium.org/issues/42450580. 2018.

[111] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, and Shweta Shinde. "SIGY: Breaking Intel SGX Enclaves with Malicious Exceptions & Signals". In: *CoRR* abs/2404.13998 (2024). DOI: 10.48550/ARXIV.2404.13998. arXiv: 2404.13998. URL: https://doi.org/10.48550/arXiv.2404.13998.

[112] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. 2018. DOI: 10.1145/3173162.3177155. URL: https://doi.org/10.1145/3173162.3177155.

[113] Andrew Tanenbaum and Herbert Bos. *Modern Operating Systems, Global Edition*. Pearson, 2023-03. ISBN: 978-1-292-72789-9.

[114] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management". In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[115] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. "TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering". In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/tatar.

[116] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. "PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 20th International Conference, DIMVA 2023, Hamburg, Germany, July 12-14, 2023, Proceedings*. 2023. DOI: 10.1007/978-3-031-35504-2_3. URL: https://doi.org/10.1007/978-3-031-35504-2%5C_3.

[117] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. "The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*. 2019. DOI: 10.1145/3321705.3329820. URL: https://doi.org/10.1145/3321705.3329820.

[118] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. "Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software". In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/wichelmann.

[119] Jan Wichelmann, Anja Rabich, Anna Pätschke, and Thomas Eisenbarth. "Obelix: Mitigating Side-Channels through Dynamic Obfuscation". In: *2024 IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 20-23, 2024*. IEEE Computer Society. 2024. DOI: 10.1109/SP54263.2024.00261. URL: https://doi.org/10.1109/SP54263.2024.00261.

[120] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. "Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. 2022. DOI: 10.1145/3548606.3560654. URL: https://doi.org/10.1145/3548606.3560654.

[121] Luca Wilke and Gianluca Scopelliti. "SNPGuard: Remote Attestation of SEV-SNP VMs Using Open Source Tools". In: *IEEE European Symposium on Security and Privacy Workshops, EuroS&PW 2024, Vienna, Austria, July 8-12, 2024*. 2024. DOI: 10.1109/EUROSPW61312.2024.00026. URL: https://doi.org/10.1109/EuroSPW61312.2024.00026.

[122] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. "TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX". In: *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Utah, USA*. 2024.

[123] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. "SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00080. URL: https://doi.org/10.1109/SP40000.2020.00080.

[124] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. "SEV-Step A Single-Stepping Framework for AMD-SEV". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 180–206. DOI: 10.46586/TCHES.V2024.I1.180-206. URL: https://doi.org/10.46586/tches.v2024.i1.180-206.

[125] Luca Wilke, Jan Wichelmann, Florian Sieck, and Thomas Eisenbarth. "undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation". In: *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*. 2021. DOI: 10.1109/SPW53761.2021.00064. URL: https://doi.org/10.1109/SPW53761.2021.00064.

[126] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015. DOI: 10.1109/SP.2015.45. URL: https://doi.org/10.1109/SP.2015.45.

[127] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[128] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd M. Austin. "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors". In: *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*. 2017. DOI: 10.1109/HPCA.2017.10. URL: https://doi.org/10.1109/HPCA.2017.10.

[129] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. "CipherSteal: Stealing Input Data from TEE-Shielded Neural Networks with Ciphertext Side Channels". In: *to appear at the 2025 IEEE Symposium on Security and Privacy*. 2025.

[130] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. "HyperTheft: Thieving Model Weights from TEE-Shielded Neural Networks via Ciphertext Side Channels". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*. 2024. DOI: 10.1145/3658644.3690317. URL: https://doi.org/10.1145/3658644.3690317.

[131] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. "CacheWarp: Software-based Fault Injection using Selective State Reset". In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-ruiyi.

[132] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. "Ultimate SLH: Taking Speculative Load Hardening to the Next Level". In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh.

Part II

Publications

5

undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation

Publication

Luca Wilke, Jan Wichelmann, Florian Sieck and Thomas Eisenbarth. In the *15th IEEE Workshop on Offensive Technologies*, 2021.

Contribution

I am the main author.

Outline

undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation

Luca Wilke, Jan Wichelmann, Florian Sieck and Thomas Eisenbarth.

University of Lübeck

The ongoing trend of moving data and computation to the cloud is met with concerns regarding privacy and protection of intellectual property. Cloud Service Providers (CSP) must be fully trusted to not tamper with or disclose processed data, hampering adoption of cloud services for many sensitive or critical applications. As a result, CSPs and CPU manufacturers are rushing to find solutions for secure and trustworthy outsourced computation in the Cloud. While enclaves, like Intel SGX, are strongly limited in terms of throughput and size, AMD's Secure Encrypted Virtualization (SEV) offers hardware support for transparently protecting code and data of entire VMs, thus removing the performance, memory and software adaption barriers of enclaves. Through attestation of boot code integrity and means for securely transferring secrets into an encrypted VM, CSPs are effectively removed from the list of trusted entities. There have been several attacks on the security of SEV, by abusing I/O channels to encrypt and decrypt data, or by moving encrypted code blocks at runtime. Yet, none of these attacks have targeted the attestation protocol, the core of the secure computing environment created by SEV. We show that the current attestation mechanism of Zen 1 and Zen 2 architectures has a significant flaw, allowing us to manipulate the loaded code without affecting the attestation outcome. An attacker may abuse this weakness to inject arbitrary code at startup—and thus take control over the entire VM execution, without any indication to the VM's owner. Our attack primitives allow the attacker to do extensive modifications to the bootloader and the operating system, like injecting spy code or extracting secret data. We present a full end-to-end attack, from the initial exploit to leaking the key of the encrypted disk image during boot, giving the attacker unthrottled access to all of the VM's persistent data.

1 Introduction

An increasing number of software applications, from enterprise management software to messengers used in nearly everyone's daily life, rely on storing information and performing computations in the cloud. Solutions are moved from local, trusted environments to the data centers of big cloud service providers, and are now running in untrusted

environments under control of a third party—in order to save costs, reduce management effort and to improve scalability.

The loss of trust comes with significant challenges for services such as banking, private secure messaging or health services, which require strict isolation and confidentiality to ensure the safety of their assets and to comply with data privacy laws: Computing resources in the cloud are often shared, which in case of broken isolation does allow co-located users to spy on each other [24, 31, 45]. Another concern is the security of the cloud service provider's systems themselves, where internal or external attackers may leverage elevated privileges for extracting private data.

In order to deliver isolated, confidential and authenticated execution and processing of data in an otherwise untrusted setting, processor vendors added hardware features to build a root-of-trust and ensure confidential computing in a local Trusted Execution Environment (TEE). One example is AMD Secure Encrypted Virtualization (SEV) [1, 6, 28], which allows to run VMs confidentially and isolated from their hypervisor. AMD added new features to SEV with every generation of its processor architecture. In 2017, The first generation of EPYC processors (Zen) came with the initial version of SEV. The second generation (Zen 2) added an encrypted state for context switches with SEV-ES [27] and was released in 2019. The newest addition, SEV-SNP [3], will be available on the third generation of EPYC processors (Zen 3), which are set to be released in early 2021. Intel TDX [25] aims to provide a similar solution, but is only available as a concept as of writing this work. With Intel Software Guard Extensions (SGX) [9, 18, 26], Intel offers an established TEE which enables software vendors to run smaller programs in isolated enclaves. All of these solutions provide memory encryption during execution, and attestation of the software loaded into the TEE.

Recently, cloud service providers like Microsoft and Google started to offer confidential computing environments which isolate the customer's software using Intel SGX [32] or AMD SEV [22]. Popular examples, like the secure private messenger Signal, are already using these technologies to protect the sensitive data of their customers [39]. Moreover, open source solutions enable simple development and deployment of software for TEEs [10, 11, 20].

A fundamental challenge for TEEs is having to guarantee their promises against attackers with system level privileges, resulting in a large variety of attacks [15, 16, 17, 33, 38, 44]. In this work, we extend the arsenal of attacks against TEEs and in particular against AMD SEV, with an attack targeting and circumventing its very core of trust, the remote attestation.

Remote attestation allows the owner of a software, which runs in a confidential or trusted execution environment, to verify the initial integrity and authenticity of the software

loaded into the TEE, which afterwards is preserved at runtime by the properties of the TEE. Generally, remote attestation works by creating a signed measurement, usually a hash, of the initially loaded application through the trusted hardware and sending this measurement to the software owner for verification. In case of AMD SEV, the trusted hardware is an additional on-chip co-processor called Secure Processor (SP), which cannot be externally controlled.

1.1 Our Contribution

If the attestation process, however, is broken, the isolation and confidentiality guarantees of AMD SEV are inconsequential as the software owner cannot be sure whether their intended software was loaded or whether an attacker manipulated it during startup.

In this work, we

- show that the measurement used in AMD SEV's attestation is block permutation-agnostic, meaning that changing the order of measured memory blocks does not affect the attestation outcome, and thus allows the attacker to modify the execution flow without detection by the VM's owner;
- construct an universal attack primitive, which reorders the measured blocks of an initially loaded UEFI and sets up a Return-oriented Programming (ROP) chain to load and execute arbitrary code;
- demonstrate a full end-to-end attack which leaks the key of an encrypted disk image, and gives the attacker full control over the VM's operating system;
- propose several countermeasures and discuss why the underlying problem ultimately cannot be solved under SEV Encrypted State (SEV-ES).

1.2 Attack Overview

The attack described in this work targets the measurement of the initially loaded binary during startup of an SEV-ES-protected Virtual Machine (VM), which we also refer to as the "guest". When the VM is started, the hypervisor instructs the AMD SEV secure processor to load the initial binary, e.g. an Open Virtual Machine Firmware (OVFM) UEFI binary, into encrypted memory and calculate a measurement of the initial VM content using the `LAUNCH_UPDATE_DATA` and `LAUNCH_MEASURE` commands. We find that the initial binary can be split into blocks as small as 16 bytes, which we are able to load in an arbitrary order using `LAUNCH_UPDATE_DATA`, while still getting the same measurement when calling `LAUNCH_MEASURE`. This allows us to construct our own execution flow, which

we use for redirecting the stack pointer to an unencrypted shared page. Consequently, we leverage this control over the stack to mount a ROP attack, allowing us to write arbitrary code and data into the encrypted VM's memory. We use the injected code to leak the protected secret values which have been provided by the guest owner. As our meddling with the block ordering does not change the launch measurement, AMD SEV's remote attestation will succeed and the guest owner will be unaware of our changes to their VM's execution flow.

1.3 Responsible Disclosure

We responsibly disclosed our findings to AMD via Email on January 19th, 2021. AMD requested an embargo until May 11, 2021 and provided us with the following statement: "AMD has assigned CVE-2021-26311 for this issue and provided mitigations in the SEV-SNP feature available for enablement 3rd Gen AMD EPYC™ processors. AMD appreciates the coordination efforts made by the research team.".

2 Background

2.1 AMD SEV

In 2016, AMD introduced their Secure Memory Encryption (SME) and SEV technologies [28], which were implemented only in 2017 with the first generation of EPYC processors (Zen 1). SME offers hardware-based encryption of RAM content. The memory encryption key is managed by the SP, an ARM-based co-processor, and is thus never accessible by system software. The encryption/decryption takes place directly in the on-die memory controllers. Each page table entry has a special status bit, which controls whether the associated page is encrypted or not. The whitepaper [28] does not explain the mode of operation in detail, but only states that AES with an 128-bit key and a physical address-based tweak is used. In [19, 44] it is shown that early versions use the Xor-Encrypt (XE) or Xor-Encrypt-Xor (XEX) encryption mode with static, low entropy tweak values, while later versions use stronger, randomized tweak values. In addition, none of the encryption modes offer integrity protection.

While SME uses the same key for all memory pages, SEV adds the ability to encrypt the memory content of VMs with different keys, that are only known to the SP but not to the Hypervisor (HV), preventing a malicious HV from directly reading the memory content of its guests. However, VMs can also share pages with the HV. In addition, the SP offers an API to the HV to manage the SEV-protected VMs. This includes a mechanism to attest

the initially loaded code of the VM and a mechanism to securely move secrets into the VM.

SEV-ES was introduced by AMD in 2017 and implemented in 2019 with the second generation of EPYC processors. It addresses one major remaining attack surface of SEV: The unencrypted Virtual Machine Control Block (VMCB), a data structure storing certain configuration bits as well the VM's register values on context switches. Certain sensitive parts of the VMCB were moved to a substructure called Virtual Machine State Save Area (VMSA) that is encrypted and integrity protected on context switches to the HV, and thus prevents an attacker from inferring or modifying a VM's state during context switches.

However, there are also several instructions that need interaction with the HV, like `cpuid`, which previously shared and received data with/from the HV via the VM's registers. To enable this with SEV-ES, AMD introduced a new communication mechanism between HV and VM, consisting of the Guest Hypervisor Communication Block (GHCB) and a new exception called VMM Communication Exception (#VC). The GHCB is simply a shared page, that gets setup by the VM. Instructions that require data sharing with the HV cause a #VC, allowing a VM exception handler to share the data via the GHCB.

SEV Secure Nested Paging (SEV-SNP) aims to address several remaining issues, like remapping attacks due to the HV's control over the nested page tables, or attacks on the missing integrity protection. It was announced by AMD in January 2020 [3] and will only be available on the 3rd gen EPYC processors, that are set to be released after the submission deadline. The most important change is the introduction of an additional page table called Reverse Map Table (RMP), to which the HV only has mediated access. The RMP aims to ensure a one-to-one mapping between GPAs and HPAs, and will prevent the HV from writing to VM memory, mitigating problems arising due to the lacking integrity protection.

2.2 Booting physical and virtual environments

When booting a physical system, the CPU is in a well-defined state, but completely unaware of its environment. Its program counter is set to start execution at a fixed address, which points to a FLASH or EPROM. Firmware is loaded from this start position and is responsible for initializing the memory controller, creating a memory mapping for RAM, and configuring I/O peripherals. On modern computers, this firmware usually is UEFI-based, which is platform specific and performing the aforementioned tasks. The advantage of UEFI compared to legacy BIOS is its standardization of the platform initialization procedure [42]. The UEFI is configurable through variables in a non-volatile memory (e.g. NVRAM), so configuration is persisted over restarts. Additionally, it can provide *secure boot*, which allows to build up a chain of trust from the UEFI to the finally

loaded kernel of the OS. In this chain, the UEFI, which contains a set of configurable certificates and keys, forms the root of trust. Every component of the chain verifies its successor before handing over the execution [42].

When the UEFI finishes the system configuration, it hands over the control to an EFI binary [42], which usually is an OS bootloader, e.g. Grub [21]. The bootloader sets up the stage for the OS kernel, loads the kernel into memory and calls its main method. However, an EFI binary does not necessarily have to be a bootloader.

In case of booting a virtual environment, e.g. with QEMU [37], the process is similar: When the hypervisor starts up the virtualization, it launches an UEFI. For virtual environments, OVMF [41] is a common choice. OVMF performs the necessary virtual system configuration and hands over control to a bootloader [12]. The bootloader, which is just a regular EFI application, can be provided in different ways. Usually it is expected to be located on a FAT-formatted disk with GUID partition table [42], thus requiring the guest owner to provide the hypervisor with a disk image. Another way is to include the bootloader, i.e. the EFI application, into the UEFI volume which also contains the UEFI firmware [13].

3 SEV(-ES) Guest Launch Process

In this section, we describe the typical workflow required to launch an SEV-secured VM, as described in [6] and implemented in AMD's patches to the the Linux kernel [4] and QEMU [5]. This includes encrypting an initial code image, proving its integrity to the guest owner, and loading secret data without leaking it to the HV.

3.1 Prerequisites

There are three parties involved in launching a SEV VM: The guest owner, the HV and the SP. The guest owner wants to start a SEV-secured VM. The HV is typically controlled by the cloud service provider. In order to provide the SEV functionality, the HV must interact with the API provided by the SP.

The goal of the launch process is to enable the HV to prove to the guest owner that the initial content is trustworthy. Furthermore, it enables the guest owner to send secrets, like disk encryption or SSH keys, to the VM in a secure manner. The entire launch process is illustrated in Figure 1.

For each VM, the SP maintains a Guest Context (GCTX) that, among other values, contains a handle, the VM Encryption Key (VEK), the launch digest (LD), and the current state.

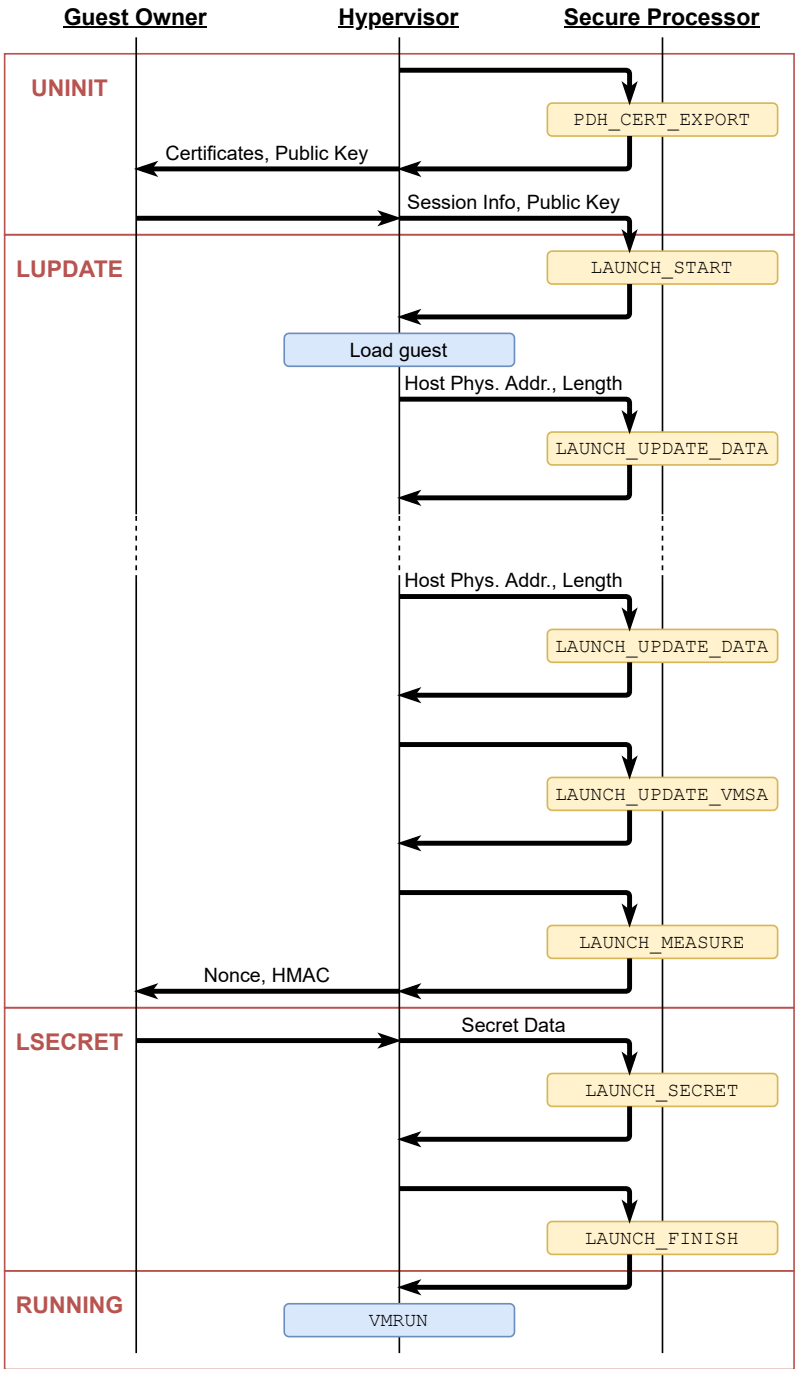


Figure 1: SEV Guest Launch Process. This simplified illustration shows the various states during VM startup and attestation. First, keys are exchanged and a cryptographic session is established. In the `LUPDATE` state, the hypervisor loads the guest and then asks the SP to encrypt it via repeated `LAUNCH_UPDATE_`$*$ commands. The final `LAUNCH_MEASURE` call retrieves a signed hash of the loaded guest data. If the guest owner approves, various secret data can be safely loaded into the VM during the `LSECRET` state. On completion, the hypervisor and the VM transition into the RUNNING state, and the VM is executed.

Table 1: Overview of the VM-specific commands provided by the SP in different states. For brevity, commands that are not relevant for our work were omitted.

State	Command	→	New State
UNINIT	LAUNCH_START	→	LUPDATE
LUPDATE	LAUNCH_UPDATE_DATA	→	LUPDATE
	LAUNCH_UPDATE_VMSA	→	LUPDATE
	LAUNCH_MEASURE	→	LSECRET
LSECRET	LAUNCH_SECRET	→	LSECRET
	LAUNCH_FINISH	→	RUNNING
RUNNING	(other commands)		

The VEK is a VM-specific key used for memory encryption. The launch digest contains a hash value of the VM contents loaded during the launch measurement phase. The state determines which API commands are usable.

Table 1 shows an overview of the states along with the usable commands and the resulting state transitions. We omitted all states and commands related to migrating VMs between different hosts, as we do not use them in this paper. In addition to the VM-specific commands, there are several commands which affect the SP itself. They are used to update its firmware and to generate or export cryptographic key material.

Before any VM-specific commands are issued, the HV starts an ECDH key exchange by issuing the PDH_CERT_EXPORT command, upon which the SP exports a public ECDH key and some certificates. The latter are part of a public key infrastructure, that is ultimately rooted at an AMD controlled key hardcoded into the SP. The HV then sends this data to the guest owner.

3.2 UNINIT state

A new VM assumes UNINIT as initial state. In order to start the launch process, the guest owner verifies the authenticity of the ECDH key sent by the HV. Then they use their own ECDH key pair to derive the Transport Encryption Key (TEK), the Transport Integrity Key (TIK) and some other keys used for transport security. Afterwards, they send this data together with a configuration object called POLICY to the HV.

Upon receiving the data from the guest owner, the HV calls the LAUNCH_START command, which finalizes the ECDH handshake between guest owner and SP. The SP can now derive the shared secret and use it to unwrap/verify the received data. Next, it initializes the GCTX using the received guest policy and generates a new VEK.

3.3 LUPDATE state

In the LUPDATE state, the HV has access to three primary commands: LAUNCH_UPDATE_DATA, LAUNCH_UPDATE_VMSA and LAUNCH_MEASURE. The LAUNCH_UPDATE_DATA command allows the HV to specify a guest handle, a 16 byte aligned Host Physical Address (HPA) PADDR and a multiple of 16 bytes L as a length. The SP will then in-place encrypt the next L bytes starting at PADDR with the VEK of the VM denoted by the handle. In addition, the launch digest field of the GCTX is updated with the plaintext of the encrypted data (see Section 5.1). The intention of LAUNCH_UPDATE_DATA is to encrypt and measure the initial content of the VM, such that the HV can no longer modify it. Encrypting the initial content is mandatory, since the VM initially assumes that all memory accesses are encrypted, so it can only execute the initial code if it has been encrypted beforehand.

The LAUNCH_UPDATE_VMSA command is only applicable to SEV-ES VMs and works very similar to LAUNCH_UPDATE_DATA, except that it can only load 4096 bytes, as it is intended to encrypt the VMSA. In addition, it also initializes the VMCB. While not enforced, this is intended to be called only once. Again, the launch digest is updated with the loaded data.

The third and final command, LAUNCH_MEASURE, generates a launch measurement and transfers the VM to the LSECRET state. The measurement consists of a 128-bit nonce MNONCE and a 256-bit HMAC MEASURE, that is calculated as follows:

1. Replace launch digest (LD) with $\mathrm{hash}(\texttt{LD})$
2. Calculate

$$\begin{aligned}\mathrm{HMAC}(&\texttt{0x04} \,\|\, \texttt{API\_MAJOR} \,\|\, \texttt{API\_MINOR} \\ &\|\, \texttt{BUILD} \,\|\, \texttt{POLICY} \,\|\, \texttt{LD} \\ &\|\, \texttt{MNONCE}, \texttt{TIK})\end{aligned}$$

MNONCE is generated by the SP, API_MAJOR and API_MINOR and BUILD specify the version of the firmware on the SP. POLICY is the configuration structure that was sent by the guest owner in the UNINIT state.

Next, the HV sends the launch measurement to the guest owner, in order to prove that it did not manipulate the initial content. It is assumed that the guest owner and the HV/cloud service provider negotiated the initial content of the VM, e.g., that the guest owner stated that they want a specific UEFI version to be loaded. Thus, the guest owner has all the information required to compute the HMAC themselves and compare it to the value they received.

After successfully checking the launch measurement, the guest owner can be sure that the initial memory content matches their specification. Since, on startup, the VM treats any memory as encrypted, it is unlikely that the HV can achieve any meaningful manipulation of the VM's code and data by tampering with its memory. The only possibility for the HV to encrypt data with the VM's key is by using the designated `LAUNCH_UPDATE_*` commands, but, as already explained, this has the side effect of updating the launch digest and thus changing the HMAC sent in the attestation report, allowing detection by the guest owner. As only the SP and the guest owner know the TIK used to key the HMAC, the HV cannot produce valid HMACs itself.

3.4 `LSECRET` state

After the VM has transitioned into the `LSECRET` state, two commands become available: `LAUNCH_SECRET` and `LAUNCH_FINISH`. The `LAUNCH_SECRET` command again allows to encrypt data with the VM's VEK. However, contrary to the previous commands, the data passed to the command now is encrypted with the TEK and integrity protected by an HMAC keyed with the TIK. Both keys are only known to the SP and the guest owner. If the integrity check fails, the command aborts. The guest owner can use this mechanism to safely send confidential data (e.g., disk encryption keys) to the VM, while using the HV as a proxy. The HV could refuse to relay the data to the SP, but it can neither manipulate the data nor call the command with self-generated data, as it does not know the TIK needed to pass the HMAC check.

Finally, the `LAUNCH_FINISH` command transitions the VM into the `RUNNING` state, indicating that the VM is ready to be started. The `LAUNCH_SECRET` and `LAUNCH_FINISH` commands are disabled afterwards.

4 Attacker Model

The attacker model is in line with SEV's security model: The attacker controls the hypervisor, and is able to modify arbitrary physical memory and run or pause the VM according to their wishes. However, they are not able to read or modify the current register state and program counter of the VM, as its state is encrypted using SEV-ES.

They know the initially launched code, since that needs to be available in plaintext in order to be loaded into the VM. We assume that the attestation is working in so far that the attacker has to actually load and attest the supplied initial code image, and cannot simply replace it with their own. We also assume that the attestation protocol is carried

out correctly, such that the guest owner is assured that supposedly the correct image was loaded, and subsequently launches the virtual machine.

The attacker is not able to read or modify encrypted disk images without knowing the corresponding key. Finally, we consider the SP itself to be secure.

5 Exploiting SEV's Permutation Agnostic Launch Measurement

Given the VM attestation process laid out in Section 3, we show how an attacker can deviate from the intended startup process in order to make the VM execute arbitrary code, which corresponds to a full break of confidentiality and integrity.

In a first step, we show that SEV's launch measurement can be tricked into producing the same measurement for any blockwise permutation of the initial VM content. We illustrate how an attacker can use this flaw to construct an encryption/decryption/code execution gadget, that runs within the VM, but does not change its launch measurement and thus cannot be detected by the guest owner. Finally, in Section 6, we discuss the implications of our attack for the transition from initially attested code to code residing on a virtual hard disk, and demonstrate how we can use our attack to leak secrets.

5.1 Breaking the Launch Measurement

First, we show how a malicious HV can abuse a flaw in the launch process to change the semantics of the loaded data without changing the launch measurement.

As described in Section 3, the HV uses the `LAUNCH_UPDATE_DATA` command to load and encrypt the initial memory content of the VM. The command takes a 16-byte aligned HPA `PADDR` and a multiple of 16 bytes as length `L`, and then in-place encrypts `L` bytes starting at `PADDR` with the VM's VEK. In addition, the command updates the launch digest which is later used in the launch measurement.

In our experiments, we observed that the content of the launch digest is neither influenced by the HPAs passed to `LAUNCH_UPDATE_DATA`, nor by the used block size and the resulting varying number of calls to the command. Instead, the encrypted data is simply "appended" to the launch digest. While the official documentation is unclear at this point, we suspect that the launch digest internally manages a SHA-256 hash state, which is updated each time after a certain amount of data was inserted.

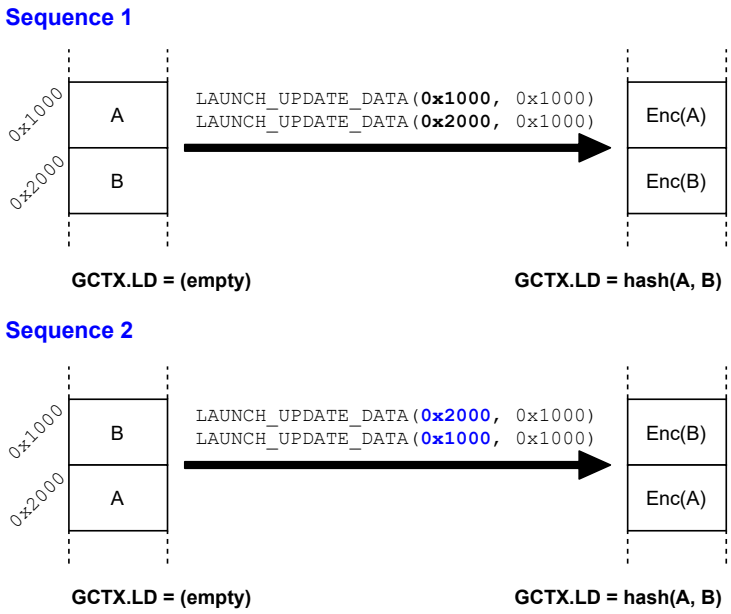


Figure 2: Two encryption sequences, yielding the same launch measurement GCTX.LD for different orders of memory blocks. In the first sequence, the memory pages A (address 0x1000) and B (address 0x2000) are encrypted in memory order, i.e., LAUNCH_UPDATE_DATA is first called for block A, then for block B. In the second sequence, the blocks are swapped in memory: A now resides at address 0x2000, while B is at address 0x1000. By changing the order of calls to LAUNCH_UPDATE_DATA, we are able to acquire the same value for GCTX.LD in sequence 2 as for the "correct" ordering in sequence 1. The guest owner thus has no means for distinguishing which sequence has been used by the HV.

This implies that the HV can change the memory layout of the loaded data without any impact on the resulting hash value, as long as it makes sure that the order in which the data is passed to LAUNCH_UPDATE_DATA matches the original order. The modified ordering is illustrated in Figure 2.

5.2 Constructing Malicious Code Gadgets

We can now use our observations to construct malicious code gadgets, solely by moving around 16-byte blocks and triggering interaction with the HV.

The general idea is very similar to the approach presented in [44], where the authors leverage control over the first and last bytes of 16-byte blocks to stitch together a sequence of "payload" instructions and direct jumps, which they subsequently use to build an encryption oracle within the VM. However, we cannot change a block's content here, as this would be detected during attestation.

We first scan the binary of the initial VM content, which, in our case, can be split up into 230'000 16-byte blocks, for the instructions that we want to execute in our gadgets. For this, we are not bound to the originally intended decoding order: As x86 instructions have variable length and are not prefix-free, starting to decode the binary with different offsets can lead to different valid instructions. On the downside, this also means that decoding may fail because it encounters an invalid instruction encoding. To address this, we only look for "payload" instructions which reside at the end of a block or are followed by a direct jump, such that we can proceed to the next block.

Finally, we analyze the control flow of the original program to find a location where we can place our gadget, so it is executed at some point during the startup of the VM. We also make sure that our changes to the block ordering do not destroy the code needed to boot up the VM to the point where our gadget is entered.

5.3 Encryption/Execution Gadget

We can now use this block chaining technique to build a code gadget that enables the HV to encrypt (and decrypt) arbitrary data with the VEK and inject it into the VM.

For this, we assume that the VM is started with the default OVMF UEFI provided by AMD as initial memory content. Note that the ideas presented here are also applicable to other UEFI implementations that support SEV.

The encryption/execution gadget is constructed in two stages:

1. Permute the initial VM content, creating a gadget that maps the stack to an unencrypted shared memory page;

2. Use the control over the stack to construct a ROP gadget that copies data from the unencrypted shared page to encrypted memory and execute it as code. This code may later copy decrypted memory back to the shared page, or can be used to conduct other, more complex attacks.

Stage 1 In the first stage, we want to set the VM's stack (i.e., its `rsp` register) to an unencrypted memory page, to allow manipulation by the HV. Until reaching either long mode or legacy Physical Address Extension (PAE) mode, the VM's memory accesses are unconditionally treated as encrypted. Afterwards, the `C` bit in the page table controls whether a page is accessed in encrypted or unencrypted mode. The only exception are page table walks and instruction fetches, which are always treated as encrypted [1, Sec 15.43.4, 15.34.5].

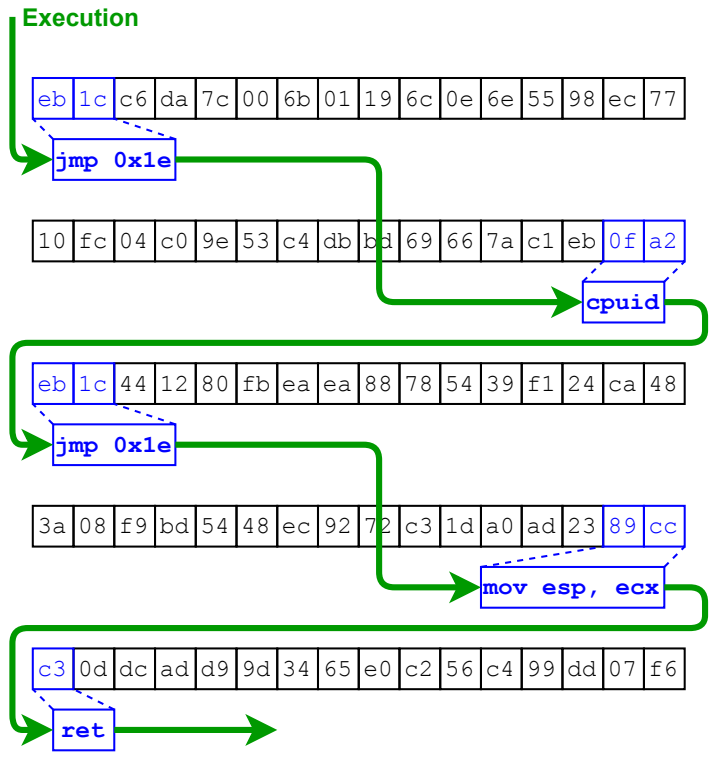


Figure 3: Sequence of 16-byte blocks for setting the stack pointer to a HV-controlled address. The shown 16-byte blocks were taken from various places in the initial code image and moved to an address which is reached by the execution flow. The `jmp` instructions allow us to chain several blocks and skip potential junk bytes in between. After the stack pointer has been changed, the `ret` statement reads its return address from HV-controlled memory and thus triggers a ROP chain.

After startup, OVMF quickly progresses to long mode. While constructing its long mode page tables, OVMF also sets up a shared page for the GHCB protocol [2], which, under SEV-ES, is required to handle the emulation of instructions that need to share data with and/or receive data from the HV (c.f. Section 2.1).

To load the address of the shared GHCB page into the `rsp` register, we opted for the following payload instruction sequence:

- `cpuid`
  Fills the `eax`, `ebx`, `ecx` and `edx` registers with processor feature information. As shown in previous work [44], one can abuse that this instruction is emulated by the HV and fill the `ecx` register with the virtual address of the shared GHCB page.

- `movesp, ecx`
  Updates the stack pointer with the address of the shared page. Note the usage of 32-bit registers: This has the advantage of having a shorter instruction encoding than the 64-bit equivalent, while still being sufficient, as in OVMF the virtual address of the shared page is hardcoded to a small constant.

- `ret`
  Starts the ROP chain. As the stack pointer now points to the unencrypted shared page, the HV can place arbitrary return addresses (and other values) on the stack, which allows to conduct a classic ROP attack.

The resulting block chain is illustrated in Figure 3. To ensure the right timing for sending the manipulated `cpuid` register values, the HV simply counts the number of emulated `cpuid` instructions, which is deterministic in the executed OVMF code.

Stage 2 Coming from Stage 1, we now have full ROP capabilities.

In order to load additional data into the VM and execute it as code, we need to construct a ROP chain that copies data from the unencrypted stack to a memory location that is marked as encrypted. Then, we jump to that address using a `ret` instruction. This indirect approach of encrypting the code before execution is necessary, since, as mentioned in Stage 1, instruction fetches always assume that the underlying memory is encrypted. We abuse that, at the time of gadget injection, OVMF's page tables have all pages marked as readable/writable, without having a "no execute" bit. This simplifies our attack, since we do not have to build a ROP chain for modifying the page tables first.

Our resulting ROP chain is illustrated in Figure 4. It only needs three gadgets, and allows us to write 8 bytes to an arbitrary 64-bit address. We can then reuse the chain to write complex new code into the VM. In Section 6, we show how this code can be used to leak disk encryption keys.

```
(1) 0x00000000fffd1ad0    pop rax
                          pop rbx
                          ret

(2) 0x00000000fffcef21    pop rdx
                          ret

(3) 0x00000000fffcfde6    mov qword ptr [rax], rdx
                          xor eax, eax
                          pop rbx
                          pop rbp
                          pop r12
                          ret
```

Figure 4: ROP chain for writing data to VM memory. The ROP chain consists of three gadgets. Each gadget begins with a payload instruction (highlighted blue) and ends with a return statement, potentially with a few other instructions in between. Gadget (1) loads an unencrypted 8-byte address from the stack and writes it into `rax`. Gadget (2) loads an unencrypted 8-byte value from the stack and writes it into `rdx`. Finally, gadget (3) stores the value from `rdx` at the address pointed to by `rax`. The memory write in (3) triggers encryption of the data stored in `rdx`, as the address in `rax` points to encrypted memory.

In summary, we have seen that SEV's launch measurement mechanism is flawed, as it only attests that an arbitrary 16-byte granular permutation of the initial data has been loaded. Next, we have demonstrated the creation of malicious code gadgets just by swapping around 16-byte blocks. We have shown an instantiation of such a gadget, that maps the VM's stack to a shared page and employs a ROP attack to write additional data and execute it as code.

6 Attack Case Study

With our encryption and code execution gadget from Section 5.2, the attacker gains control over the initially executed code, and is able to insert their own. In the following, we highlight critical moments in the startup of a SEV-secured VM, and show how we can use our attack to take it over.

6.1 Experimental Setup

Our experiments are performed on a second generation AMD EPYC Processor 7232P. The firmware of the SP is version 24 build 0a [7] (most recent at time of writing). On the HV side, we use Linux Kernel 5.6 from the official AMD repository [4] (extended with our attack code) and QEMU [36] to start the VM. QEMU was extended with

AMD's SEV-ES patches [5] and the proposed patches for the secret injection mechanism from [14] (see Section 6.3). Inside the SEV-ES victim VM, we run OVMF [40] and Grub [21]. Both were extended with the secret injection mechanism patches from [14]. We provide our proof-of-concept code alongside with the used software online at `https://github.com/UzL-ITS/undeserved-trust`.

6.2 Trust Gap

For simplicity and scalability, only a small part of the VM's code is attested. In most cases, it suffices to attest a tiny initial code image, which takes owner-supplied secrets to load and decrypt a much larger encrypted disk image, which in turn contains the operating system and the processed data. The operating system can be considered trusted, as the attacker should not be able to modify the encrypted disk image without being in possession of the key.

The primary challenge is bridging the trust gap between the attested initial code image, which is available in plaintext, and the encrypted operating system: The guest owner needs to be able to supply secret information for decrypting the disk image, without an attacker being able to learn those secrets.

If an attacker gains access to a disk encryption key, they also gain unthrottled read/write access to all of the VM's data, even after the VM was shut down. They can abuse this access to extract secret data or manipulate the operating system.

6.3 Securely Injecting Secrets

To address this challenge, SEV offers the `LAUNCH_SECRET` command, which allows the guest owner to inject arbitrary secret values into the VM.

Since there is not yet a standardized toolchain, we focus on the proposed launch flow from [14], which has some of its patches already merged into the respective upstream repositories. Note that any other possible launch process will also need to bridge the aforementioned trust gap, and will thus be quite similar to the launch flow discussed here.

The proposed launch flow works as follows: The initial attested code image consists of both the OVMF UEFI binary and the Grub bootloader, which thus cannot be modified by a malicious HV. The UEFI performs the initial startup, and then transfers control to the bootloader, which in turn unlocks an encrypted disk image and boots the contained Linux kernel.

```
5e      pop rsi    ; source address
5f      pop rdi    ; destination address
59      pop rcx    ; n
f3 a4   rep movsb  ; copy n bytes
f4      hlt        ; halt VM
```

Figure 5: Code gadget for copying data. The register values are passed via the stack, to minimize the size of our injected code: By using the string copy instruction `repmovsb`, we can fit the entire gadget into 6 bytes, so we only need a single execution of the ROP chain from Figure 4. Since we only leak the secret, we halt the VM after copying the data; however, if we wanted to copy more data, we could replace the `hlt` instruction by a `ret` instruction and execute the gadget multiple times with different parameters.

Both OVMF and Grub have been adjusted to respect the secret injection mechanism: At build time, OVMF includes a configuration table, which specifies the Guest Physical Address (GPA) where the HV (QEMU) should inject the secret. While preparing the VM startup, the HV scans the OVMF binary, locates the configuration table and subsequently injects the secret at the indicated address. OVMF then passes the secret to the Grub bootloader, which uses it to unlock the disk.

6.4 Leaking the Disk Encryption Key

Given our attack primitives from Section 5, leaking the disk encryption key is quite straightforward. We already know the length of the secret data, since the HV is responsible for receiving the encrypted secret from the guest owner and forwarding it to the SP via `LAUNCH_SECRET`, which expects a public length parameter. In addition, we know the GPA of the secret from OVMF's configuration table. Using our ROP chain, we can now inject a small code gadget which loops over the secret data and copies it into shared memory. The code gadget is shown in Figure 5.

We halt the VM after extracting the secret; however, to avoid detection by the guest owner, we could also use our copy gadget to repair the code which we damaged by our block moving approach. Then, potentially after injecting further spy code into the previously encrypted disk image, we resume the boot process.

7 Mitigations

Our attacks severely undermine the validity of AMD SEV's remote attestation. Unfortunately, it is not possible to fully mitigate our attack with the current capabilities of SEV-ES,

due to the lack of page remapping protection, which will only be available with SEV-SNP on upcoming 3rd generation EPYC processors. Nonetheless, we discuss changes that could be applied to systems limited to SEV-ES, to make exploitation harder.

7.1 Increasing the Block Size

A simple countermeasure, which renders creating an exploit by reordering code blocks much harder, is to increase the minimal size of each measured block.

Given, e.g., our OVMF binary of 3.5 MB, a block size of 16 bytes yields around 230'000 blocks; for a block size of 4 kB, this number shrinks to merely around 900, greatly reducing a malicious HV's ability to find a block ordering that produces meaningful code, although it does not completely mitigate it. Note that `LAUNCH_UPDATE_DATA` already supports large block sizes: To improve performance, the corresponding kernel code tries to process contiguous physical memory blocks, which are chosen as large as possible.

Currently the protocol does not support specifying a certain block size or including it in the launch digest. However, it would be possible to implement a fixed block size without any changes to the protocol, by hardcoding a size of 4 kB (page) or even 2 MB (huge page) directly in the SP firmware. This way, the guest owner just needs to require the corresponding firmware version during attestation. Since the version check is already included in the protocol and the HV only has to update the SP firmware, this countermeasure is rather cheap. Only the SP firmware and client applications which verify the launch digest have to be changed.

7.2 Potential Changes to the Attestation Protocol

In order to further increase the power of the proposed countermeasure or even fully close the vulnerability, one may also include the physical addresses in the measurement and attestation, or increase the size of the measured blocks and add the block size to the measurement hash. This could be achieved by computing

$$h_i = \mathrm{hash}(h_{i-1} \,\|\, \mathrm{HPA}_i \,\|\, \mathrm{data}_i) \quad \text{or}$$

$$h_i = \mathrm{hash}(h_{i-1} \,\|\, \mathrm{block\_size}_i \,\|\, \mathrm{data}_i)$$

on each call i to `LAUNCH_UPDATE_DATA`, for a total of n calls, and submitting the list of addresses or block sizes along the measurement h_n. Both of these changes require changing the protocol of the remote attestation, since the address list or list of block sizes must be sent along the measurement itself.

However, both approaches are intrinsically limited by the underlying hardware assumptions and address mappings which are in place during virtualization. The HV can still legally reorder physically contiguous 4 kB pages, since it controls the mapping of host physical to guest physical memory addresses through its control over the Nested Page Table (NPT). I.e., the hypervisor is capable of performing page remapping attacks, as already exploited by Morbitzer et al. [35]. Thus, both approaches are limited to assure the correct order within and for the size of one memory page, which can already be achieved with fixing the block size to 4 kB (2 MB) as described in the previous section. The only advantage of including the physical addresses in the measurement is that we can ensure the order inside a 4 kB page, while still allowing 16 byte blocks as the smallest block size.

In conclusion, the attestation process is in need of fixing the loaded binary to addresses within the *guest's* address space: Adding the guest physical address, instead of the host physical address, to the measurement, and assuring that a remapping between guest physical address and host physical address after the initial allocation will be detected by the secure processor, would completely close the vulnerability described in this work. However, SEV-ES does not allow to detect a page remapping and thus only allows for a partial mitigation.

7.3 Changes in SEV-SNP

AMD SEV-SNP [3] is an upcoming extension to SEV-ES, which is only supported on the third generation of EPYC processors. As those only become available after the submission deadline, the following is solely based on the documentation [1, 3, 8].

One of the major changes in SEV-SNP is the introduction of the Reverse Map Table (RMP). The RMP is an additional page table, indexed by the HPA of a page. It adds several new attributes, mostly for distinguishing VM and HV pages, such that the HV cannot write to guest pages. In addition, the RMP contains the GPA of VM pages and can be used to ensure a one-to-one mapping between GPA and HPA to prevent remapping attacks. In contrast to traditional page tables, the HV does not have full control over the RMP, as it must use hardware- and firmware-mediated ways to access it.

While the general flow of the launch process does not appear to have changed, there are two essential changes to the `LAUNCH_UPDATE_DATA` command, preventing our attack. The first one implements the idea that we also proposed in Section 7.1: The HV must either pass a 4 kB or a 2 MB page to the command (however, 2MB pages are internally treated as multiple 4 kB pages). The second change is the calculation of the launch digest. The

hash is now finalized after each call to the launch command and calculated as follows:

$$h_i = \mathrm{hash}(h_{i-1} \parallel \mathrm{hash}(\mathrm{data}_i) \parallel \mathrm{block\_size}_i \parallel \ldots \parallel \mathrm{GPA}_i),$$

where "..." represent additional fields that we omitted for brevity. Due to hash finalization after each call, the launch digest h_n now reflects the amount of `LAUNCH_UPDATE_DATA` calls used to load the data. In addition, the `GPA` field is of special interest, because it includes the memory layout, as it is observed by the guest, in the measurement. According to the documentation, the GPA value is computed by the SP and also stored in the RMP. Furthermore, the page is marked as a guest page. The GPA value is enforced, because the hardware page table walker checks that the GPA to HPA mapping in the NPT matches the one in the RMP.

8 Related Work

Since the initial release of AMD's memory encryption technology, first SME and later SEV, there has been a wide range of attacks against its security guarantees. Hetzelt et al. [23] exploited the unencrypted register state of the first version of SEV to construct simple encryption/decryption oracles. In addition they explored memory replay attacks.

Du et al. [19] unveiled the encryption mode, tweak values and the resulting lack of integrity protection of SME on Ryzen processors, which is closely related to SEV on EPYC processors. They used this knowledge in addition with a network service inside the VM to create an encryption oracle on a simulated version of SEV, which was not yet available.

Li et al. [30] used the lack of integrity protection combined with the knowledge of the tweak values to construct encryption as well as decryption oracles. For this, they exploited the fact that Direct Memory Access (DMA) operations issued by the VM are mediated by the HV through shared memory pages.

Wilke et al. [44] extended the analysis of the encryption mode and the tweak values to first generation EPYC and EPYC-embedded CPUs, unveiling an updated encryption mode. They showed how to abuse the missing integrity protection combined with knowledge of the tweak values, to bootstrap an encryption oracle from malicious code gadgets, solely by moving around ciphertext blocks in memory. However, the attack does not work on second generation EPYC CPUs, as these feature an enhanced randomization of tweak values. While our attack follows a similar approach of reordering memory blocks, it does not rely on the encryption mode at all, and is therefore also applicable to the currently available second generation EPYC CPUs.

Morbitzer et al. [35] leveraged the HV's control over the NPT as well as the page fault side channel to construct a decryption oracle. Similar to Du et al. [19], they require a service running in the VM. In their follow-up paper [34], they showed how to locate pages containing secrets, like OpenSSH keys, in the VM.

Werner et al. [43] showed that it is possible to use the unencrypted register values in the first SEV version to reconstruct the code executed in the VM. Furthermore, they showed how to use Instruction Based Sampling, a performance counter subsystem, to fingerprint code executed in SEV-ES VMs.

Radev et al. [38] described multiple attacks, exploiting insufficient value sanitization at the HV to VM boundary. For example, they showed how to trick the VM into treating arbitrary memory accesses as Memory Mapped I/O (MMIO), as well as into using malicious virtualized cryptographic accelerators provided by the HV. In addition, they demonstrated how faking `cpuid` results can be used to corrupt the VM page tables to mark all pages as unencrypted. They then used the unencrypted stack to launch a ROP attack, similar to our stage 2 gadget. However, in contrast to our attack, the page table manipulation used by them can be detected by a simple software countermeasure, as described in their paper.

Li et al. [29] demonstrated that the "security by crash" philosophy behind AMD's use of the Address Space Identifier (ASID) for mapping VMs to their memory encryption keys is flawed, as a malicious HV can swap the ASIDs of an attacker VM and the victim VM to leak limited amounts of data.

Buhren et al. [15] explored another attack vector by analyzing the firmware loading mechanism of the SP. They discovered a bug allowing them to load customized firmware on the SP, breaking the hardware root of trust.

9 Conclusion

In this work, we have shown that the current attestation mechanism of SEV has a significant flaw, as it allows the HV to reorder blocks of the initially loaded image without influencing the launch measurement, leaving the guest owner unaware of our attack. We have been able to use this vulnerability to redirect execution and inject arbitrary code into the encrypted VM, giving us full control over its execution flow. Moreover, we have shown how this vulnerability in the remote attestation allows us to extract secret data and conduct other attacks, like manipulating the booted operating system.

The attack described in this work undermines the validity of AMD SEV's remote attestation and thus its trustworthiness as a trusted execution environment. Especially,

when authenticated and confidential execution in otherwise untrusted environments are required, additional means for verifying the authenticity of the loaded and executed software should be taken into consideration, until the vulnerability is fixed.

We have described possible changes to the firmware of the secure processor, allowing for a simple and reasonably secure mitigation which could be rolled out by means of a firmware update to existing EPYC processors of the first and second generation. However, as argued in Section 7, we do not think that it is possible to completely close this vulnerability with the capabilities of SEV-ES. If the information provided in the SEV-SNP white paper holds, full protection should only become available with the third generation EPYC processors.

Acknowledgments

This work was partially supported by the DFG grants 439797619 and 427774779.

References

[1] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Rev. 3.36. 2020-10.

[2] AMD. *SEV-ES Guest-Hypervisor Communication Block Standardization*. `https://developer.amd.com/wp-content/resources/56421.pdf`. 2020.

[3] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. `https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf`. 2020-01.

[4] AMD. *Github - AMDESE/linux*. `https://github.com/AMDESE/linux`. On branch "sev-es-5.6-v3" at commit "d0d7b0f09359da7316aee077bac92ec21a1a047b".

[5] AMD. *Github - AMDESE/qemu*. `https://github.com/AMDESE/qemu.git`. branch "sev-es-v12", commit "1cfb0be5fdad55948f42f9056dfc16dc435099cf".

[6] AMD. *Secure Encrypted Virtualization API*. `https://www.amd.com/system/files/TechDocs/55766_SEV-KM_API_Specification.pdf`. Rev. 3.24. 2020-04.

[7] AMD. *SEV firmware for Rome*. `https://developer.amd.com/wordpress/media/2013/12/amd_sev_fam17h_model3xh_0.24b0A.tar.gz`. Version 0.24b0A. 2021.

[8] AMD. *SEV Secure Nested Paging Firmware ABI Specification*. Tech. rep. 56860, Rev. 1.55. AMD, 2023-09.

[9] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. "Innovative technology for CPU based attestation and sealing". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. 7. 2013.

[10] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. "SCONE: Secure Linux Containers with Intel SGX". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[11] Asylo Project. https://github.com/google/asylo. Accessed: 2021-01-26.

[12] Alex Bennée. *Anatomy of a Boot, a QEMU perspective*. https://www.qemu.org/2020/07/03/anatomy-of-a-boot/. accessed: 2021-01-29.

[13] James Bottomley. *[edk2-devel] [PATCH v3 0/6] SEV Encrypted Boot for Ovmf*. https://www.redhat.com/archives/edk2-devel-archive/2020-November/msg01247.html.

[14] James Bottomley. *Deploying Encrypted Images for Confidential Computing*. https://blog.hansenpartnership.com/deploying-encrypted-images-for-confidential-computing. Accessed: 2021-01-22. 2020-12.

[15] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. "Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019. DOI: 10.1145/3319535.3354216. URL: https://doi.org/10.1145/3319535.3354216.

[16] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[17] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00089. URL: https://doi.org/10.1109/SP40000.2020.00089.

[18] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: *IACR Cryptol. ePrint Arch.* (2016), p. 86. URL: http://eprint.iacr.org/2016/086.

[19] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. "Secure Encrypted Virtualization is Unsecure". In: *CoRR* abs/1712.05090 (2017). arXiv: 1712.05090. URL: http://arxiv.org/abs/1712.05090.

[20] Enarx Project. https://github.com/enarx/enarx. Accessed: 2021-01-26.

[21] GNU Project. *GNU GRUB*. https://www.gnu.org/software/grub. Accessed: 2021-01-22.

[22] Google. *Google Cloud Confidential Computing / Confidential VMs*. https://cloud.google.com/compute/confidential-vm/docs/about-cvm. Accessed: 2021-01-26.

[23] Felicitas Hetzelt and Robert Buhren. "Security Analysis of Encrypted Virtual Machines". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*. 2017. DOI: 10.1145/3050748.3050763. URL: https://doi.org/10.1145/3050748.3050763.

[24] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cache Attacks Enable Bulk Key Recovery on the Cloud". In: *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. 2016. DOI: 10.1007/978-3-662-53140-2_18. URL: https://doi.org/10.1007/978-3-662-53140-2%5C_18.

[25] Intel. *Intel Trust Domain Extensions*. https://cdrdv2.intel.com/v1/dl/getContent/690419. Accessed on 19.09.2024. 2023-02.

[26] Intel. *Intel®Software Guard Extensions (Intel®SGX) - Developer Guide*. https://download.01.org/intel-sgx/sgx-linux/2.11/docs/Intel_SGX_Developer_Guide.pdf. Rev. 2.11. 2020-08.

[27] David Kaplan. *Protecting VM Register state with SEV-ES*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf. 2017-02.

[28] David Kaplan, Jeremy Powell, and Wolle. *AMD Memory Encryption*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf. 2021-10.

[29] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. "CrossLine: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3485253. URL: https://doi.org/10.1145/3460120.3485253.

[30] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. "Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan.

[31] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015. DOI: 10.1109/SP.2015.43. URL: https://doi.org/10.1109/SP.2015.43.

[32] Microsoft. *Azure confidential computing*. https://docs.microsoft.com/en-us/azure/confidential-computing. Accessed: 2021-01-26.

[33] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat.

[34] Mathias Morbitzer, Manuel Huber, and Julian Horsch. "Extracting Secrets from Encrypted Virtual Machines". In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX , USA, March 25-27, 2019*. 2019. DOI: 10.1145/3292006.3300022. URL: https://doi.org/10.1145/3292006.3300022.

[35] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. "SEVered: Subverting AMD's Virtual Machine Encryption". In: *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*. 2018. DOI: 10.1145/3193111.3193112. URL: https://doi.org/10.1145/3193111.3193112.

[36] QEMU Project. *Github - qemu*. https://github.com/qemu/qemu. at commit "3dcfd4e3f285cd69d7cf581d3a688e421d28e07e".

[37] QEMU Project. *QEMU*. https://www.qemu.org/. Accessed: 2021-01-22.

[38] Martin Radev and Mathias Morbitzer. "Exploiting Interfaces of Secure Encrypted Virtual Machines". In: *Reversing and Offensive-oriented Trends Symposium (ROOTS)*. 2020.

[39] Mark Russinovich. *DCsv2-series VM now generally available from Azure confidential computing*. https://azure.microsoft.com/en-us/blog/dcsv2series-vm-now-generally-available-from-azure-confidential-computing/. 2020-04.

[40] TianoCore. *Github - OVMF*. https://github.com/tianocore/edk2.git. at commit "6c5801be6ef36e35f0b4ff906a4c99d68ca6f69a".

[41] TianoCore. *OvmfPkg*. https://github.com/tianocore/edk2/tree/master/OvmfPkg. Accessed: 2021-01-21.

[42] Unified EFI Forum. *Unified Extensible Firmware Interface (UEFI) - Specification*. https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_final.pdf. Version 2.8. 2019-03.

[43] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. "The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*. 2019. DOI: 10.1145/3321705.3329820. URL: https://doi.org/10.1145/3321705.3329820.

[44] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. "SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00080. URL: https://doi.org/10.1109/SP40000.2020.00080.

[45] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-VM side channels and their use to extract private keys". In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 2012. DOI: 10.1145/2382196.2382230. URL: https://doi.org/10.1145/2382196.2382230.

6

A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP

Publication

Mengyuan Li*, **Luca Wilke***, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu and Yinqian Zhang. In *IEEE Symposium on Security and Privacy (SP)*, 2022.

Contribution

Mengyuan and I are co-first authors. We independently worked on the core ideas for the attacks and got merged by AMD during the disclosure process. The theoretical countermeasure were conceived in collaboration. I implemented the collision attack and the proof-of-concept countermeasure for Linux.

Outline

A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP

Mengyuan Li[1]*, **Luca Wilke[2]***, Jan Wichelmann[2], Thomas Eisenbarth[2], Radu Teodorescu[1] and Yinqian Zhang[3]

[1]The Ohio State University
[2]University of Lübeck
[3]Southern University of Science and Technology

Hardware-assisted memory encryption offers strong confidentiality guarantees for trusted execution environments like Intel SGX and AMD SEV. However, a recent study by Li et al. presented at USENIX Security 2021 has demonstrated the CipherLeaks attack, which monitors ciphertext changes in the special VMSA page. By leaking register values saved by the VM during context switches, they broke state-of-the-art constant-time cryptographic implementations, including RSA and ECDSA in the OpenSSL.

In this paper, we perform a comprehensive study on the ciphertext side channels. Our work suggests that while the CipherLeaks attack targets only the VMSA page, a generic ciphertext side-channel attack may exploit the ciphertext leakage from any memory pages, including those for kernel data structures, stacks and heaps. As such, AMD's existing countermeasures to the CipherLeaks attack, a firmware patch that introduces randomness into the ciphertext of the VMSA page, is clearly insufficient. The root cause of the leakage in AMD SEV's memory encryption—the use of a stateless yet unauthenticated encryption mode and the unrestricted read accesses to the ciphertext of the encrypted memory—remains unfixed. Given the challenges faced by AMD to eradicate the vulnerability from the hardware design, we propose a set of software countermeasures to the ciphertext side channels, including patches to the OS kernel and cryptographic libraries. We are working closely with AMD to merge these changes into affected open-source projects.

1 Introduction

For years, the main obstacle to cloud adoption has been a lack of trust in Cloud Service Providers (CSP). The concept of confidential Virtual Machine (VM) has been enabled by an emerging security feature in modern CPUs, dubbed Trusted Execution Environment

(TEE), which removes the need to trust the CSP [15]. Aiming at providing data-in-use protection, confidential VM uses hardware-based memory encryption to protect the integrity and the confidentiality of VMs against both physical access attacks and privileged software-level attacks. Another key benefit of confidential VM is that any VM can be deployed as confidential VM on systems that support them, without costly adaption and rewriting that is necessary to turn applications into secure enclaves [12]. Due to the enormous market potential, all main processor vendors have released or are working on releasing confidential VM features in their server CPU lines, including AMD Secure Encrypted Virtualization (SEV) [23], Intel Trust Domain Extension (TDX) [19], and ARM Confidential Compute Architecture (CCA) [8].

Currently, only AMD's confidential VM solution—AMD SEV—is available and has been deployed in public clouds [15, 29]. Since its first deployment, SEV has been exhaustively analyzed by the security community. Due to the powerful adversarial scenario of a malicious hypervisor, several weaknesses have been found, including unauthenticated encryption [10, 14, 36], Nested Page Table (NPT) remapping [17, 30, 31], unprotected I/O [26], and unauthorized Address Space Identifiers (ASID) [25]. With the newest version of SEV—the recently released SEV-SNP (Secure Nested Paging [3])—most of the attacks are now mitigated.

The only software-based attack that still applied to SEV-SNP is CIPHERLEAKS [27], a novel side-channel attack where a malicious hypervisor can steal the secret keys of RSA and ECDSA algorithms in the OpenSSL implementation by monitoring the guest VM Save Area (VMSA). Specifically, SEV's memory encryption engine adopts a deterministic XOR-Encrypt-XOR (XEX) mode of operation. For each physical address, the same 128-bit plaintext block is always encrypted to the same ciphertext block during the life cycle of the VM. Meanwhile, whenever there is a guest-host world switch, register values are encrypted and then stored in the VMSA. With the power of read access to the guest VM's VMSA area, the malicious hypervisor can continuously monitor and record the ciphertext of encrypted registers. The authors show that the ciphertext of certain registers (*e.g.*, RAX) can be used to inspect inner execution states of cryptographic algorithms and eventually reveal the private key or secrets.

Due to its severity, AMD recently released a microcode patch (MilanPI-SP3_1.0.0.5) [2] to mitigate the CIPHERLEAKS attacks. The microcode patch enables the 3rd generation AMD EPYC processors (Milan series) to include a nonce into the encryption of the VMSA area, such that the link between the plaintext and the ciphertext is broken. As such, CIPHERLEAKS attacks against register values in the VMSA are no longer feasible. Note that the patch only changes the encryption of the VMSA, while the remaining memory space of the VM is still protected with the same deterministic XEX encryption as before.

In this paper, we perform a comprehensive study on the exploitability of leakage caused by ciphertext in encrypted VM memory and try to answer the question:

> *Are current cryptography implementations still safe when an attacker has access to the ciphertext of the encrypted memory?*

We broadly group ciphertext side channel attacks into two categories: *the dictionary attack* and *the collision attack*. We show that these two classes of attacks can be applied to general memory regions during cryptographic activities, including kernel data structures, stacks, and heaps, which all lead to key leakage. Most main cryptography libraries (including OpenSSL, WolfSSL, GnuTLS, OpenSSH, and libgcrypt) are shown to be vulnerable against the ciphertext side channel.

Contribution. The contributions of this paper can be summarized as follows:

- Systematically studies the ciphertext side channel in the entire memory of SEV-protected VMs. It shows that the ciphertext side channel can be exploited in *all* memory regions, including kernel structures, stacks, and heaps.
- Presents end-to-end ciphertext side-channel attacks against the ECDSA implementation of the OpenSSL library. Other main cryptography libraries (including OpenSSL, WolfSSL, GnuTLS, OpenSSH, and libgcrypt) are also shown to be vulnerable to the ciphertext side channel.
- Discusses both hardware and software countermeasures. Presents a kernel patch to mitigate ciphertext side channels caused by kernel structures. The ciphertext side channel can be mitigated when adopting the kernel patch together with software fixes for cryptographic libraries.

Responsible disclosure. We disclosed the generic ciphertext side-channel attacks on kernel data structures, heaps, and stacks to the AMD SEV team in August 2021. Henceforth, we provided more supplementary materials via email communications. AMD has acknowledged the vulnerability and had several discussions with us about potential countermeasures and stated interest in a kernel level fix. While hardware countermeasures might not be feasible in the near future for both performance and design concerns, AMD assisted us with the development of the software countermeasures, including both kernel patches (Section 6) and helping us get connected to other projects like OpenSSL.

We also disclosed the vulnerability on the code level to the communities of cryptography libraries (including OpenSSL, WolfSSl, GnuTLS, OpenSSH and libgcrypt). At the time of writing, we had received feedback from both OpenSSL and WolfSSL. They both

acknowledged the concerns and recognized the necessity of addressing this vulnerability from software. WolfSSL has already provided a draft version of software fixes.

Paper outline. The rest of the paper is organized as follows: Section 2 introduces necessary background of this paper; Section 3 illustrates the root causes of ciphertext side channels in general; Section 4 shows how an attacker can break current cryptography implementations by monitoring ciphertext changes in the operating system's process control block; Section 5 shows that the secret leakage can also be caused by stack variables and heap buffers in user space; Section 6 discusses the potential countermeasures, including a kernel patch and application fixes; Section 7 discusses the threat of ciphertext side channels to other confidential VM implementations; Section 8 presents state-of-the-art related work and Section 9 concludes the paper.

2 Background

2.1 Secure Encrypted Virtualization

AMD Secure Encrypted Virtualization (SEV) is a trusted execution environment (TEE) supported by AMD server-level EPYC processors with "Zen" Architecture. SEV aims at providing confidential virtual machines for cloud customers. In SEV's threat model, other virtual machines, as well as the cloud host itself, are considered untrusted. The attacker may execute arbitrary code at the privileged hypervisor level and may also have physical access to the machine (*e.g.*, DRAM chips) [23]. To achieve this ambitious goal, a dedicated security subsystem consisting of the AMD Secure Processor (AMD-SP) and an AES memory encryption engine is introduced by SEV to protect data in use.

Hardware Memory Encryption. When SEV is enabled, the cryptographic isolation provided by Hardware Memory Encryption protects the confidentiality of the VM. Specifically, the VM's memory pages are always stored in encrypted form, and the VM encryption keys are guarded by the AMD-SP. SEV adopts a 128-bit AES encryption with the XOR-Encrypt-XOR (XEX) encryption mode, which incorporates a physical address-specific *tweak* such that the same plaintext yields different ciphertexts for each memory location. However, for a fixed address, an identical plaintext always yields the same ciphertext.

Nested Page Tables (NPT) and the page fault controlled channel. When SEV is enabled, the address translation between the VM's guest physical addresses and the host physical

addresses is managed by the hypervisor with the help of a NPT, which is a two-layer page table consisting of a Guest Page Table (GPT) and a Nested Page Table (NPT). The GPT is managed inside the guest VM and thus protected by the VM encryption key. The NPT is solely managed by the hypervisor.

As shown in prior work [25, 31, 36], the hypervisor can leverage the control over the NPT to intercept the execution of the guest with page granularity. To achieve this, the hypervisor can unset the Present bit (P bit) in the NPT. The next time the VM tries to access the corresponding guest physical page, a nested page fault (NPF) will be generated, revealing the addresses of the access and the causes.

SEV extensions. Two extensions of SEV have been introduced by AMD to add additional security protections since SEV's first release in 2016.

The second generation of SEV is called SEV-ES (Encrypted State) [22], which was first introduced in 2017. SEV-ES adds additional protection for CPU registers. Prior to SEV-ES, CPU registers were stored unencrypted in the Virtual Machine Control Block (VMCB) during world switches from the VM to the hypervisor (VMEXIT). In SEV-ES, the hardware automatically encrypts the registers in a designated Virtual Machine Save Area (VMSA) along with additional integrity protection. In addition, a guest-host communication protocol was introduced for instructions that need to expose registers to the hypervisor (*e.g.*, `CPUID`, `RDMSR`, *etc.*). A VMM Communication handler (#VC handler) inside the guest VM assists the instruction emulation. Specifically, the #VC handler intercepts those instructions with the help of hardware, passes necessary register values to a shared area called Guest-Host Communication Block (GHCB), triggers a special VMEXIT by the `VMGEXIT` instruction, and reads the resulting register values from the GHCB afterwards.

The third generation of SEV is called SEV-SNP (Secure Nested Paging) [3], which was released in 2020. As a response to attacks which used remapping or modification of guest memory in order to inject code into the VM [36], a structure called Reverse Map Table (RMP) was introduced. It maintains a second translation of host physical addresses to guest physical addresses as well as keeps track of the ownership of memory pages, and thus, prevents the hypervisor from modifying or remapping the guest VM's private memory. Most of the existing attacks against SEV and SEV-ES can be mitigated by SEV-SNP (Section 8).

2.2 Ciphertext Attacks against SEV-SNP

Ciphertext attacks against SEV-SNP were first introduced by Li *et al.* in CIPHERLEAKS [27]. The work exploited leakage caused by the ciphertext of the registers inside the VMSA.

Specifically, by inspecting the ciphertext stored in the VMSA during VMEXITs, an attacker could (1) infer the execution state of a known binary inside the guest VM, and (2) build a ciphertext-plaintext mapping for certain registers. For example, the ciphertext of the `RAX` register could reveal the return value of function calls. Since the ciphertext was deterministic, functions that returned the same value produced an identical ciphertext for the `RAX` register inside the VMSA, which is sufficient for the attacker to distinguish secret-related data content and steal secrets from an application using the OpenSSL library.

In response to that attack, AMD added additional randomization when encrypting and saving register values into the VMSA during VMEXITs [2]. Thus, the ciphertext of the register state is now completely different even if the register values inside CPU did not change between two VMEXITs, which fully mitigates the CIPHERLEAKS attacks.

2.3 Off-chip Attacks

Off-chip attacks are usually classified into *stolen DIMM attacks* and *bus snooping attacks*. Stolen DIMM attacks directly grab data from the Non-Volatile Memory (NVM) or perform cold boot attacks on volatile memory [33]. Bus snooping attacks target the data transmission between two components of the computer (*e.g.*, CPU and DRAM). These attacks involve both data eavesdropping and even data altering [12].

Off-chip attacks are also considered as one of the potential attacks in a TEE's threat model [3]. While the plaintext is protected inside the chip and can hardly be inspected, all data outside the CPU might be inspected, either on the external memory buses or on the NVM. TEEs like Intel SGX and AMD SEV protect data outside the CPU by an in-chip memory encryption engine. While it is widely accepted that attacks by monitoring the data bus flow can be thwarted by memory encryption [34], researchers move their attention to the unencrypted address bus [12]. Recent results [24, 32] showed that an attacker could recover some data by monitoring memory address patterns. For those attacks, an interposer is needed to be installed on the DIMM socket. The interposer can duplicate signals on the memory bus and pass the data to a signal analyzer on the fly with CPU cycle granularity.

2.4 Operating System Context Switch

Under x86_64, there are four different privilege levels that can be used to implement a hierarchy in the software [4, Sec. 4.9.1]. Under Linux, ring 0 is used to run the kernel, while ring 3 is used to run user space applications. When a privilege level change occurs, *e.g.* due to an interrupt or exception, the CPU automatically switches to a separate stack

and fills it with some information about the previous software. The stacks are configured in the Task State Segment (TSS). The register values, however, remain unchanged and are not stored by a hardware mechanism [4, Sec. 12.2.5]. Under Linux, one TSS per CPU is used, meaning that each CPU has its own set of stacks. Most Interrupt/Exception handlers use TSS managed stacks as an entry point to intialy store the register values, before eventually copying them to the so-called thread stack. The thread stack is part of the Process Control Block (PCB, also called `task_struct` in Linux), a data structure that bundles all information related to a process/thread. The saved registers are referred to as the `pt_regs` structure, which simply consists of the register values stored next to each other.

Note that in other scenarios a context switch is also used to describe a switch between different processes and threads. In this work, we always refer to the aforementioned privilege level change if not stated otherwise.

3 A generic ciphertext side channel

In this section, we are going to show that the ciphertext-based attack demonstrated in the CIPHERLEAKS paper is not limited to the VMSA register storage mechanism of SEV-SNP, but applies to any deterministically encrypted memory. We define a generic attacker model and show two primitives that allow the attacker to infer memory contents and runtime behavior of any application which relies on deterministically encrypted memory for protecting the confidentiality.

3.1 Attacker Model

We consider the standard threat model of confidential VM: The attacker has both software and physical access to the system, *i.e.*, they have unrestricted administrator capabilities and can physically access the machine. The confidential VM shields the VM's secrets from the attacker by encrypting the memory consumed by the user's application, using a deterministic memory encryption scheme with an address-based tweak, such that the ciphertext depends on the encryption key, the plaintext and the current physical address. Specifically, we target SEV-SNP, which also prevents the attacker from remapping memory containing ciphertext to other physical addresses, denies them write access to any encrypted memory, but leaves the attacker the ability to read ciphertext by software.

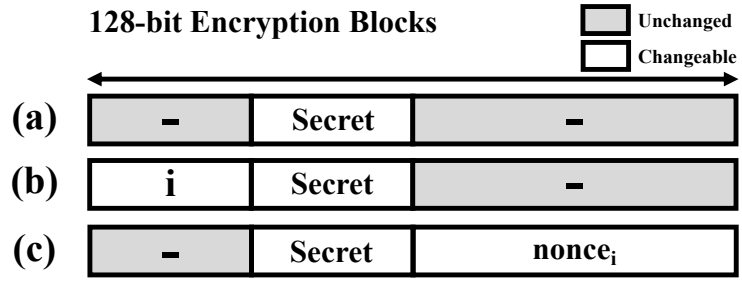


Figure 1: Encryption block configurations with different exploitability by the dictionary attack. In the first scenario (a), most of the block's plaintext is constant, with the secret being the only variable. Thus, the attacker can build a one-to-one mapping of ciphertexts to secrets. In (b), the block also contains a loop counter i, so there are many different ciphertexts mapping to the same secret. If the attacker can always observe the secret for a specific fixed value of i, they may still be able to build a dictionary, as this is equivalent to scenario (a). In the last scenario (c), the secret is followed by a random nonce which is regenerated before spilling secret to the memory. This prevents the attacker from creating a dictionary, as he never observes the same ciphertext twice.

3.2 Attack Primitives

We suggest two general methods for exploiting deterministic memory encryption: A *dictionary attack* and a *collision attack*.

Dictionary attack. A dictionary attack is applicable when a secret-dependent variable features a small, predictable value range with a fixed memory address. In this case, the attacker can build a dictionary of ciphertext-plaintext mappings for this variable and selectively recover the plaintext. This is a generalization of the approach taken in the CIPHERLEAKS attack, where the authors learned ciphertext mappings for the registers stored in the VMSA.

Contrary to CIPHERLEAKS, the dictionary attack targets arbitrary memory locations and variable types. Two examples about recovering ECDSA key using stack variables (Section 5.1), or registers stored during a context switch (Section 4) are presented. While this attack is quite powerful, it is restricted by the number of possible plaintexts for a given encryption block, since the attacker cannot tell which part of the plaintext has changed when observing a new ciphertext. If the targeted variable shares an encryption block with other variables which get new values frequently (*e.g.*, a loop counter), the number of possible plaintexts becomes too large to efficiently build a mapping, as is illustrated in Figure 1. We use this fact in Section 6.2 to propose a countermeasure which appends random nonces to small variables.

Collision attack. A collision attack transfers the concept of secret dependent code execution to memory writes. In secret-dependent branching, the attacker exploits that the

Algorithm 1 Constant time swap (CSWAP)

Require: Byte arrays a, b of same length and decision bit c
  $mask \leftarrow 0 - c$ ▷ $0 - 1$ underflows to 0xff
  for $i \leftarrow 0$ to $\mathrm{length}(a)$ **do**
    $x \leftarrow a[i] \oplus b[i]$
    $x \leftarrow x \ \& \ mask$
    $a[i] \leftarrow a[i] \oplus x$
    $b[i] \leftarrow b[i] \oplus x$
  end for

targeted algorithm executes a certain code region depending on specific values of a secret value (*e.g.*, an *if* statement checking key bits). By observing the access pattern to the respective code chunks, the attacker can learn the secret. A common countermeasure is so-called *constant-time* code, *i.e.* code that always exhibits the same control flow and memory accesses, independent of the secret. This is usually achieved by converting secret-dependent branch decisions into fixed expressions, which compute all possible results of a given operation and then use a mask to pick the desired one. One such primitive is the constant time swap CSWAP (Algorithm 1), which is used for example by the Montgomery ladder: CSWAP takes two variables a and b and a (secret) decision bit c. If the bit is set, the values of a and b are swapped; if the bit is cleared, a and b remain unchanged. The depicted code gadget always executes the same amount of instructions in the same order, and always accesses the same memory addresses, making it resistant against microarchitectural side-channel attacks.

But, if the attacker is able to observe whether the values of a or b *change*, they can immediately learn the decision bit c_i. The collision attack again exploits the fact that ciphertext blocks are deterministic. However, contrary to the dictionary attack, the attacker does not aim to learn the direct mapping of ciphertexts to actual plaintext values, but they only check whether certain ciphertexts *repeat* or *change*. Going even further, if the attacker knows that a memory write was executed (*e.g.*, through a control flow side-channel), but they do not see any ciphertext change, they learn that the instruction wrote the same value as was present in memory before. Given knowledge of the executed program, they may use this to infer more information other than the traditional control flow.

4 Leakage due to context switch

We now take the dictionary attack primitive from Section 3 and show how it can be used for extracting register values from a VM running with SEV-SNP. After CIPHERLEAKS,

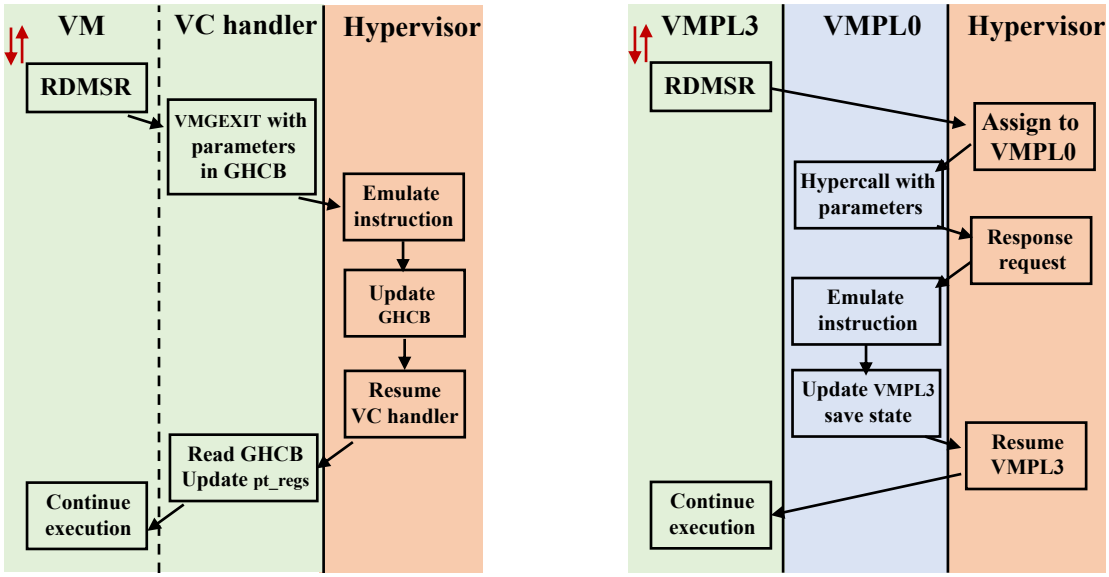
AMD published a firmware patch which added protection to the VMSA area [2]. However, the VM-hypervisor world switch is not the only occasion where the entire register state is written to memory. When moving from user space to kernel space (*e.g.*, after an interrupt or an exception), the Linux kernel pushes all register values of the user program onto the stack, and then copies those into the PCB of the current thread, such that the exception handler can access the register values through the `pt_regs` structure. The PCB address is fixed per-thread, allowing an attacker to build a dictionary of register values by causing repeated interrupts within the VM and observing the resulting ciphertexts. We show how an attacker can use nested page faults to indirectly trigger internal user-kernel context switches and use the learned register values to attack the constant-time ECDSA implementation of OpenSSL. Given their source code, similar attacks should also be applicable in WolfSSL, GnuTLS, OpenSSH, and libgcrypt.

4.1 Leaking Register Values via Context Switches

Forcing context switches in the VM. SEV-SNP restricts the hypervisor's ability to inject interrupts and exceptions into the VM, so we will show how a malicious hypervisor can work around this limitation by forcing the VM to pause at a certain execution point until a "natural" internal context switch is triggered, which should also be detectable by the hypervisor.

First, the hypervisor interrupts the targeted application at certain execution points by using the well-known page fault controlled channel, that allows the attacker to force a NPF when the VM tries to access or execute a given page. However, the NPF itself does not lead to a context switch inside the VM, as it is immediately intercepted by the hypervisor. To do so, the hypervisor now simply waits for a short amount of time and then resumes the VM without handling the NPF. As a result, the attacker can trap the execution of the targeted program and the victim application cannot resume its execution. After a short amount of waiting time, a time-driven internal context switch will be performed by the guest OS, which updates the victim application's register values in main memory (`pt_regs`).

Even though the internal context switch is out of the hypervisor's control, we show that the VM-host interaction mechanism adopted by SEV can work as an indicator of a finished context switch. Specifically, we observed that the guest VM has frequent interaction with the hypervisor through reading and writing hypervisor-managed registers of the Advanced Programmable Interrupt Controller (APIC), like `IA32_X2APIC_TMR1`, which are used for scheduling and timekeeping. These `RDMSR` and `WRMSR` accesses result in a special exception called #VC exception inside the VM, as they require the VM to share registers with the hypervisor. The #VC exception handler inside the VM then calls `VMGEXIT` after



(a) #VC handler (b) VMPL0 emulation

Figure 2: Workflow of how #VC exceptions are handled. Red arrows represent a context switch between processes.

putting the necessary register values into the GHCB (shown in Figure 2a). As the #VC exception is handled in VM's kernel space, a `VMGEXIT` also indicates a user-kernel context switch. Thus, the hypervisor simply waits for a `VMGEXIT` with the appropriate exit code, as an indicator of updated registers' ciphertext in `pt_regs`. We analyze the necessary pause time for triggering a `VMGEXIT` in Section 4.4.

Other than the traditional #VC handler mechanism, SEV-SNP has another option to adopt a more secure VM-host communication mechanism that moves the APIC emulation into the trust domain of the guest VM. As shown in Figure 2b, the VM is divided into multiple Virtual Machine Privilege Levels (VMPLs) that provide additional hardware isolated abstraction layers. However, the hypervisor can still sense a finished context switch due to the interaction triggered by the hypercall from VMPL0.

Locating `pt_regs` after VMEXIT. Besides using the `VMGEXIT` to detect a context switch, the attacker can also use it to locate the `pt_regs` struct. For that, after reaching a `VMGEXIT`, the attacker clears the P bit for all guest pages and resumes the VM. This will hand back control to the #VC handler in the VM, which will subsequently try to copy the results of the emulated instruction from the GHCB to `pt_regs`. Since all guest pages were marked as not present, this causes a nested page fault. In our experiments, the second NPF caused by data page read access after resuming the VM is the memory page containing `pt_regs`. We did not encounter any false positives during our experiments.

4.2 Attacking Constant-time ECDSA

In this section, we demonstrate how to use the context switch primitive from the previous section to attack the constant-time ECDSA implementation in OpenSSL. More precisely, we show that the adversary can infer the nonce k in the constant-time ECDSA algorithm by inspecting the ciphertext changes in the `pt_regs` structure of the targeted process. This can then be used to recover the secret key.

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a widely used signature algorithm that works as follows:

1. Prepare the curve parameters (CURVE, G, n), where G is the elliptic curve base point of prime order n.
2. Prepare a key pair by choosing uniform $d_A \in \mathbb{Z}_n^*$. d_A is the private key. The public key is $Q_A = d_A G$.
3. Generate a cryptographically secure random integer $k \in \mathbb{Z}_n^*$ (also known as the nonce k).
4. Calculate a non-zero r by $r = (kG)_x \mod n$ (only the x-coordinate of the resulting point is used).
5. Calculate $s = k^{-1}(h(m) + rd_A) \mod n$, where m is the message and $h(m)$ is a hash of m. (r, s) then forms the ECDSA signature pair.

A predictable or leaked nonce k allows to immediately recover the private key d_A by:

$$d_A = r^{-1}((ks) - h(m)) \mod n.$$

Targeted ECDSA implementation. Our attack targets the ECDSA implementation of the OpenSSL library[1] for the curve `secp384r1` that is commonly used for TLS/SSL connections. The goal of our attack is to steal the nonce k and thus infer the private key d_A. In OpenSSL, ECDSA signing is handled by the `ECDSA_do_sign` function, which in turn calls `ec_scalar_mul_ladder` to calculate `r`. Note that the implementation of the function is specifically designed to protect k against side channel attacks (Listing 6.1).

Identify instruction pages. Besides monitoring context switches and locating `pt_regs` via the methods shown in the previous part, we also need to identify the appropriate code locations in order to intercept the guest VM at proper execution points, which gives the attacker the opportunity to extract valuable ciphertext. In our work, we combine the widely-used page fault controlled side channel [26, 31, 35, 36] with performance counters to build a fine-grained tool to identify instruction pages' physical addresses.

[1] Commit: `c4b2c53fadb158bee34aef90d5a7d500aead1f70`.

Listing 6.1: Part of the elliptic curve scalar multiplication `ec_scalar_mul_ladder()` from OpenSSL. The function uses the Montgomery ladder algorithm and constant-time primitives to protect the secret scalar k against side channels.

```
int i, cardinality_bits, group_top, kbit, pbit, Z_is_one;
...
for (i = cardinality_bits - 1; i >= 0; i--) {
    kbit = BN_is_bit_set(k, i) ^ pbit;
// kbit is used to determine the conditional swap
    EC_POINT_CSWAP(kbit,r,s,group_top,Z_is_one);
// single step of the Montgomery ladder
    if (!ec_point_ladder_step(group, r, s, p, ctx)){
        ERR_raise(ERR_LIB_EC,
        EC_R_LADDER_STEP_FAILURE);
        goto err;
    }
// pbit helps to merge CSWAP with that of the next iteration
    pbit ^= kbit;
}
```

Specifically, we make use of the *Retired Instructions* counter [6, Event `PMCx0C0`], which can be configured to only count the amount of retired instructions inside the VM and thus reveal the number of instructions executed between two pages faults. The attacker can simply build a template of the retired instruction counts for code paths in a known binary. In our experiments, we were able to locate the target pages on the fly, without relying on repeated access patterns.

4.3 End-to-end attack against Nginx

We now show the steps needed to steal the nonce k generated by an Nginx webserver. The nonce, together with the corresponding signature, allows the attacker to recover the secret key of the server.

① **Send HTTPS request.** The attacker sends a HTTPS request to the Nginx server in order to trigger the targeted code paths.

② **Locate target function in physical memory.** Right after sending the HTTPS request, the attacker clears the P bit of all VM pages. The attacker then locates the guest physical addresses of the functions `ec_scalar_mul_ladder()` (gPA_0) and `BN_is_bit_set` (gPA_1) using the page fault channel combined with the retired instruction counter.

③ **Locate** `pt_regs`**.** The attacker pauses the VM for a while (*e.g.*, by trapping the VM in the NPF handler for a few milliseconds) when they intercept a NPF of gPA_0. They then

use the method from Section 4.1 to find the physical address gPA_3 of the current thread's `pt_regs` structure.

④ **Single-step loop iterations.** The attacker iteratively clears the P bit of gPA_1 to pause the VM when it enters `BN_is_bit_set`. After intercepting the corresponding NPF for gPA_1, the attacker clears the P bit for gPA_0, causing an NPF when the `ret` instruction inside `BN_is_bit_set` is executed, *i.e.* the function tries to return to `ec_scalar_mul_ladder()`. The attacker then pauses the VM in the gPA_0 NPF for a while (several milliseconds) and resumes the VM without handling the NPF. The attacker might observe several consecutive NPFs for gPA_0, but keeps the P bit cleared until a `VMGEXIT` is encountered.

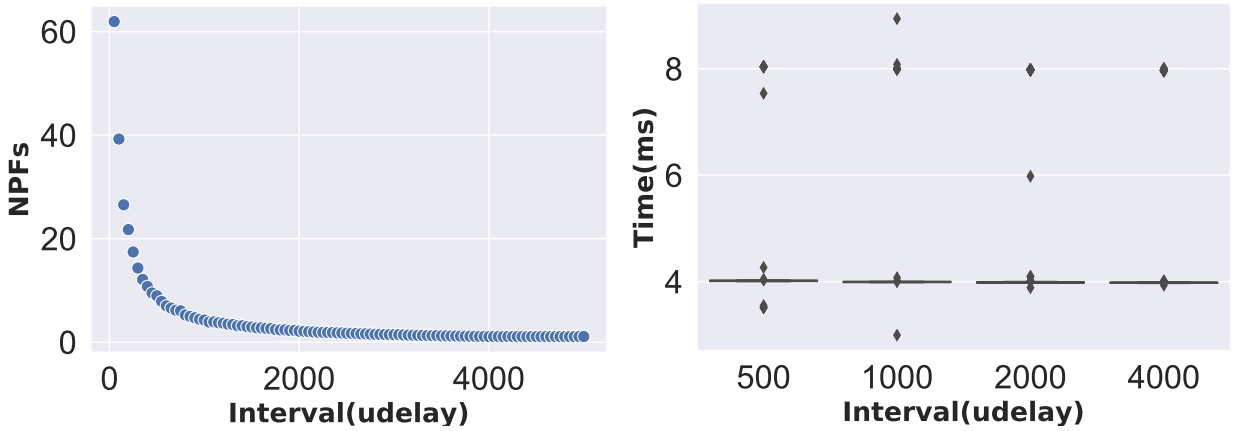
⑤ **Record the ciphertext and recover the nonce k.** The attacker records the ciphertext of the RAX field in `pt_regs` after the `VMGEXIT`, which contains the return value of `BN_is_bit_set` at this execution point. The conjunct register stored near RAX in `pt_regs` is R8, which remains unchanged during the `for` loop. The attacker then sets the P bit of gPA_0, clear the P bit of gPA_1 in order to intercept `BN_is_bit_set` for the next iteration and repeat step ④. After 384 iterations, the attacker has collected a sequence of ciphertexts. Since RAX can only take two distinct values, they can recover the nonce `k` with only 1 bit of entropy.

4.4 Evaluation

All experiments throughout this paper were conducted on an AMD EPYC 7763 64-Core Processor. The host kernel (branch `sev − snp − part2 − rfc4`), QEMU (branch `sev − snp − devel`), and OVMF (branch `sev − snp − rfc − 5`) were directly forked from AMD SEV's GitHub repository [5]. The victim VMs were protected by SEV-SNP and used the unmodified guest kernel provided by AMD (branch `sev − snp − part2 − rfc4`). The victim VMs were configured with 2GB DRAM, 30GB disk, and one virtual CPU (vCPU). However, the capacity of the victim VMs (including vCPU, DRAM, and disk) is not relevant for the attack procedure.

For the attack on Nginx, an unmodified Nginx server and an OpenSSL library were installed inside the victim VMs. The Nginx version is 1.21.3, which was released on 07 Sep. 2021. The Nginx server supports HTTPS requests with a self-signed ECC certificate with 384-bit key. The curve used is `secp384r1`. The OpenSSL was forked from OpenSSL's Github repository (Commit: `c4b2c53fadb158bee34aef90d5a7d500aead1f70`) and was modified to log the ground truth after the signing procedure, so we could verify the extracted secret.

Proof of concept code is available at `https://github.com/UzL-ITS/sev-ciphertext-side-channels/`.



(a) #NPFs for `udelay` intervals.

(b) Avg. time for context switch.

Figure 3: Relationship between `udelay` interval and internal context switch.

Identifying target functions. To estimate the attacker's ability to locate target functions on the fly, we sent 500 consecutive HTTPS requests. For each request, we monitored the page access pattern along with the number of retired instructions and tried to locate the target functions in real-time. The reference page access pattern and the corresponding performance counter values were collected in a different VM with the same Nginx and OpenSSL version, but without SEV-SNP's protection and with a different kernel version, to show the pattern's independence of the exact kernel version.

In 496 out of those 500 requests, the target function's physical addresses were successfully located, while a miss was reported for the remaining four requests. The average time needed to locate the target functions was 59.28 milliseconds with a standard deviation of 2.12 milliseconds. No false positive was reported.

Context-switch latency. To collect the ciphertext of the updated `pt_regs`, the attacker needs to wait until an internal context switch, which is the most time-consuming part of the end-to-end attack. In our implementation, the attacker pauses the VM by calling `udelay`(< `interval` >), which takes a delay in microseconds. We evaluated both the proper interval for a direct context switch and the average waiting time. Since the attacker doesn't set the P bit at the execution point unless observing the `VMGEXIT`, the attacker might get several repeated NPFs in a row. Figure 3a shows the number of NPFs we observed under different intervals. We usually directly detected a context switch when `interval` was larger than 2000 (two milliseconds). Figure 3b shows the average waiting time. It usually took four milliseconds until an internal context switch occurred, thus we paused the victim VM by using `udelay`(`4000`) in our attack.

Performance. We repeated the attack 50 times and measured the overall time for an end-to-end attack. The average time was 8.53 seconds with a standard deviation of 0.33

seconds. The main latency is caused by waiting for an interval context switch. For a 384-bit nonce k, the attacker can intercept 384 * 5 = 1920 NPFs for gPA_0 in total. In our setting, we chose to wait for a context switch every time when intercepting an NPF of gPA_0. However, for each iteration, only one out of five NPFs is caused by the `ret` instruction inside `BN_is_bit_set`. Thus, the attacker could also choose to only wait and grab ciphertext at that NPF. By doing that, approximately 6 seconds (384 * 4 * 4ms) waiting time can be avoided. However, one side effect is that some internal events (*e.g.*, an unexpected context switch) might cause a repeated NPF of gPA_0, which will confuse the attacker and reduce the accuracy. In our implementation, the average accuracy for the recovered nonce `k` is 89.1%.

5 Exploiting memory accesses in user space

In the previous section, we have seen how an attacker can exploit the context switch mechanism of the Linux OS inside the VM to leak register values of running processes. We now turn our attention to leakages directly caused by the victim application's memory access behavior. We demonstrate that the OpenSSL ECDSA code from the previous section is also vulnerable to the dictionary attack targeting stack variables, and show an example of the collision attack against the EdDSA implementation in OpenSSH.

5.1 Breaking Constant-time ECDSA via Dictionary Attack

As shown in Listing 6.1, `ec_scalar_mul_ladder` uses several local integer variables: `kbit` controls the conditional swaps by `EC_POINT_CSWAP` in the `for` loop. Assuming that k_i refers to the i-th bit of k, at the beginning of a loop iteration, `pbit` stores k_{i-1}. After calling `BN_is_bit_set(k, i)` to retrieve k_i, `kbit` stores $k_{i-1} \oplus k_{i-2}$ (XOR). `pbit` is later updated to k_i at the end of the iteration.

Stack layout. We target the 16-byte memory block where `pbit` is stored. By our observation, the memory block containing `pbit` also contains additional variables, which is not surprising given the small size of `pbit`. In our case, `pbit`, `kbit` and `cardinality_bits` all share the same 16-byte memory block. The `cardinality_bits` variable does not change during the runtime of the `for` loop from Listing 6.1. Thus, the value range of the ciphertext is only dependent on the secret, *i.e.* `pbit` and `kbit`.

Recovering `k` from ciphertext pairs. Recall that, at the end of each loop iteration, `pbit` stores the i-th bit of the nonce k. The attacker thus can recover k if they can infer the value of `pbit` in each iteration. We use gPA_0 to denote the guest physical address of the stack

Table 1: All possible `pbit` and `kbit` pairs when intercepting `BN_is_bit_set()` in `ec_scalar_mul_ladder()`. The letters `A` to `D` represent the 16-byte ciphertexts the attacker may observe, which depend on the values of `kbit` and `pbit`. The value of `kbit` and `pbit` in the $i+1$-th iteration is updated depending on k_i.

i-th iteration				$i+1$-th iteration		
`pbit`	`kbit`	**Pair**	k_i	`pbit`	`kbit`	**Pair**
0	0	A	0	0	0	A
0	0	A	1	1	1	D
0	1	B	0	0	0	A
0	1	B	1	1	1	D
1	0	C	0	0	1	B
1	0	C	1	1	0	C
1	1	D	0	0	1	B
1	1	D	1	1	0	C

page where `pbit` is stored, and gPA_1 for the address of `BN_is_bit_set()`. Similar to the attack in Section 4.1, the attacker uses the page fault controlled channel in combination with the retired instructions performance counter for locating the pages.

The attacker records the ciphertext of gPA_0 when he intercepts the NPF of `BN_is_bit_set()` (gPA_1), which corresponds to the state after the previous loop iteration (*i.e.*, `pbit` still has its old value). As shown in Table 1, in the i^{th} iteration, the attacker can observe one of four possible `pbit` and `kbit` pairs. We use the letters `A` to `D` to denote the four possible ciphertexts. At the end of the i-th iteration, `pbit` and `kbit` are updated according to k_i (0 or 1). Thus, when the attacker intercepts the NPF of gPA_1 in the $i+1$-th iteration, there are 8 possible observation cases.

They then analyze the ciphertext of gPA_0 to (1) locate the offset of the 16-byte block where `pbit` is in and to (2) infer the value of `pbit` for this iteration. For (1), the attacker can easily identify the offset because they should observe the four different ciphertext randomly but repeatedly at a certain offset, which reveals the ciphertext changes of the pair (`pbit`, `kbit`). For (2), the attacker can infer the value of `pbit` by analyzing two subsequent ciphertext of (`pbit`, `kbit`) as shown in Table 1. The attacker applies the following algorithm to recover the `pbit` sequence: In the first iteration, both `kbit` and `pbit` are initialized to 1, thus producing ciphertext `D`. The attacker then finds an n-th iteration that has the same ciphertext as the following $n+1$-th iteration. Then (`pbit`, `kbit`) for the n-th and $n+1$-th iterations must either be `A` or `C`. If the next $n+x$-th iteration with a different ciphertext produces a ciphertext other than `D`, then the ciphertext for n^{th} and $n+1^{th}$ iterations must be `C`. Otherwise, the ciphertext represents `A`. After identifying `A`, `C`, and `D`, the remaining ciphertext represents `B`.

5.1.1 Attack Steps

① **Locate the two target physical addresses.** The attacker first needs to locate the guest physical addresses of the target stack page gPA_0 and the target function page gPA_1 . We use the same methods as in Section 4.1 to locate the pages.

② **Intercept the `for` loop.** The attacker iteratively clears the P bit in the NPT to interrupt the execution of the `for` loop. Specifically, the attacker clears the P bit of gPA_0 when a NPF of gPA_1 is intercepted and clears the P bit of gPA_1 when a NPF of gPA_0 is intercepted later. The attacker thus tracks the internal execution states of the `for` loop.

③ **Record the ciphertext of gPA_0.** Given the structure of the loop, there are 5 NPFs for both gPA_0 and gPA_1 for one iteration. Thus, for a 256-bit nonce k, the attacker needs to intercept 256 * 5 = 1280 NPFs for both gPA_0 and gPA_1. In each iteration, the first NPF for gPA_0 is triggered when `BN_is_bit_set` finishes execution and the program tries to touch the stack page where (`pbit` and `kbit`) is in. At this execution point, both `kbit` and the `pbit` are not yet updated. The attacker records the ciphertext of the whole stack page since the offset of `pbit` and `kbit` change slightly between different runs of the algorithms.

④ **Infer the value of k.** After all 256 iterations of the `for` loop, the attacker determines the offset and recovers the nonce k using the strategy we introduced in Section 5.1.

5.1.2 Evaluation

The test platform was the same as described in Section 4.4. Instead of targeting the `secp384r1` curve, we picked a different curve `secp256k1`, which is widely used in Bitcoin, to show that the attack works for different curves. The victim VM computes an ECDSA signature by calling `ECDSA_do_sign` in the OpenSSL library. We repeated the attack 50 times. In 92% of the attempts, we could recover the nonce `k` with 100% accuracy. After identifying the target functions, which we only needed to do once, the average time used to conduct the attack is 1.23 seconds with a standard deviation of 1.01 seconds.

5.2 Breaking Constant-time EdDSA via collision attack

In the previous attack case studies we have used the dictionary attack primitive by guessing and recording plaintext-ciphertext mappings. We now show how the attacker can break constant-time EdDSA by monitoring the collision of the secret dependent value's ciphertext. While the attack would also be applicable to the constant time swaps

used by the ECDSA variant described above, we show how the collision attack can work on the constant time EdDSA implementation of OpenSSH with the `ed25519` curve. As this implementation processes the secret in a batched manner, it is less susceptible to the dictionary attack previously applied to the ECDSA implementations.

The EdDSA signature algorithm [9] works similar to ECDSA, with the most noticeable difference being the deterministic nonce generation to prevent attacks based on flawed random number generators. The algorithm works as follows:

1. Provide a valid EdDSA parameter set (CURVE, G, n, c, l, H) with $2^c \cdot l = |\text{CURVE}|$, where G is the elliptic curve base point of prime order l and thus $l \cdot G = 0$. H is a cryptographic hash function with $2b$ output bits.
2. Prepare a key pair. Choose a secure random b-bit string d_A as the secret key. Calculate the public key $Q_A = d_s G$, where d_s is derived from the hash of d_A.
3. Deterministically compute a nonce for the signature as $k = H(H_{b,\ldots,2b-1}(d_A) \parallel m)$, where m is the message.
4. Calculate $R = kG$.
5. Calculate $s = k + H(R \parallel Q_A \parallel m) \cdot d_s \mod l$. The final EdDSA signature is defined as the tuple (R, s).

Targeted EdDSA implementation. We target the EdDSA implementation of OpenSSH 8.2p1, which is the version shipped with the latest Ubuntu LTS 20.04. The targeted implementation uses the `ed25519` curve. More precisely, we attack the multiplication $R = kG$ to learn k which then allows us to recover d_s from s, by computing

$$d_s = (s - k) \cdot H(R \parallel Q_A \parallel m)^{-1} \mod l.$$

While d_s is not the actual private key d_A, it is sufficient to create valid signatures.

Listing 6.2 shows the function performing the calculation $k \cdot G$. The arithmetic is implemented using a windowing technique with pre-computed partial sums in a lookup table. First, in line 6, the secret scalar is broken down into 3-bit chunks. In addition, a transformation is applied converting the chunks to signed values. However, this is reversible. Lines 12 and 13 in the for loop contain the main multiplication work. In `choose_t` the partial sum is loaded from the precomputation table in a cache attack resistant manner by accessing multiple values and choosing the correct one using a constant time swap operation. Line 13 performs the actual multiplication.

For our attack, we focus on the constant time swap operation `cmov_aff` that is used in `choose_t`. Both functions are shown in Listing 6.3. The idea of the attack is to use the collision attack to leak the value of `b`, which corresponds to d_s in our EdDSA description,

Listing 6.2: Function performing the multiplication of the secret scalar with the curve base point. In the original code, the variable k is named s.style

```
void ge25519_scalarmult_base(ge25519_p3 *r, const sc25519 *k) {
   signed char b[85];
   int i;
   ge25519_aff t;
   sc25519_window3(b,k);
   choose_t((ge25519_aff *)r, 0, b[0]);
   fe25519_setone(&r->z);
   fe25519_mul(&r->t, &r->x, &r->y);
   for(i=1;i<85;i++) {
      choose_t(&t, (unsigned long long) i, b[i]);
      ge25519_mixadd2(r, &t);
   }
}
```

in the calls to `cmov_aff`. We compare the values of `t` before and after the function call. While the constant-time swap will write to the memory locations regardless of the value of `b`, to be secure against cache and timing side channels, the actual value that is written still depends on `b`. Although the written data has a large value range, making a dictionary attack infeasible, it suffices to compare the ciphertext of `t` before and after the call to `cmov_aff` without knowing the plaintext for the ciphertext. The information whether the ciphertext value has changed or not allows us to directly infer `b`.

After leaking the value of `b`, the attacker inverts the operations applied in `sc25519_window3` (Listing 6.2) to recover the secret scalar k. Knowing k and the corresponding signature (R, s) allows to recover d_s, which is sufficient to create arbitrary valid signatures. Knowing d_s is not equal to knowing the secret key d_A, as the latter is still required to compute the nonce k according to step 3. However, only a party knowing the *private* key d_A can detect this subtle difference.

5.2.1 Attack Steps

① **Trigger the OpenSSH server.** The attacker opens an SSH connection with the server, and explicitly requests the usage of the EdDSA key. EdDSA is enabled in the default configuration under Ubuntu.

② **Locate the target physical addresses.** The attacker uses the page fault controlled channel and the performance counter technique from Section Section 4.1) to infer the physical addresses of the `choose_t` and `fe25519_cmov` functions.

```
static void cmov_aff(ge25519_aff *r, const ge25519_aff *p, unsigned char b) {
    fe25519_cmov(&r->x, &p->x, b);
    fe25519_cmov(&r->y, &p->y, b);
}

static void choose_t(ge25519_aff *t, unsigned long long pos, signed char b) {
  fe25519 v;
  int i = 0;
  *t = ge25519_base_multiples_affine[5*pos+0];
  cmov_aff(t, &ge25519_base_multiples_affine[5*pos+1],equal(b,1) | equal(b,-1));
  cmov_aff(t, &ge25519_base_multiples_affine[5*pos+2],equal(b,2) | equal(b,-2));
  cmov_aff(t, &ge25519_base_multiples_affine[5*pos+3],equal(b,3) | equal(b,-3));
  cmov_aff(t, &ge25519_base_multiples_affine[5*pos+4],equal(b,-4));
  fe25519_neg(&v, &t->x);
  fe25519_cmov(&t->x, &v, negative(b));
}
```

Listing 6.3: Swap and lookup table access functions.

③ **Intercept execution before and after the constant time swap operation.** The attacker then uses the page fault controlled channel to intercept the execution of the VM by unsetting the P bit of the targeted pages in the NPT.

④ **Take snapshots of the buffer `t`.** The attacker obtains the physical address of the buffer `t` by tracking the write access pattern during the execution of the constant time swap operation using the NPF side channel. The attacker then steps the loop using the page fault controlled channel and takes snapshots of the buffer `t` in each iteration.

⑤ **Recover the secret scalar `t`.** Using the snapshots of the buffer `t` before and after each call to `fe25519_cmov` in `choose_t` (note that `cmov_aff` wraps this function), the attacker can immediately deduce the value of `b`. After knowing the value of `b`, the attacker inverts the windowing and sign transformation operations applied in `sc25519_window3`$(\texttt{b}, \texttt{s})$ to obtain the secret scalar k. The attacker uses the first parameter R of the signature that the server sends in step ① to validate the value of k, and extracts the signing secret d_s from the second parameter S of the signature using k.

5.2.2 Evaluation

We ran the end-to-end attack 500 times. In 86% of the attacks, we could fully recover the signing secret with 100% accuracy. Of the failed attack runs, only 7 where due to errors in detecting the correct code pages. The remaining errors are most likely misdetections of the memory location of the buffer `t`. The average runtime of the attack was 7.9 seconds with 2.2 seconds standard deviation.

6 Countermeasures

There are two categories of countermeasures against the attacks presented in this paper: First, the underlying issue may be addressed at the architectural level, which would likely be the most reliable approach. Otherwise, the identified problems can be also tackled at the software level, with a certain performance overhead. We discuss both hardware/architecture-based and software-based countermeasures, and point out methods for hardening existing software against the attacks presented in this paper.

6.1 Architectural Countermeasures

There are two possible hardware approaches for closing the ciphertext side channels. However, both approaches introduce high overhead.

First, one may change the encryption mode of SEV to use *probabilistic* encryption: a random nonce or incremental counter is included in the encryption and is updated on each memory write, effectively randomizing the resulting ciphertexts on each write. However, probabilistic memory encryption requires additional memory for storing the nonces. For example, Intel SGX combines AES-based probabilistic encryption with MACs to achieve confidentiality, integrity and replay protection. In SGX, data is encrypted in a tweaked counter mode, where the nonce depends on both the physical address of the encrypted memory block and a 56 bit counter value, to ensure replay protection [16]. The counter values are kept in the integrity tree, together with the MAC tags that ensure integrity protection. Only the head nodes of the tree are stored on-chip, while the remaining integrity tree remains in memory and needs to be checked on each memory access, resulting in a significant memory and latency overhead.

A second approach is preventing the attacker from reading the VM's physical memory: On a software/firmware layer, this could be achieved by using a similar RMP mechanics as in SEV-SNP (Section 2.1), which already prevents write accesses through an additional RMP check. However, this would introduce a certain overhead when applied to all read operations due to the more frequent read access and the extra RMP lookup. For example, for a single read access inside the VM, a series of RMP checks are needed, including four checks for the 4-level GPT and one check for the data page. For each GPT level, four additional RMP checks are needed for the 4-level NPT. In addition, on-chip access control may still be susceptible to the off-chip attacks described in Section 2.3.

6.2 Software-based Countermeasures

While hardware-based countermeasures would be preferable due to stronger security guarantees, their feasibility and practicality demand further validation. Thus, in the following sections, we describe general methods for mitigating the vulnerabilities on a software level. There is no single software-based method that is perfectly suited for all scenarios, as kernel structures, stack, and heap are all vulnerable. Thus, we present how applications can mitigate ciphertext side channels in three different ways, building on the assumption, that register values are immune to the ciphertext side channel. However, as shown in Section 4, this is not the case, as the kernel stores the registers' content in memory upon context switches. Thus, we also present how the ciphertext side channel caused by register states stored inside kernel structures can be mitigated with a kernel patch, to achieve the invariant of secure registers (Section 6.3), and measure the kernel patch performance (Section 6.4).

Secret-aware register allocation. If secret-related variables would fit into a register, but are kept in memory due to register pressure, changing the register allocation strategy may be worth pursuing. The secret-related variables can be protected by staying inside the register during their lifecycle and never being spilled to memory.

In order to do that, compiler-level modifications are needed. Even though developers can suggest the compiler to keep some variables into registers by applying a `register` hint (*e.g.*, `registerintvar;`), the variables are not guaranteed to be placed inside registers. Thus, a compiler can be modified to prioritize variables marked as 'secret' when allocating registers. An example of a similar scheme is GINSENG [38], which employs a custom register allocation strategy and a secure storage in a TEE to shield sensitive variables from a malicious operating system. In case a register containing a secret must be spilled to the stack anyway (*e.g.*, it is frequently used in function calls or large variables), it can be protected using a random mask as described in the later software-based probabilistic encryption part.

Limiting reuse of memory locations. Both the dictionary attack and the collision attack rely on repeated writes to a fixed physical memory address. Thus, limiting reuse of a fixed memory address leads to fresh ciphertext and can prevent the attacker from inferring secrets via the ciphertext.

To achieve this, the application developer has to identify and rewrite vulnerable code sections. For example, in our collision attack (Section 5.2), the conditional swap operation should not be written to be performed in-place, but should store the result in a newly allocated memory area. In this way, an attacker always observes a fresh ciphertext in a new location, independent from the value of the decision byte c_i.

Software-based probabilistic encryption. If the aforementioned methods are not applicable, one can mimic probabilistic encryption in software and add a random nonce to the secret data each time when the data is written to the memory.

This can be approached in two ways: First, one can modify the memory layout of the affected data structures to include random nonces in between, such that each memory block gets a sufficient amount of random bits. Second, the memory layout is left as-is, but a second buffer of the same size is allocated for storing masks, which are then XOR-ed onto the plaintext.

The first approach can be implemented by reserving the high 8 bytes of each 16-byte encryption block for a random nonce, while the low 8 bytes are used for payload. When storing a value in this block, the nonce is incremented to ensure that the ciphertext changes. In addition, the old plaintext must be overwritten with a random value before storing the new plaintext, to keep the attacker from detecting consecutive writes of the same value. In the second approach, the nonces and the data are stored in separate locations, and the nonces are XOR-ed onto the data as a mask. On each memory write, the corresponding location in the mask buffer is resolved, the mask value is updated and then XOR-ed to the new plaintext. Finally, the masked plaintext is written to the desired memory address. As the nonces are high entropy values and updated independently of the written data, they are not susceptible to the dictionary attack or collision attack. Due to its high locality, the first approach is better suited for small variables (*e.g.*, variables on the stack), while the second approach has better support for pointer arithmetic and should thus be used for buffers and complex data structures. Both countermeasures could be implemented as a compiler extension, that automatically applies them to variables marked as secret.

6.3 Software-based Countermeasures: Kernel Context Switch

While the generic software-based countermeasures are sufficient to protect applications in user mode, they make the critical assumption that registers are immune to ciphertext side channels. However, our attack in Section 4 shows that the attacker can inspect the ciphertext in the kernel's `pt_regs` structure to infer register values. To mitigate the ciphertext leakage on register-level, we developed a kernel patch that protects registers during context switches. We focus on the Linux kernel, but similar methods can also be applied to other operating systems.

Specifically, the kernel patch protects the `pt_regs` structure, which stores x86-64 user space registers as described in Section 2.4. We present two methods for securing this structure. One is to insert a random nonce alongside each register. The other is to randomize the stack location on each context switch.

Storing a nonce alongside registers. A random 64 bits nonce can be stored next to each register (64-bit) to add enough randomization. In this way, on a context switch, the kernel doesn't simply push all registers to the stack, but interleaves them with pushes of a random value, which is incremented on every context switch. This method gives us 64 bits of security, which makes it impossible for the attacker to infer the plaintext even for long running VMs. However, this strategy comes with a major caveat: It requires significant changes to existing highly-optimized code paths, as a lot of exception/signal handling functions rely on the exact offset of the registers in `pt_regs` and would thus may not be adapted by the upstream kernel committee.

Context switch stack randomization. As an alternative strategy, we adapt the memory address randomization idea to the kernel entry point stack. Instead of inserting nonces between the saved registers, we randomize the address of the stack where the exception/interrupt handlers store the register values of the interrupted user space application.

This method is much less intrusive than the nonce approach and easy to hide behind a feature flag, as we only need to keep track of stack pages and replace the stack pointer on each exit from kernel space to user space. However, it also comes with a high memory overhead, as we have to reserve a lot physical memory only for the kernel entry point stacks. Also, at some point we will run out of physical memory, giving us a hard limit on the reachable entropy.

For example, if we assume that we have 8 GB of physical memory which can be freely used for our stack countermeasure, with a stack size of 4 KB (one page) we get 2^{21} possible stack locations (21 bits of entropy). This is significantly less than the 64 bits obtained with the nonce approach, but still considerably reduces the attack bandwidth, as the attacker would have to wait until a stack page repeats. To assess the practicality and the resulting overhead, we implemented the stack randomization countermeasure in the Linux kernel.

6.4 Case Study: Randomizing pt_regs Location

For our case study, we focused on the common exception and interrupt path described by `idtentry_body` which is defined in `arch/x86/entry/entry_64.S`. The `idtentry_body` path is *e.g.* used for the high frequency page fault exception as well as for the local APIC timer interrupt. The latter is especially interesting, as it is the main driver in determining if a task has used up its time slice, leading to a reschedule to a different task. While interrupts and exceptions can also occur when the CPU is already in kernel mode, we

restrict our countermeasure to events that interrupt a user space application, as they contain the register values that we want to protect.

Since the thread stack is empty upon entering the kernel from user space, we can simply replace it with a newly allocated stack. For the entry stack, randomizing the stack upon entry to the kernel is more difficult, as all general purpose registers hold user data and thus cannot be used to perform the change. To circumvent this, we randomize the stack on the exit path before returning back to user space. Thus upon the next entry, we have a fresh entry stack.

Using the regular memory allocation mechanisms of the Linux kernel for the stack allocation proves difficult, as they were not build with guarantees regarding not returning a recently freed page upon a new allocation. In addition, they share a common memory pool with the rest of the system, which increases the collision probability under high memory load, if taking random pages from the pool. Instead we allocate a large chunk of memory at boot time and manage the stacks in a first-in-first-out queue, maximizing the time between reuses.

To evaluate the performance of our prototype implementation, we call the `cpuid` instruction 10 million times in a tight loop from a user space application. Under SEV, this is an emulated instruction that will directly trigger the modified code paths in `idtentry_body` without doing further expensive computations, allowing us to efficiently measure the performance impact of the modifications to the context switch. Using this strategy, we measured a total average overhead of 1063 nanoseconds per context switch with standard derivation 4.93. We also ran a modified benchmark, where the application also loops over a large memory buffer each iteration, to measure the additional cache pressure created by randomizing the kernel stack. We ran the experiment 1000000 times resulting in a total average overhead of 2232 nanoseconds with standard derivation 297.

7 Discussion

Secure encryption of large memory. Memory encryption is a basic building block used in TEEs to establish the confidentiality of data that leaves the CPU. Ideally, a probabilistic authenticated encryption scheme needs to be used, as was implemented for the first generation of Intel SGX [16]. However, managing and updating authentication tags and counter values consumes additional storage, costs latency and decreases the memory bandwidth for payload data. Thus, we do not believe that integrity trees can scale to protect large amounts of memory, as it is required for the confidential VM usage model.

To cope with these conflicting properties, many confidential VM designs use a mixture of cryptography and additional, architectural permission checks to achieve their security guarantees. Since random memory access latency is a critical performance property for the entire system, ECB would be the best candidate from a performance point of view. However, the independent encryption of all memory blocks with the same key leaks repetition patterns, as there is only one ciphertext for each plaintext. Thus, current confidential VM designs (AMD SEV [23]), but also designs to be commercially available in the near feature (Intel TDX [19] and ARM CCA [7, 8]) all adopt a tweaked block cipher, like AES XTS/XEX. Table 2 shows a more comprehensive overview. These modes offer a middle ground between performance and security, as the tweak mechanism offers a cheap way to ensure that the same plaintext encrypts to different ciphertexts when stored in two different addresses. However, for a given memory block, there is still only one ciphertext for each plaintext. As we have seen throughout this paper, this is the root cause of the ciphertext side channels.

To prevent attacks on the missing integrity protection, systems like SEV-SNP or Intel TDX and Intel SGX prevent untrusted parties from writing to protected memory[3, 13]. Intel TDX and SGX also prevent read accesses to the ciphertext[13, 19]. However, as discussed in Section 2.3, these checks do not prevent physical attacks like bus snooping.

Finally, the implementation of access right checks also comes with technical hurdles. On the one hand, they need to be fast, as they influence the memory access latency. On the other hand, static approaches that simply block access to a fixed range, like in Intel SGX, hinder efficient memory use and scaling. These hurdles remain open research questions to be answered in the future works.

Side-channel resistant cryptosystems. With decades of studies on micro-architectural side channels, including cache or TLB side channels, building side-channel resistant cryptographic implementations has become a common practice. Most practically used cryptographic libraries adopt some levels of side-channel defenses, to prevent exploitation from a remote attacker [1] or another user on shared machines [39, 40]. The known best practice for defeating side channels is data-oblivious constant-time implementation, which dictates the execution time of the cryptographic operations (or an arbitrary portion of it) is constant regardless of the secret values used in the computation and that branch decisions or memory accesses may not depend on secret values. Data oblivious Constant-time implementation has been shown to defeat all known micro-architectural side-channel attacks, except the ciphertext side-channel attacks discussed in this work.

The ciphertext side channel opens up a new way of exploiting cryptographic code, which the data oblivious constant-time implementation is no longer sufficient to guard against. Given the difficulties of securing accesses to the ciphertext through memory access or

bus snooping (Section 2.3), we envision cryptographic code to be used in TEEs with large memory needs to adopt a new paradigm that achieves indistinguishability not only on execution time and access patterns, but on the ciphertext values. We hope our work will inspire a new research direction on secure implementation of cryptography, such as tools to automate the discovery of such vulnerabilities, compilers to transform a vulnerable code to a secure one, or formal provers to assert the absence of such vulnerabilities.

Table 2: Comparison of hardware memory encryption-based TEEs. Drop-In replacement means that applications do not need to be adjusted to work with the TEE. * denotes the release time of the whitepapers while the commercial machine is not available yet. † to our understanding only a recommendation for a possible instantiation.

Project	Vendor	Release	TCB type	TCB size	Drop-In replacement	Encryption mode	Block size
SEV [23]	AMD	2016	VM	No Limit	✓	XE or XEX	128-bit
SEV-ES [22]	AMD	2017	VM	No Limit	✓	XE or XEX	128-bit
SEV-SNP [3]	AMD	2020	VM	No Limit	✓	XEX	128-bit
SGX [13]	Intel	2015	Enclave	256 MB [18]	✗	AES-CTR + integrity + freshness	128-bit
SGX on Ice Lake SP[20, 21]	Intel	2021	Enclave	up to 1 TB	✗	XTS	128-bit
TDX [19]	Intel	*2020	VM	No limit	✓	XTS	128-bit
CCA[7]	ARM	*2021	VM	No limit	✓	AES XTS or QARMA†	128-bit†

8 Related work

To protect SEV-protected VMs against an untrusted cloud service provider, SEV adopts some additional designs atop traditional Virtualization. Some of those adjustments are challenged, including *AES memory encryption*, the *I/O bounce buffer* and *ASID-based key management*. Meanwhile, some designs inherited from AMD's traditional hardware-based virtualization are also proven to be insecure under the assumption of the untrusted host, including the *VM control block*, *Nested Page Tables*, and *ASID-tagged TLB entries*. Besides the Ciphertext leakage caused by VMSA, this section summarizes other attacks against SEV.

Intercept plaintext in VMCB (SEV). The original SEV allows the adversary to intercept and manipulate register values inside the unencrypted VMCB. Several existing works exploit the unencrypted VMCB vulnerability. Hetzelt *et al.* showed that the attacker could control the VM's execution and perform ROP attacks [17]. Werner *et al.* showed that the attacker can infer VM's instructions, fingerprint applications, and steal secret data [35]. From SEV-ES, registers are encrypted and stored in VMSA. For SEV-ES, an additional integrity check is performed on every VMRUN. For SEV-SNP, the RMP table restricts software's write access towards the VMSA area.

Manipulate Nested Page Table (SEV-ES). By changing the mapping between the guest physical address and the system physical address in the nested page table, the attacker can disturb the VM's execution and turn the VM's benign activities into malicious activities. In the SEVered attack [31], Morbitzer *et al.* showed that programs with a network interface (*e.g.*, web server) could be used to decrypt the VM's memory. Specifically, the attacker sends some file query requests to the webserver inside a SEV-enabled VM and then remaps the guest physical address belonging to those data files to some host physical addresses of private data. The private data will then be sent back to the attacker. The latest SEV-SNP mitigates this vulnerability by prohibiting the hypervisor from unauthorized NPT remapping.

Note that the hypervisor-controlled nested page table also results in a page-level controlled channel. The page fault controlled channel is widely used in numerous attacks against AMD SEV ([25, 27, 35, 36], *etc.*), and is used to infer the VM's activities and step its execution. SEV-SNP also suffers from this controlled channel. According to SEV-SNP's whitepaper [3], the page-level controlled channel is not in the scope of SEV-SNP's designed features.

Modify encrypted memory (SEV-ES). Before SEV-SNP, the hypervisor had write access to the VM's memory, which led to some delicate attacks ([10, 14, 36], *etc.*) that broke the integrity of SEV-enabled VMs by carefully overwriting their encrypted memory. Wilke

et al. [36] improved the analysis of the encryption modes on Zen 1 Embedded CPUs, discovering the updated XEX encryption mode and extending the reverse engineering of the tweak function. Using the tweak values in combination with a known plaintext-ciphertext dictionary, they built malicious code gadgets by copying ciphertext blocks in memory. Based on that, they bootstraped an encryption oracle. From Zen 2 onwards these attacks are no longer possible due to an improved tweak function.

Tamper with the I/O bounce buffer (SEV-ES). Because of the encrypted memory, DMA is not directly supported in SEV. A shared bounce buffer (SWIOTLB) is then introduced for I/O traffic. For incoming I/O traffic, the guest VM copies the data from the bounce buffer to its private memory. For outgoing I/O traffic, the guest VM copies the data from the private memory to the bounce buffer. The memory copy activities give the attacker a chance to construct encryption and decryption oracles. Li *et al.* [26] showed that the attacker could overwrite I/O traffic to encrypt/decrypt the VM's memory stealthily. SEV-SNP or processors with XEX mode memory encryption can mitigate this attack.

ASID-based momentary execution (SEV-ES). In SEV, including SEV-ES and SEV-SNP, the Address Space Identify (ASID) is managed by the untrusted hypervisor. While ASIDs play some rather important roles in SEV-enabled VMs, including cache tagging, TLB tagging, and identifying the VM encryption keys, the hypervisor has the ability to modify a VM's ASID during the VM's lifecycle. SEV relies on a "Security-by-Crash" principle that an improper ASID always causes a meaningless VM crash, assuming good behavior of the hypervisor. Li *et al.* [25] exploited this improper principle and introduced the CROSSLINE attacks. The authors showed that the attacker could extract the victim VM's encrypted memory blocks by setting an adversary-controlled attacker VM and changing the attacker VM's ASID to the victim VM's ASID. Because of the lack of ASID checks, the hardware always tried to execute the VM directly, which enabled momentary execution and a time window for leaking secrets. Even though SEV-SNP still gives the hypervisor the permission of ASID management, the additional ownership check mitigates the CROSSLINE attacks by restricting read access from the attacker VM to the victim VM.

ASID-tagged TLB (SEV-ES). Li *et al.* studied the hypervisor controlled TLB flush problem in SEV and SEV-ES [28] and presented TLB poisoning attacks. A TLB control field inside the VMCB controls the TLB flush during VMRUN. The authors exploited the fact that the hypervisor can skip TLB flushes by intentionally clearing the TLB control field. By doing so, the attacker could breach the TLB isolation between vCPUs from the same VM. The authors showed that an SSH connection controlled by the attacker could reuse other SSH connections' TLB entries and bypassed the login authentication. SEV-SNP adds a hardware-controlled TLB flush mechanism to mitigate this vulnerability.

Permutation agnostic attestation (SEV-ES). Wilke *et al.* [37] exploited that the attestation mechanism of SEV and SEV-ES was not able to detect permutations of the attested data in memory on a 16-byte granularity. They further showed how an attacker can use the ability to reorder code blocks to construct malicious code gadgets allowing to encrypt/decrypt arbitrary data. This attack is mitigated with SEV-SNP.

Voltage glitching attack (SEV-SNP). Buhren *et al.* studied a fault injection attack against AMD-SP, named voltage glitching attack [11]. Different from other works in this section, voltage glitching attack needs additional equipment (including a μController and a flash programmer) and real-physical access to SEV's machine. By inducing errors in AMD-SP's bootloader and implanting a malicious SEV firmware, voltage glitching attack are shown to be able to extract secrets used in SEV's remote attestation.

9 Conclusion

In this paper, we have performed a comprehensive study on the ciphertext side channels. Our work extends ciphertext side-channel attack to exploit the ciphertext leakage from *all* memory pages, including those for kernel data structures, stacks and heaps. We have also proposed a set of software countermeasures, including patches to the OS kernel and cryptographic libraries, as a workaround to the identified ciphertext leakage.

As a general design lesson, deterministic encryption modes like XEX must be combined with both read and write protection to prevent software-based attacks. To also prevent physical memory attacks, freshness and integrity protection are required.

References

[1] Nadhem J. AlFardan and Kenneth G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. 2013. DOI: 10.1109/SP.2013.42. URL: https://doi.org/10.1109/SP.2013.42.

[2] AMD. *AMD Secure Encryption Virtualization (SEV) Information Disclosure (Bulletin ID: AMD-SB-1013)*. https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1013. 2021.

[3] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. 2020-01.

[4] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Manual. AMD, 2019.

[5] AMD. *AMDSEV/SEV-ES Branch*. https://github.com/AMDESE/AMDSEV/tree/sev-es. 2020.

[6] AMD. "Open-Source Register Reference For AMD Family 17h Processors Models 00h-2Fh". In: *Manual* (2018-07). Rev 3.03.

[7] ARM. *Arm CCA Security Model*. Rev 1.0, Document Number DEN0096. 2021-08.

[8] ARM. *Introducing Arm Confidential Compute Architecture*. https://developer.arm.com/documentation/den0125/latest. Revision 0300-01. 2023-06.

[9] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-Speed High-Security Signatures". In: *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*. 2011. DOI: 10.1007/978-3-642-23951-9_9. URL: https://doi.org/10.1007/978-3-642-23951-9%5C_9.

[10] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. "Fault Attacks on Encrypted General Purpose Compute Platforms". In: *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ , USA, March 22-24, 2017*. 2017. DOI: 10.1145/3029806.3029836. URL: https://doi.org/10.1145/3029806.3029836.

[11] Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean -Pierre Seifert. "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3484779. URL: https://doi.org/10.1145/3460120.3484779.

[12] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. 2017. DOI: 10.1145/3152701.3152706. URL: https://doi.org/10.1145/3152701.3152706.

[13] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: *IACR Cryptol. ePrint Arch.* (2016), p. 86. URL: http://eprint.iacr.org/2016/086.

[14] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. "Secure Encrypted Virtualization is Unsecure". In: *CoRR* abs/1712.05090 (2017). arXiv: 1712.05090. URL: http://arxiv.org/abs/1712.05090.

[15] Google. *Introducing Google Cloud Confidential Computing with Confidential VMs*. https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms. 2020.

[16] Shay Gueron. "A Memory Encryption Engine Suitable for General Purpose Processors". In: *IACR Cryptol. ePrint Arch.* (2016), p. 204. URL: http://eprint.iacr.org/2016/204.

[17] Felicitas Hetzelt and Robert Buhren. "Security Analysis of Encrypted Virtual Machines". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*. 2017. DOI: 10.1145/3050748.3050763. URL: https://doi.org/10.1145/3050748.3050763.

[18] Intel. *10th Generation Intel Core Processor Families*. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/10th-gen-core-families-datasheet-vol-1-datasheet.pdf. Volume 1, Document Number: 341077-005. 2020-07.

[19] Intel. *Intel Trust Domain Extensions*. https://cdrdv2.intel.com/v1/dl/getContent/690419. Accessed on 19.09.2024. 2023-02.

[20] Intel. *Product Brief, 3rd Gen Intel Xeon Scaleable Processor for IoT*. https://www.intel.com/content/www/us/en/products/docs/processors/embedded/3rd-gen-xeon-scalable-iot-product-brief.html. 2021.

[21] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. *Supporting Intel SGX on Multi-Socket Platforms*. White Paper. Intel, 2021.

[22] David Kaplan. *Protecting VM Register state with SEV-ES*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf. 2017-02.

[23] David Kaplan, Jeremy Powell, and Wolle. *AMD Memory Encryption*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf. 2021-10.

[24] 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: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lee-dayeol.

[25] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. "CrossLine: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3485253. URL: https://doi.org/10.1145/3460120.3485253.

[26] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. "Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan.

[27] 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: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan.

[28] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. "TLB Poisoning Attacks on AMD Secure Encrypted Virtualization". In: *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. 2021. DOI: 10.1145/3485832.3485876. URL: https://doi.org/10.1145/3485832.3485876.

[29] Microsoft. *Azure and AMD announce landmark in confidential computing evolution*. https://azure.microsoft.com/en-us/blog/azure-and-amd-enable-lift-and-shift-\confidential-computing/. 2021.

[30] Mathias Morbitzer, Manuel Huber, and Julian Horsch. "Extracting Secrets from Encrypted Virtual Machines". In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX , USA, March 25-27, 2019*. 2019. DOI: 10.1145/3292006.3300022. URL: https://doi.org/10.1145/3292006.3300022.

[31] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. "SEVered: Subverting AMD's Virtual Machine Encryption". In: *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*. 2018. DOI: 10.1145/3193111.3193112. URL: https://doi.org/10.1145/3193111.3193112.

[32] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[33] Patrick Simmons. "Security through amnesia: a software-based solution to the cold boot attack on disk encryption". In: *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*. 2011. DOI: 10.1145/2076732.2076743. URL: https://doi.org/10.1145/2076732.2076743.

[34] Shivam Swami and Kartik Mohanram. "COVERT: Counter OVErflow ReducTion for efficient encryption of non-volatlle memories". In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. 2017. DOI: 10.23919/DATE.2017.7927117. URL: https://doi.org/10.23919/DATE.2017.7927117.

[35] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. "The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*. 2019. DOI: 10.1145/3321705.3329820. URL: https://doi.org/10.1145/3321705.3329820.

[36] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. "SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00080. URL: https://doi.org/10.1109/SP40000.2020.00080.

[37] Luca Wilke, Jan Wichelmann, Florian Sieck, and Thomas Eisenbarth. "undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation". In: *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*. 2021. DOI: 10.1109/SPW53761.2021.00064. URL: https://doi.org/10.1109/SPW53761.2021.00064.

[38] Min Hong Yun and Lin Zhong. "Ginseng: Keeping Secrets in Registers When You Distrust the Operating System". In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. 2019. URL: https://www.ndss-symposium.org/ndss-paper/ginseng-keeping-secrets-in-registers-when-you-distrust-the-operating-system/.

[39] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-Tenant Side-Channel Attacks in PaaS Clouds". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. 2014. DOI: 10.1145/2660267.2660356. URL: https://doi.org/10.1145/2660267.2660356.

[40] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-VM side channels and their use to extract private keys". In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 2012. DOI: 10.1145/2382196.2382230. URL: https://doi.org/10.1145/2382196.2382230.

7

SEV-Step: A Single-Stepping Framework for AMD-SEV

Publication

Luca Wilke, Jan Wichelmann, Anja Rabich and Thomas Eisenbarth. Published at the *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2024.

Contribution

I am the main author. Jan implemented the algorithm for the AES key recovery in the LUKS2 attack and Anja conducted the interrupt latency measurements.

Outline

SEV-Step: A Single-Stepping Framework for AMD-SEV

Luca Wilke, Jan Wichelmann, Anja Rabich and Thomas Eisenbarth.

University of Lübeck

The ever increasing popularity and availability of Trusted Execution Environments (TEEs) had a stark influence on microarchitectural attack research in academia, as their strong attacker model both boosts existing attack vectors and introduces several new ones. While many works have focused on Intel SGX, other TEEs like AMD SEV have recently also started to receive more attention. A common technique when attacking SGX enclaves is single-stepping, where the system's APIC timer is used to interrupt the enclave after every instruction. Single-stepping increases the temporal resolution of subsequent microarchitectural attacks to a maximum. A key driver in the proliferation of this complex attack technique was the SGX-Step framework, which offered a stable reference implementation for single-stepping and a relatively easy setup. In this paper, we demonstrate that SEV VMs can also be reliably single-stepped. To lay the foundation for further microarchitectural attack research against SEV, we introduce the reusable SEV-Step framework. Besides reliable single-stepping, SEV-Step provides easy access to common attack primitives like page fault tracking and cache attacks against SEV. All features can be used interactively from user space. We demonstrate SEV-Step's capabilities by carrying out an end-to-end cache attack against SEV that leaks the volume key of a LUKS2-encrypted disk. Finally, we show for the first time that SEV is vulnerable to Nemesis-style attacks, which allow to extract information about the type and operands of single-stepped instructions from SEV-protected VMs.

1 Introduction

Microarchitectural side-channel security of computer systems has been one major pillar of computer security research in recent years. In microarchitectural attacks, the adversary aims to infer/extract secret information through observations of the system's microarchitectural state. With the ever increasing popularity and availability of Trusted Execution Environments (TEEs), side-channel attacks are more relevant than ever, as the attacker model of TEEs includes powerful system-level attackers. Naturally, such an attacker has more capabilities to observe the system's microarchitectural state, extending the potential attack surface. While attacks targeting TEEs build on a variety of data sources, like cache state, microarchitectural buffers or power reporting interfaces, they share the

property that they have to synchronize their data sampling with the execution flow of the victim. For example, monitoring the cache state only leaks meaningful information if the victim is about to perform a vulnerable memory access. An increased temporal or spatial resolution of the attacker's ability to infer the victim's execution state often drastically improves the amount/quality of leaked data.

One commonly used technique with both Intel SGX and AMD SEV is disabling certain memory pages, such that the victim is forced to handle a page fault when it tries to access those pages [35, 43, 45, 48, 54, 60]. This allows the attacker to synchronize with the victim's execution flow on a page-granular level. For Intel SGX, researchers tried to further increase the resolution by interrupting the SGX enclave with a high frequency, e.g., by using the system's APIC timer. Eventually, Van Bulck et al. demonstrated in SGX-Step [16] that an attacker can even achieve the maximum temporal resolution of interrupting the victim after every single instruction. However, besides this technical improvement over prior work that was only able to interrupt SGX enclaves every few instructions, they were also the first to introduce a reusable framework. Now, at the time of writing this paper, SGX-Step has been used in 33 publications [14], clearly showing the benefits of reusable building blocks in a research area where the technical challenges and nuances are ever increasing.

While, for example, page fault tracking is also commonly used in SEV, most prior work has either not released any artifacts at all [20, 34, 35, 36, 46, 58] or artifacts that are highly specific to the demonstrated attack [60, 61]. Exceptions to this are [45] and [33]. The framework from [45] allows to track pages accessed by the SEV VM as well as remapping pages, but only applies to the first two versions of SEV. In addition, it does not allow to interactively react to page faults in a synchronous manner, making it unsuitable for many types of side-channel attacks. While the framework from [33] offers such interactivity, it also is restricted to page fault granularity.

Our Contribution

is twofold: First, we introduce *reliable single-stepping in the context of SEV(-SNP)*. The second contribution is making interactive single-stepping, page fault tracking and eviction set-based cache attacks available in a *single, reusable framework*. Our framework shifts most of the complex attack logic from kernel space into user space, allowing the development of new attacks entirely with user space code. In the hope that the framework inspires a similar community as SGX-Step, we dubbed it SEV-Step. Concurrent to our work, PwrLeak [57] also uses single-stepping, but only with plain SEV VMs, which are insecure anyways due to the unencrypted register state [22, 58]. They do not provide a rich framework and do not analyze the reliability.

Furthermore, to showcase the capabilities of our framework as well as its academic relevance, we demonstrate an end-to-end key extraction attack against a SEV VM and utilize SEV-Step to detect and quantify instructions based on their execution time. The end-to-end cache attack succeeds in extracting a LUKS2 disk encryption key from a SEV-protected VM using a single trace. The SEV-Step-based instruction latency analysis confirms that an attacker can leak information about the type and operands of certain instructions in SEV by measuring the time required for single-stepping them. Such classification was previously shown for SGX by Van Bulck et al. in Nemesis [15].

In summary, this work

- introduces reliable single-stepping against SEV VMs;
- provides a reusable framework facilitating future attack research against SEV;
- steals disk encryption keys in an end-to-end cache attack; and
- shows SEV's vulnerability to Nemesis-style [15] attacks.

The remainder of the paper is organized as follows: Section 2 provides background about relevant x86 system architecture and SEV. Section 4 starts with a general overview over the SEV-Step framework, before explaining its implementation in detail. Next, Section 5 evaluates the single-stepping and cache attack features of the framework. Finally, Section 6 demonstrates an end-to-end cache attack stealing disk encryption keys and shows that SEV is vulnerable to Nemesis-style [15] attacks.

2 Background

This section is structured as follows: First, we provide general background information on relevant x86 (micro)architecture and AMD's virtualization technology. Next, we introduce AMD SEV. Finally, we discuss related work.

2.1 AMD SVM

AMD Secure Virtual Machines (SVM) is AMD's instruction set extension for hardware-accelerated virtualization. It introduces the concepts of *guest mode* and *host mode*. Both modes have the full set of privilege levels of the x86 architecture. However, in guest mode certain instructions have slightly different semantics in order to enable the virtualization concept. As shown in Figure 1, from host mode, we can enter the guest mode using the `VMRUN` instruction. The CPU runs in guest mode until an intercepted event occurs, which leads to a `VMEXIT`, returning the execution flow to the instruction immediately following

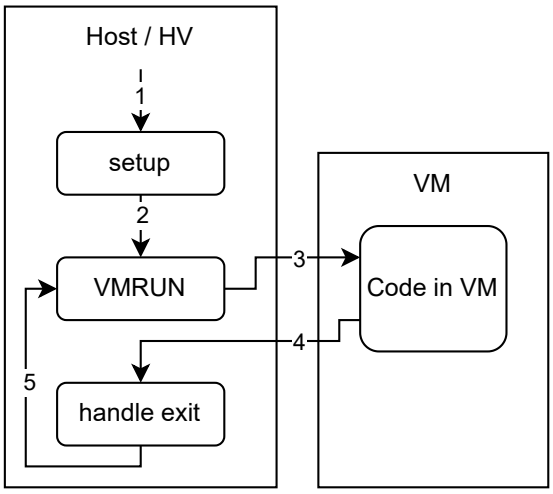


Figure 1: Basic control flow of a hypervisor using hardware assisted virtualization on AMD. After some initial setup (1), the hypervisor enters the main control loop. The `VMRUN` instruction takes care of performing the context switch into the VM (3). Afterwards, the VM is running until a `VMEXIT` event occurs (4), upon which the hardware restores the host context and resumes the execution immediately after the `VMRUN` instruction where the exit is handled, (5) before the VM is entered again.

the `VMRUN` instruction used to enter the VM. For both, VMRUN and VMEXIT, the hardware takes care of storing/restoring the current context, like the register values.

The host mode can pass a configuration struct called VMCB to the VMRUN instruction to configure, among other things, which events lead to a VMEXIT. This interception mechanism enables the host/hypervisor (HV) to transparently simulate certain behavior to the guest. Furthermore, the mechanism enables the HV to ensure that it stays in control of the hardware by causing periodic VMEXITs through APIC timer interrupts. Interrupt handling is discussed in detail in the next section.

SVM also introduces the concept of nested page tables (NPT) easing the virtualization of memory. With NPT, VM can no longer address real physical memory with its page tables. Instead the memory subsystem uses a second set of page tables, the hypervisor-controlled NPT, to translate the so-called Guest Physical Addresses (GPA) of the VM's page table to real physical addresses.

2.2 APIC and APIC Timer

According to the AMD Programmer's Manual [3, Sec. 16], the Advanced Programmable Interrupt Controller (APIC) is located between the CPU core and the rest of the system. It is responsible for providing the CPU core(s) with interrupts. Those interrupts can either

originate from sources local to the APIC, like the APIC timer interrupt, or from sources remote to the APIC, e.g., from the IOAPIC.

The APIC timer is part of the APIC Controller. It is a counter that is decremented with a configurable frequency. Once it reaches zero, it generates an interrupt. It can either be used in oneshot mode or in periodic mode. The latter restarts the timer once it reaches zero, while the former does not. The APIC timer is commonly used by the OS to implement periodic jobs like process scheduling [3, Sec. 16.4.1].

2.3 Interrupt Handling in AMD SVM

Under the x86 architecture, the delivery of interrupts is controlled via the `EFLAGS.IF` field. If set to 0, interrupt delivery is suppressed. This is called *masking* an interrupt. Masked interrupts are held waiting/pending until `EFLAGS.IF` is set to 1 again[3, Sec. 8.1.4].

In contrast to exceptions or traps, interrupts are inherently asynchronous to the currently executing program. However, instead of immediately aborting program execution, they are only processed on *instruction boundaries*, meaning that the currently executing instruction will still be retired before the interrupt is handled [3, Sec 8.2.24].

When using AMD's SVM to run a virtual machine, we distinguish between physical interrupts and virtual interrupts. Physical interrupts are interrupts that are actually generated by the hardware. As discussed in the previous section, the HV can configure the VMCB such that certain interrupts lead to a VMEXIT, returning control from the guest mode to the HV. However, to facilitate virtualization, the HV may decide to "pass on" the interrupt to the VM as a *virtual interrupt*. This mechanism is called *interrupt injection* and is performed via configuration fields in the VMCB.

To ensure that the VM cannot simply mask all physical interrupts using its version of the EFLAGS.IF register, the HV can configure the VMCB such that the VM's EFLAGS.IF flag only affects virtual interrupts. This way, the VM cannot prevent actual physical interrupts from being delivered [3, Sec. 15.21].

2.4 AMD SEV

AMD Secure Encrypted Virtualization (SEV) [29] is a Trusted Execution Environment (TEE) protecting whole virtual machines from a malicious HV and to some extent against physical attackers. It builds on the AMD SVM hardware acceleration for virtualization. These kinds of TEEs are also known as confidential VMs. With SEV, each VM's memory content is encrypted with AES-128 using the XOR-Encrypt-XOR (XEX) [53] mode before

leaving the main processor. A dedicated co-processor, the AMD Platform Security Processor (PSP), forms the root of trust of the system. It takes care of securely handling the memory encryption keys and offers an API to the HV to setup and manage SEV VMs. While located inside the main processor, for example in the cache, each VM's data is assigned a different tag, called Address Space Identifier (ASID) to ensure isolation. After the initial release of SEV, there were two iterative enhancements called SEV-ES [28] and SEV-SNP [2], the latter being the latest version.

2.5 Attacks on AMD SEV

Since its release, there has been a long line of attacks against AMD SEV.

Unencrypted VMCB: In [22, 58] the authors exploit the unencrypted VM register state inside the VMCB, which has been mitigated with SEV-ES.

Nested Page Tables: In [22, 44, 45, 46] the authors exploit the HV's control over the nested page tables to remap pages either leaking data or injecting code. These attacks are mitigated with SEV-SNP.

Encryption Mode: In [11] the attacker exploits the unauthenticated encryption to fault computations inside the VM by flipping ciphertext bits. [20, 60] reverse engineer the encryption mode together with the tweak values and show how this can be used to leak or inject data into the VM. However, on more recent EPYC CPUs, the updated XOR-Encrypt-XOR (XEX) [53] mode prevents the tweak reverse engineering, and SEV-SNP additionally prohibits writes to the VM's memory. [35] show that the bounce buffers required for I/O interaction between HV and VM in combination with the insufficient binding of ciphertext to its memory location prior to the XEX mode can be used to leak/inject data. Finally, [33, 36] demonstrate than even with SEV-SNP, the attacker can still exploit the fact that the memory encryption mode is deterministic to leak data through a side-channel.

Miscellaneous: In [57] the authors exploit the software-accessible power reporting features on AMD CPUs to unveil the type of executed instructions. However, the attack was only demonstrated with plain SEV, and the applicability to more recent versions is uncertain. In [12, 13] Buhren et al. show hardware-based power glitching attacks against SEV's root of trust, the Platform Security Processor (PSP), granting them custom code execution on the PSP. Further attacks on SEV versions prior to SEV-SNP also exploited flaws in the ASID-based isolation [34], in the calculation of the attestation value [61], as well as in the software interface between HV and VM [52].

2.6 Interrupt-Based Single-Stepping

The idea of improving the temporal resolution of microarchitectural attacks via triggering frequent interrupts was first explored in the context of SGX. There are several works [21, 32, 42] that significantly improved the temporal resolution from the page fault level down to a few instructions. However, reliable single-stepping was only achieved with SGX-Step [16].

While the general techniques for single-stepping SGX enclaves and SEV VMs are similar, the technical implementation is quite specific to the targeted platform. For SGX, the Asynchronous Enclave Exit (DBLP:conf/uss/ConstableBCXXAK23) mechanism can conveniently be used to place the attacker framework code close to the enclave entry/exit. For SEV, we need to modify the KVM kernel module and thus also need to implement a communication mechanism between the kernel space and user space part of our implementation. Furthermore, we need to modify the virtual interrupt delivery logic to prevent the injection of virtual APIC timer interrupts while single-stepping the SEV VM. Finally, SEV VM's usually run a fully fledged OS consisting of the Linux kernel and dozens of user space applications while SGX enclaves are more narrowly scoped. Thus, targeting a specific program inside a SEV VM is more involved.

The idea of APIC timer-based stepping was first applied to SEV in Cipherleaks [36]. However, they *did not achieve reliable single-stepping* (c.f. Figure 3 in [36]) and did not publish any code artifacts. Concurrent to our work, PwrLeak [57] also uses APIC timer-based single-stepping. However, they only performed their experiments on the outdated, plain SEV variant (not on SEV-ES or SEV-SNP) and do not provide a comprehensive framework. They also do not discuss reliability.

2.7 Cache Attacks

Since CPUs are much faster than main memory, they use caches to store recently accessed data in order to minimize latency. Modern CPUs usually use set-associative caches, where each memory address maps to a specific location in the cache, called the *cache set*. Each cache set has a limited amount of slots to store data, called *ways*. If all ways of a cache set are used, new data will evict one of the older entries. Each cache entry is identified via a unique tag value. Cache attacks use timing to infer whether a certain address is currently cached or not. As shown in many works [8, 19, 41, 51], an attacker can use this to leak secrets from other processes/entities on the system.

In *Prime+Probe* [41, 50], the attacker accesses a specifically crafted set of memory addresses, a so-called *eviction set*, to fill up a cache set. Next, the attacker waits for the victim to perform a memory access. Finally, the attacker accesses the eviction set again,

measuring the required time. A long access time indicates that the victim's memory access mapped to the same set, and thus evicted one of the attacker's entries.

The *Load+Reload* [40] attack is a more recent variation of the Prime+Probe attack. It exploits a specific behavior of the way predictor present on AMD CPUs since the Bulldozer microarchitecture: Accesses to the same physical address but with different virtual addresses always encounter a L1 data cache miss. This allows an attacker to perform the Prime+Probe step using only a single memory access for each step, irrespective of the number of ways the cache has.

Another popular cache attack is *Flush+Reload* [63]. Like with the Load+Reload attack, Flush+Reload requires shared memory between the attacker and the victim. First, the attacker uses an architectural flush command, like `clflush`, to remove the shared data/-code from the cache. As with the other techniques, the attacker waits for the victim to execute. To probe if the victim has accessed the memory location, the attacker finally measures the time required to access the flushed data with his mapping.

3 Attacker Model

In this paper, we assume a software-level attacker with full system-level privileges, which matches the threat model of AMD SEV. Using these capabilities, the attacker acts as a malicious hypervisor running a modified Linux kernel. Furthermore, the attacker can freely tweak nearly all system settings, like fixing the CPU frequency or disabling hardware cache prefetchers. However, a few features, like the availability of simultaneous multi threading (SMT) or the firmware version of the root of trust are part of the attestation report [6]. Thus their configuration status is visible to the VM owner. The attacked VMs are protected with AMD SEV-SNP. Due to SEV's attestation feature, the software inside the VM is assumed to be benign and under the VM owner's control.

4 SEV-Step Design

In this section, we first motivate the design of SEV-Step and its components, and then describe each component in-depth. The framework consists of the following main components: Single-stepping, page fault tracking and eviction set-based cache attacks.

4.1 Design Goals

We identified two major design goals for SEV-Step: Interactivity and reusability.

Interactivity: One key component for side-channel attacks in general is to precisely link the (micro)architectural observations with the victim's execution state. In the context of TEEs, like Intel SGX or AMD SEV, this is commonly achieved by interrupting the victim at defined points in its execution state, allowing the attacker to either prepare or sample the (micro)architectural state. Thus, the SEV-Step framework should not only allow the attacker to interrupt the VM, but also notify the attacker about the interruption, keeping the VM paused until the attacker signals that they are ready for the VM to resume.

Reusability: Since features like page fault tracking or programming the APIC timer require the use of certain privileged OS resources, it is natural to implement them directly inside the OS kernel. However, patching the kernel comes with several downsides. Small errors can easily lead to system crashes or hard-to-debug instabilities. Furthermore, the programming environment is limited to C, without any external libraries. Finally, recompiling the Linux kernel is quite resource-intensive, leading to long iteration times during development. Thus, we aim for a design that only implements the basic primitives that are dependent on privileged OS resources inside the kernel. These primitives are then made available to a user space library via an API allowing the development of complex attack logic in the richer and less error-prone programming environment available to user space code. Given our first goal of interactivity, this requires us to build a synchronous, bidirectional channel between the kernel space and the user space components. In addition, bundling the API in a separate library also makes it easy to separate attack specific logic from the framework code itself. This is showcased in the end-to-end attack in Section 6.1, which is a completely separate code base that only links to the SEV-Step library.

4.2 User Space API

We built SEV-Step on top of AMD's reference hypervisor implementation for SEV, which is based on the Linux KVM kernel module and QEMU.

Figure 2 shows an overview of the interaction between user space and kernel space in SEV-Step, as well as the basic workflow of the framework. The left-hand side shows the kernel space part, while the right-hand side shows the user space part. There are two communication channels between the kernel space and the user space part: ioctls and shared memory.

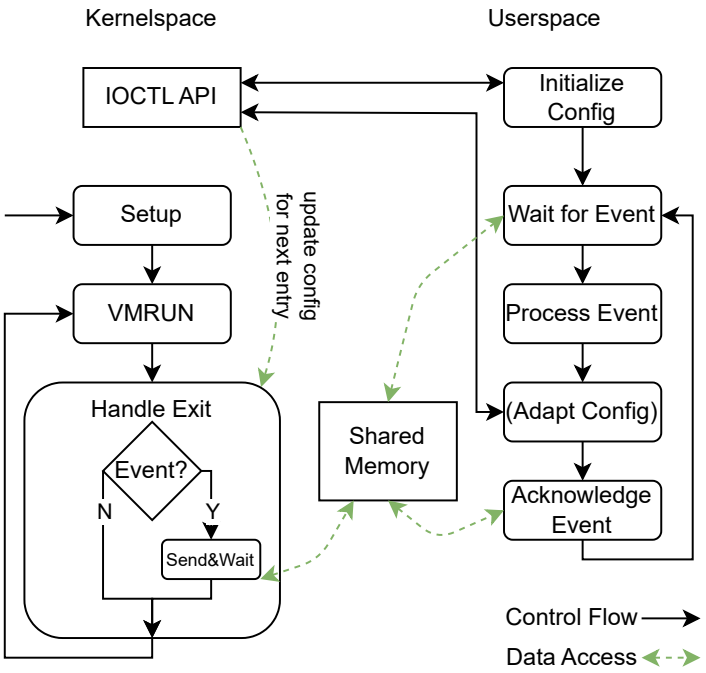


Figure 2: Overview of the kernel space and user space parts of the SEV-Step framework. There are two communication channels: An ioctl API, and communication over shared memory. Sending and acknowledging (single-stepping) events is done over shared memory. Upon sending an event, the kernel space part blocks until the event is acknowledged, delaying the next VMRUN. Both waiting for new events and for acknowledgments are implement via active polling to reduce latency. As changes to the VM can only be made upon the next exit, the ioctl API only updates a central configuration struct, deferring the application of the changes to the next exit. However, in combination with the blocking event handling, the user space library can synchronize these changes to the VM state.

Ioctls are a commonly used approach to implement kernel space to user space APIs. An ioctl is a basically a wrapper system call, that can be filled with custom behavior. However, being a system call, they require a full user space to kernel space context switch. In addition ioctls do not allow the kernel space to push events to user space. Thus, we only use ioctls for low-frequency operations, like initialization or configuration. For the high-frequency page fault and single-step event notifications, we use a custom, lightweight protocol over shared memory.

As explained in Section 2.1, the core part of the KVM hypervisor kernel module is a control loop around the VMRUN instruction. For the SEV-Step framework, we mainly add additional control logic before and after the VMRUN instruction, that, e.g., primes/probes the cache or starts the APIC timer. In addition, we also need to patch KVM's page fault handling code and overwrite the default APIC timer handling. This additional control logic can be configured via the ioctl API. As we can only reconfigure the VM between

VMRUNs, the ioctl API inherently is not synchronized with the control loop, i.e., changes only take effect on the next entry/exit from the VM. While this seems to contradict the interactivity design goal, the situation can be resolved by the blocking event notification mechanism explained in the next paragraph.

When a VMEXIT occurs due to a single-step or page fault event, the kernel space part uses the shared memory channel to deliver an event to the user space counterpart. However, after sending the event, the kernel space does not continue the execution of the VM, but instead waits for the user space to acknowledge the event, keeping the VM in a paused state. This enables the user space part to make configuration changes via the ioctl API that immediately take effect on the next VMRUN. In addition, the semantics conveyed by the page fault/single-step events allow the user space application to deduce the internal state of the VM, as required by the interactivity design goal.

To synchronize the memory accesses to the shared memory area, both sides actively poll a spin-lock. Compared to, e.g., mutexes, which might lead to an immediate reschedule when encountering an already taken lock, this results in lower overhead.

4.3 Single-Stepping

This section describes how single-stepping is implemented in the SEV-Step framework. We start by describing the basic mechanism before giving more details on tweaking the mechanism to achieve reliable single-stepping.

First, the HV uses the VCMB configuration structure, which is passed to the VMRUN instruction when entering the VM, to ensure that an APIC timer interrupt leads to a VMEXIT (c.f. Section 2.3). Next, the HV programs the APIC timer and enters the VM with the VMRUN instruction. Once the timer expires, the resulting interrupt results in a VMEXIT, handing control back to the HV. This workflow is part of the HV's regular operations, as it uses the APIC timer anyway to implement a periodic tick/callback. Next, we discuss how to use this mechanism to achieve single-stepping.

As explained in Section 2.3, the hardware does not immediately trigger a VMEXIT upon receiving, e.g., a timer interrupt. Instead, the interrupt handling, and thus the VMEXIT, is postponed until the next instruction boundary is reached. As shown in Figure 3, to achieve single-stepping, we need to configure the timer such that the interrupt is triggered before the first instruction in the VM's execution flow is finished. However, the timer interval also needs to be long enough for the first instruction of the VM to be issued into the execution pipeline. Otherwise, the VM would exit without having executed a single instruction. If the APIC timer interval is too large, multiple instructions are executed. We call these events, single-, zero- and multi-step, respectively.

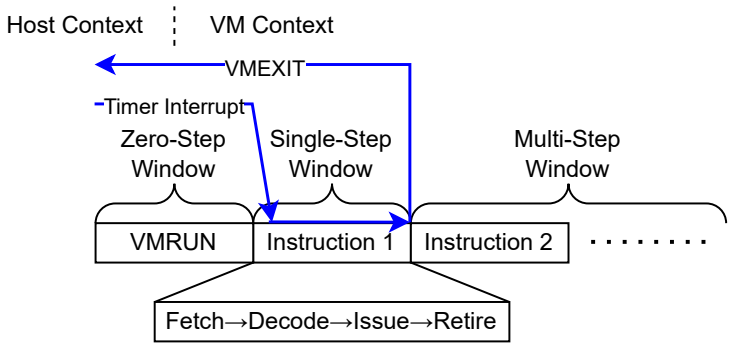


Figure 3: Timeline of the executed instructions during a HV to VM context switch. As the APIC timer interrupt is only processed at instruction boundaries, we get timing windows instead of discrete points in time, at which the interrupt leads to zero-, single- or multi-steps. The bottom row depicts that, internally, the execution of an instruction consists of several stages.

Increasing Single Step Time Window

As shown in the bottom half of Figure 3, the execution of an instruction can be decomposed into multiple parts. While the time between issuing and retiring an instruction might be very short, e.g., when executing a `nop` instruction, the other steps still require some time. This is the case especially when the CPU needs to fetch the instruction from memory and, if applicable, resolve other memory addresses used by the instruction. In our experiments, we found that the size of the single-step window is not dominated by the instruction's type itself, but rather by the instruction-agnostic execution stages (fetch, decode, . . .). To enable reliable single-stepping, an attacker must ensure that the execution of the VMRUN instruction always takes roughly the same amount of time and that the single-step window never drops below a certain threshold.

In order to maximize the time required for the first instruction executed inside the VM, we explored flushing the VM's Translation Lookaside Buffer (TLB) entries as well as resetting the "accessed" bit [3, Sec. 5.4.1] of the page containing the first instruction that would be executed after the VMRUN. By flushing the VM's TLB entries, we ensure that accessing the code page that contains the first instruction always requires a time-consuming page table walk to translate the address. The same applies to all memory operands used by the instruction. The intention behind resetting the "accessed" bit is similar. If cleared, the hardware has to set the accessed bit again [3, sec. 5.4.2]. According to the Intel SGX-specific AEX-Notify [18] paper, this requires substantial time and is one of the key factors for reliable single-stepping on Intel SGX. We evaluate the effects in Section 5.1.

Finally, we tweak the system configuration as follows to ensure a stable execution speed. We pin the kernel thread running the VM to a dedicated CPU core, that does

not run any other tasks. This is implemented via the `isolcpus`, `nohz_full`, `rcu_nocbs` and `rcs_nocb_poll` Linux kernel parameters [37]. In addition, we ensure a stable CPU frequency by either disabling dynamic frequency scaling altogether (if the BIOS permits it), or by pinning the CPU frequency using the Linux `cpufreq` subsystem [62]. Finally, we disabled hardware cache prefetchers in the BIOS. Since SEV aims to protect against a privileged system-level attacker, all of these changes are within the threat model.

Preventing Virtual Timer Interrupts

As explained at the start of Section 4.3, the Linux OS uses the APIC timer to implement a periodic tick/callback. Thus, while the HV handles the physical APIC timer interrupts, it also needs to emulate the interrupt for the VM. As explained in Section 2.3, AMD's hardware assisted virtualization offers the concept of virtual interrupts to achieve this. Thus, whenever the APIC timer interrupts the VM, the KVM hypervisor would usually inject a virtual timer interrupt into the VM upon the next VMRUN. As a consequence, the Linux OS in the VM jumps to its corresponding interrupt handler. As a result, an attacker would not single-step any user code, but only the VM's APIC timer interrupt handler. Thus, we need to modify this part of KVM's logic to prevent any virtual timer interrupt injection while we single-step the VM. For our attacks, we did not observe any instabilities in the VM's execution due to the inhibited interrupt. As a workaround for potential issues with very long single-step phases, we could periodically allow the injection of the virtual timer interrupt.

Determining the Step Size

To properly determine the APIC timer timeout value, we need a feedback channel enabling us to observe the amount of instructions executed by the guest. In SGX-Step [16] the "accessed" bit of the page table entry corresponding to the page containing the current instruction is used to differentiate single-steps and zero-steps. However, it cannot be used to detect multi-steps, which is only possible by running the enclave in debug mode to observe its instruction pointer. While these two methods also work in SEV, we additionally have access to the VM's performance counter events. As demonstrated in [33], there is a performance counter for retired instructions that can be configured to only consider instructions executed by the VM. Thus, evaluating the counter before and after entering the VM immediately reveals the step size.

4.4 Page Fault Tracking

For page fault tracking, SEV-Step uses the well-known control of the HV over the nested page tables [34, 45, 60]. By modifying the *present*, *no-execute* and *read/write* bits of a page, the HV can force the VM to encounter a page fault that also reveals the type of access. While being more coarse-grained than single-stepping, page fault-based tracking is significantly faster. Thus, for many attack scenarios, it is beneficial to rely on the coarse-grained page fault mechanism as much as possible before enabling single-stepping. For example, an attacker could use page fault tracking to get notified when the VM is about to execute a code page containing a series of secret-dependent memory lookups. Only then, the attacker activates single-stepping, allowing them to, e.g., perform a cache attack against each individual memory access.

4.5 Cache Attacks

To use SEV-Step's cache attack capabilities, the attacker first needs to perform some initial configuration like locating and defining the cache attack targets. Afterwards, while single-stepping the VM, the attacker can request that a cache attack is performed for the next single-step. The resulting step event is enriched with the measured data.

In the remainder of this section, we discuss how we measured execution times on our system as well as the applicability of the Prime+Probe, Load+Reload and Flush+Reload (c.f. Section 2.7) cache attacks in the context of SEV.

Measuring Access Times

As, e.g., discussed in [40], the `rdtsc` and `rdtscp` instructions return very coarse-grained timing data on AMD CPUs since the Zen microarchitecture. This makes them unsuitable for cache attacks without averaging over several iterations. Prior work suggests either using a so-called counting thread [39] or the `rdpru` instruction [4, 38, 40]. While the latter could be disabled for unprivileged users, we assume an attacker with kernel-level privilege. In the following sections, we use the `rdpru` instruction due to the lower footprint on the microarchitectural state compared to the counting thread.

In addition to measuring the access time, we can also use performance counters to gather information about the cache state. As described earlier (c.f. Section 4.3), SEV does not offer protection/isolation for performance counters. For the level 2 (L2) cache, there are performance counters for "L2 Cache Miss from L1 Data Cache Miss" and "L2 Cache Hit from L1 Data Cache Miss" [5, Sec. 2.1.17.2]. However, as there is no performance counter for L1 data cache hits or misses, we still require the access time to infer the L1

result in order to interpret the L2 events. E.g., if we have a L1 cache hit, the difference in both counters would be zero, leaving us with an inconclusive result until evaluating the access time.

Flush+Reload

The HV can easily obtain a mapping to any of the VM's memory pages by using the nested page tables. However, as explained in [34], in SEV the cache tag is extended with the current ASID and the encryption status of the corresponding page (C-Bit), effectively allowing the same data to reside in the cache multiple times. As the HV has a different ASID than the VM (as discussed in [34] the HV could technically change its ASID, but this would basically prevent it from executing any further code), it cannot get a hit on the data brought into the cache by the VM, when accessing the data via its own mapping. Thus, the HV cannot perform the reload part of the Flush+Reload attack.

Load+Reload

As flush-based attacks do not work with SEV, we need to look at eviction-set based approaches. One particularly efficient method is the AMD-specific Load+Reload attack, as it only needs a single memory access in each stage. In the original paper [40], the authors only demonstrated the Load+Reload attack in the context of one user space process attacking another. We were able to reproduce the attack with a malicious hypervisor attacking a regular (non-SEV) VM. However, when targeting any type of SEV VM (plain, ES, SNP), the observed effect on the cache changes. Instead of getting an L1 data cache miss and an L2 hit for the evicted address, we observed RAM access times for the *whole memory page* to which the evicted address refers. We were not able to conclusively verify the cause for this behavior. However, we suspect that this is related to the "Cache Coherency across encryption Domains" feature [3, Sec 15.34.9] available on our CPU. The manual states that without this feature, the HV is required to manually flush a data page of the VM before accessing it, if it wants to read the latest data. Thus, it is possible that the HV's access to the aliased mapping also internally triggers a cache flush.

Prime+Probe

As the specialized eviction-set technique of the Load+Reload attack does not work with SEV, we opted for the generic Prime+Probe attack. To reduce cache noise, we chose to implement the prime and probe steps in the kernel space. This way, they can be placed immediately before and after the VMRUN instruction. The eviction set finding itself is implemented in user space, to allow for maximal flexibility. We found that on our CPU

Table 1: Results of single-stepping the same `nop` slide program while: Resetting the "accessed" bit before each step (A-Bit), flushing the guest TLB before each step (TLB), doing both (TLB + A-Bit). For the rows with multi-steps, the timer value is the smallest value that did not only produce zero-steps. ⌀ M-Step denotes the average amount of instructions executed during a multi-step.

	Timer	0-Step	1-Step	M-Step	⌀ M-Step
BASELINE	0x31	6401	1534	32	37
A-BIT	0x31	6399	1548	50	34
TLB	0x33	1158	4000	0	0
TLB + A-BIT	0x33	1116	4000	0	0

the first 24 bits of the page frame number need to be equal for two pages to be mapped to the same L2 cache set. Next, the user space application passes the virtual addresses of the eviction set(s) to the kernel space component, which will create internal mappings to the used pages. The additional kernel mappings are required as the address space of the user space application will not be mapped during the prime and probe steps performed immediately before and after the VMRUN instruction.

5 Evaluation

We evaluated SEV-Step on a Dell PowerEdge R6515 Server with a 3rd generation EPYC 7763 CPU. The attacked VM was protected with SEV-SNP, running Ubuntu 22.10 with an unmodified Linux 5.19.0-26 kernel (starting with 5.19, the mainline Linux kernel supports running as a SEV-SNP guest). The attacker-controlled host is running our modified SEV-Step kernel that is based on AMD's patched Linux 5.14 kernel. The SEV-Step framework, as well as the code for the evaluation and attacks presented in this paper, is available at `https://github.com/sev-step/sev-step`.

5.1 Single Step Reliability

For the reliability evaluation, we analyzed four different scenarios, based on the ideas described in Section 4.3. The results are shown in Table 1. We define reliability as "not performing multi-steps" while still performing some single-steps. Starting with an initial guess for the timer value, we iteratively decrease it until any further decrease would result in only performing zero-steps. For all scenarios, we try to single-step a code block consisting of 4000 `nop` instructions.

In the baseline scenario, we try to achieve single stepping only using the APIC timer, i.e., without combining it with other (micro)architectural tweaks. As depicted in the table, this approach fails. Setting the timer value to $0x30$ results in only zero-steps but $0x31$ already gives us 32 multi-steps.

Next, we analyze the effect of resetting the "accessed" bit as well as flushing the VM's TLB entries. As explained in Section 4.3, the intention behind these tweaks is to increase and homogenize the timing window leading to a single-step. Resetting the "accessed" bit does not have any significant effect. However, flushing the VM's TLB drastically improves the situation, enabling us to execute the targeted program without any multi-steps. As expected, combining both methods does not yield a significant improvement.

In addition to the slide of `nop` instructions discussed here, in Section 6.1, we single-step the real world Linux kernel AES encryption and decryption code as well as a more diverse set of instruction microbenchmarks.

5.2 Event Handling Performance

To evaluate the performance of the event sending mechanism, we compare handling all events in kernel space with sending them to user space. As both page faults and single-steps use the same basic mechanism, we restrict our analysis to the code path sending single-step events. For simplicity, we again use the `nop` slide program introduced in Section 5.1. Note that the performance of the event mechanism is independent of the stepped instructions. Without sending events to user space, we require on average 1.007 ms per single-step event, with a standard deviation of 0.0054 ms. Sending events to user space requires an average 1.616 ms per single-step event, with a standard deviation of 0.01548 ms. While the user space event handling requires roughly 60% more time, we believe this overhead is acceptable given the substantial improvements in usability and attack development.

5.3 Cache Attack

We now evaluate SEV-Step's Prime+Probe attack implementation. As discussed in Section 4.5, the Load+Reload and Flush+Reload attacks do not work with SEV. We tested the Prime+Probe attack against both the first level data cache (L1D) and the level two cache (L2). However, as the L1D showed a high amount of noise, we only evaluate the L2 variant here.

For the experiment setup, we assume that the guest physical addresses of both the test program and a given lookup table have already been recovered by the attacker,

e.g., by using the page fault side-channel in combination with the "retired instructions" performance counter [33]. Next, we use page fault tracking to detect when the code is about to be executed, and then start single-stepping to interrupt the code immediately before and after each memory access to the lookup table. We analyze two variants of a crafted assembly snippet that alternates between accessing offset 64 (byte) and 960 (byte) in a cache line-aligned $16 \cdot 64$ byte lookup table (similar to the T-tables attacked in Section 6.1). In the first variant, we placed a `lfence` instruction between the memory accesses, while for the second variant, the memory accesses are performed back-to-back. Then we classify the data using a previously determined timing threshold. In the first variant, we get a success rate of 0.94, while for the second variant we only get a success rate of 0.13. While the second variant does indeed have higher cache noise, upon closer examination, one of the "noisy" cache sets is often related to the next upcoming memory access, i.e., despite the fact that we are single-stepping, future memory accesses are already fetched out-of-order and thus leave a cache trace. We discuss these effects in more detail in Section 6.1, where we demonstrate an end-to-end cache attack against the Linux kernel's AES implementation. We did not observe any cache trace when zero-stepping an instruction, indicating that the context switch needs to fully complete before any instructions from the VM are issued to the execution pipeline.

6 Case Studies

To demonstrate the capabilities of the SEV-Step framework, we performed two case studies. In the first one, we explore the common workflow of using a SEV VM in combination with an encrypted disk image. We show how an attacker can use the cache attack and single-stepping features of SEV-Step to recover the AES volume key of a disk encrypted with LUKS2. In the second case study, we analyze to which degree SEV-protected VMs are vulnerable to Nemesis-style attacks [15]. For this, we enrich the single-step events with precise time measurements. To the best of our knowledge, these kinds of attacks were not explored in the context of SEV before.

6.1 Cache Attack on Disk Encryption

We show an end-to-end, single trace cache attack that is able to steal the volume key of a disk encrypted with cryptsetup+LUKS2, which is a disk encryption system commonly used with Linux. First, we briefly introduce disk encryption, which is a highly relevant workflow for SEV and confidential VMs in general. Next, we describe how we can force the disk encryption system to decrypt the disk using a cipher implementation vulnerable

to cache attacks. Finally, we explain the technical details required for gathering the cache traces and how we recovered the volume key from them.

Linux Disk Encryption

A common approach for deploying SEV VMs is providing the HV with an encrypted disk image and a bootloader. The bootloader is attested through the SEV API and receives the disk password from the user. It then opens the disk image, loads the kernel binary into memory, and transfers control to the kernel. Finally, the kernel unlocks the disk image again and mounts the contained file system. This workflow allows to keep the attested initial code image small, improving performance and reducing the attack surface.

Under Linux, the disk encryption infrastructure [47] is split into a user space and a kernel space part. The kernel contains the disk driver and implementations for several ciphers that can be consumed by user space applications via an API. An example for this is the popular full disk encryption suite *cryptsetup*.

The kernel crypto infrastructure provides a flexible architecture of basic ciphers and so-called "templates". The former are plain block ciphers (or message digests), the latter implement additional logic on top of existing ciphers. This is commonly used to represent block cipher modes like CBC or XTS. Part of the kernel crypto API is the *CAPI* specification format, that allows to describe composed ciphers in a structured manner. For example, `capi : xts(ecb(aes)) − plain64` invocates the XTS driver with AES in ECB mode and a sector number-based IV generator.

Since Linux supports a wide range of architectures, there may be different variants of a cipher, each optimized for a certain architecture version. Each implementation is assigned two names: The `cra_name`, which is equal for all implementations of a given primitive, and the `cra_driver_name`, that uniquely identifies a specific implementation. The CAPI format supports both names. If a `cra_name` is provided, a static scoring system is used to select the best implementation for the current system. If a `cra_driver_name` is specified, the kernel uses that specific implementation if available.

The LUKS2 [10] format commonly used with cryptsetup allows specifying the block cipher for the encrypted disk in the CAPI format. As this value is neither encrypted nor authenticated, it can be arbitrarily manipulated, as, e.g., shown in [56]. The CAPI string is directly passed to the kernel crypto API. We discovered that the Linux kernel shipped with Ubuntu 22.10 contains several symmetric cipher implementations that are highly vulnerable to cache attacks. In the next section, we show two approaches how a malicious hypervisor can combine these weaknesses, by first tricking the VM into using

its secret disk encryption key with a vulnerable algorithm and then extracting the key from the resulting leakage via a cache attack.

Forcing Vulnerable Ciphers

On our test systems, cryptsetup defaults to `capi : xts(ecb(aes)) − plain64` for LUKS2 encrypted disks. XTS [23] is a tweaked block cipher mode commonly used for disk encryption. It uses two keys: The first key is used to generate a so-called tweak value by encrypting the current disk sector number. This tweak value, multiplied with a number representing the current offset inside the disk sector, is then XORed to the actual payload data before and after encrypting/decrypting it using the second key. Using the weaknesses described in the previous section, the malicious HV changes the disk header to `capi : xts(ecb(cast6)) − plain64` before passing the disk to the VM, tricking the VM into using the vulnerable CAST6 implementation. As the disk content was initially encrypted with AES, this does not yield meaningful plaintext, preventing the disk from being mounted properly. Nevertheless, the decryption routine is still invoked roughly 66k times before the mount operation eventually fails, providing sufficient opportunity to leak the key.

A second, more stealthy approach, is exploiting the ability to force a specific cipher implementation. The attacker replaces the `capi : xts(ecb(aes)) − plain64` specification by `capi : xts(ecb(aes − generic)) − plain64`, such that the AES cipher is instantiated with a leaky T-table based implementation. While we verified that such substitutions work for "templates"/composed ciphers, there is one remaining problem when applying it to the XTS implementation in Linux. Only the cipher instantiation for the payload data encryption/decryption is selected based on the exact value specified in the CAPI string. The cipher instantiation for the tweak generation always uses the priority-based `cra_name` to select the best implementation. As all SEV-enabled systems support AES-NI, this prevents us from leaking the tweak encryption key. Thus, during the key recovery, we cannot recompute the tweak value which is a prerequisite for recovering the second key, used for encrypting the actual payload data. However, as shown in the next paragraph, a malicious HV can suppress the availability of AES-NI altogether, forcing the VM to use the vulnerable implementation for both instances of AES.

The VM uses the `cpuid` instruction to determine whether AES-NI is available. As the HV can intercept this instruction, it can arbitrarily manipulate the reported features. All versions prior to SEV-SNP cannot detect such manipulations. With SEV-SNP, a new mechanism was added to provide trustworthy `cpuid` information [6]. During the attestation process, the HV has to commit to a set of `cpuid` bits, that are additionally verified by the AMD Platform Security Processor (PSP). Depending on the specific `cpuid` entry, the

PSP enforces different policies. Some entries are required to match the value on the host, but the AES-NI feature is allowed to be disabled [5, Sec. 2.1.5.3]. Thus, the VM owner has to be aware of the subtle security implications of disabling AES-NI and ensure that their expected attestation value enforces enablement of AES-NI.

As the technical aspects of the cache attack are similar and the AES key recovery is more interesting, we opted for AES in our end-to-end attack. This also leaves the possibility that the VM owner remains unaware of the attack, as the disk mount succeeds.

Performing the Cache Attack

In preparation for the cache attack, we need to solve three challenges: We have to ① locate the AES code and detect its execution, ② locate the instructions accessing the AES T-tables, and ③ locate the AES T-tables.

Similar to prior work [33, 36, 45], we solve ① through the page fault controlled channel. As explained in the previous section, the decryption of a single XTS-encrypted data block consists of two AES invocations using different keys. To build the page fault sequence fingerprint, we trace all of the kernel's page accesses while triggering disk decryption operations. By manual analysis we found that the page fault sequence in Table 2 uniquely identifies the execution of the relevant AES functions. While KASLR randomizes the location of the kernel's `.text` section at each boot, the contents and order of the `.text` section itself are not randomized. Furthermore, there are several techniques to break KASLR in the SEV context [46, 60]. Thus, we encode the page fault sequence relative to the start of the `.text` section instead of using absolute addresses, allowing its usage across reboots. Note that for the final attack, it suffices to track the pages of the sequence one by one, i.e., we no longer need to trigger a page fault on every memory access as we did while generating the fingerprint.

For ②, we first analyze the assembly code of the AES functions in an offline phase. This allows us to build a list of all instructions accessing the T-table, each annotated with the number of instructions executed since the start of the function. Then, during the attack, we single-step the VM's execution once we reach the targeted AES functions. By comparing the number of executed steps with the information gathered in the offline phase, we know whether we need to perform the cache attack for the next instruction.

To ③ locate the AES T-tables, we "sacrifice" the first memory access instruction of the encrypt/decrypt AES functions: Instead of performing the cache attack, we mark all pages as not present, yielding a list of all pages accessed during the execution of the instruction. We empirically verified that the final page fault before the instruction's retirement corresponds to the page of the T-table.

Table 2: Page fault sequence uniquely identifying the execution of the AES encrypt and decrypt operations performed during the decryption of a single payload data block of the VM's disk. Accesses with the "Marker" role don't correspond to an operation that we want to observe, but are required to accurately track the execution flow. The "PFN Offset" field states the offset of the page containing the function relative to the start of the kernel's `.text` section (measured in 4096 byte pages).

Name	PFN Offset	Role
`xts_decrypt`	0x65c	Marker
`crypto_cipher_encrypt_one`	0x64b	Marker
`crypto_aes_encrypt`	0x65f	Tweak generation
`crypto_aes_encrypt`	0x660	Tweak generation
`crypto_ecb_decrypt`	0x65b	Marker
`crypto_aes_decrypt`	0x660	Payload decryption
`crypto_aes_decrypt`	0x661	Payload decryption

With the preparation done, we can now single-step the encryption/decryption functions, and perform a L2 Prime+Probe attack on each T-table access. As a T-table has 256 4-byte entries and thus covers 16 cache lines, we need to measure 16 cache sets for each access.

Recovering the AES Key

Given the cache measurements, we now conduct an offline analysis to recover the two AES keys used by the AES-XTS disk encryption. First, we discuss how we overcame the challenge of out-of-order accesses in our cache traces. Afterwards, we describe our key recovery algorithm.

Although the VM's execution is single-stepped during the cache attack, we still observe a high amount of cache noise, as shown in Figure 4. Upon closer examination, most of the cache noise is correlated to future (out-of-order) memory accesses to the same T-table. While those are not actually retired due to the APIC timer interrupt used for single-stepping being processed immediately after the current instruction, their cache traces persist. This reasoning is supported by our experiment in Section 5.3, where we analyzed a synthetic cache attack victim with and without `lfences` between the memory accesses and found that the version with fences does not show this behavior. For our attack, a given memory access usually influences up to four preceding accesses to the same lookup table. This matches the round structure of AES, where each T-table is accessed four times with a data dependency between the accesses of different rounds.

As we also have actual cache noise, as well as occasional accesses to the same cache set within four memory accesses, separating the actual access from the noise proved

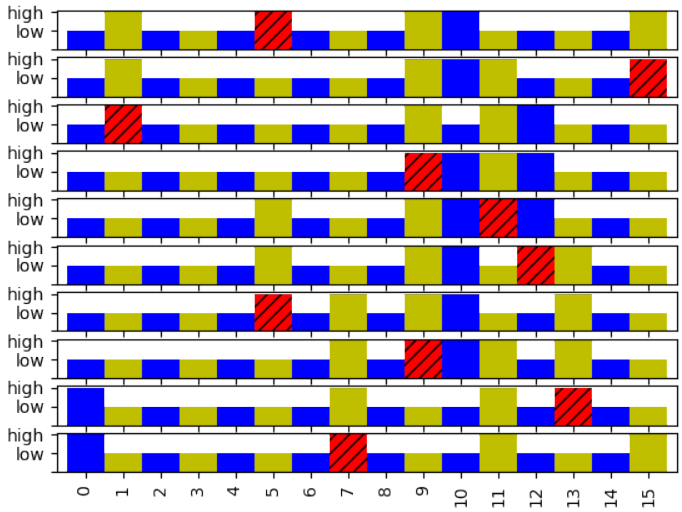


Figure 4: First 10 accesses to the first T-table of `crypto_aes_encrypt`. The X axis shows the cache sets covering the T-table, indexed from 0 to 15. The bars on the Y axis show if the cache set is considered high or low for that access. For each memory access, the actually expected cache set is striped and colored red. Due to out-of-order execution, each expected cache set leaves a trail of high cache sets in the preceding accesses.

challenging. We opted for a machine learning-based approach with a sequential neural network model consisting of 3 dense layers with 182, 64 and 16 neurons, respectively, as well as two dropout layers to enhance generalization by preventing overfitting. For the input encoding, we map each memory access to a binary vector, containing the cache traces of the access that we want to classify as well as for the 8 preceding and subsequent accesses. We use 8 instead of 4 accesses in each direction to better improve handling of situations where two close-by memory accesses use the same cache set. We label each input with a one hot encoding of the expected memory access. For the first and last 8 accesses, we use zeroes to fill up the missing preceding/subsequent accesses. Table 3 shows the accuracy of the classifier in our experiments. The cache traces for crypto_aes_decrypt classification contain a significantly higher amount of noise, leading to a worse classification.

For our key recovery, we use XTS decryptions for disk offsets that have known or easily guessable plaintext and that are always accessed during a mount operation. This includes certain magic offsets that are searched for file system headers, and the file system structures themselves. In the first step, we break the key that is used for encrypting the IVs (sector numbers), yielding the tweak. In the second step, we remove the tweak from the ciphertext and then break the key used for decrypting the payload. To break a key, we first guess a number of bits and then check whether that guess is consistent with the

Table 3: Accuracy of the ML-based classifier for the recorded AES cache traces. We trained a dedicated model for each T-table. For crypto_aes_encrypt we used approximately 60k training and 11k testing samples. For crypto_aes_decrypt we used approximately 74k training and 14k testing samples. The varying amount is due to outlier removal.

Target	Accuracy
crypto_aes_encrypt - Lut 0	0.9050
crypto_aes_encrypt - Lut 1	0.9037
crypto_aes_encrypt - Lut 2	0.8971
crypto_aes_encrypt - Lut 3	0.8610
crypto_aes_decrypt - Lut 0	0.6771
crypto_aes_decrypt - Lut 1	0.6860
crypto_aes_decrypt - Lut 2	0.6919
crypto_aes_decrypt - Lut 3	0.2545

T-table measurements, before guessing the next bits. This way, we can discard enough candidates to avoid searching the entire key space. By ordering the candidate cache sets by the probability that is returned by the classifier and discarding measurements with more than 7 candidates, the time needed for finding the correct key can be further reduced.

End-to-End Attack

To test our attack implementation, we created a LUKS2 disk with an `ext4` filesystem and a random encryption key. We manipulated the header as described to call the vulnerable `aes − generic` implementation in the kernel, and disabled AES-NI in the VM. When the kernel running inside the SEV VM starts mounting the encrypted disk, we execute steps ① to ③ to locate the relevant instructions and data structures. We continued with tracing 70 XTS decryptions, from which 34 involved a known plaintext, applied the classifier to the measured cache accesses, and then invoked the key recovery. Due to the cache noise issues described in the previous section, our key recovery requires roughly 13 hours on the 96-core EPYC 7763 CPU that we used throughout the evaluation. Thus, while not computationally trivial, the attack is feasible. Note that our attack required a *single mount operation*, making the attack hard to detect and evade.

6.2 Instruction Latency Attack

As a second case study, we analyzed whether the interrupt timing-based *Nemesis* attack [15] also applies to SEV. The core idea of the Nemesis attack is to use the time

between single-steps to infer the type of instruction executed, or extract information about its operands. The correlation between the time required for a single-step and the executed instruction stems from the fact that the interrupt used to drive the single-stepping is only processed on instruction boundaries. Thus, the single-step timing correlates to the time required by the executed instruction. The Nemesis paper analyzed this attack vector for Intel SGX and the Sancus enclave on a TI MSP430 microcontroller. To the best of our knowledge, we are the first to analyze this attack vector on AMD SEV.

Measuring Latency

For measuring the latency of a single-step, we use the `rdpru` instruction to read the Actual Performance Frequency Clock Count (`APERF`) MSR, as discussed in Section 4.5. For older Zen processors (prior to Zen 2), the `APERF` MSR can be read with `rdmsr` instead of `rdpru`. As depicted in Figure 5, we obtain a timestamp as close as architecturally possible before and after the VMRUN instruction. The `sti` instruction in line 14 is required by the virtualization interface and sets `RFLAGS.IF` to 1, enabling maskable external interrupts. However, it only takes effect after the next instruction has executed, thus our timing measurement cannot be disturbed by interrupts before entering the VM. When leaving the VM, the hardware automatically sets the global interrupt flag (GIF) to 0. This flag disables external interrupts and thus no such interrupt can trigger between line 15 and 16. As a result, our timestamp code runs in line 19 even before the handler for the APIC timer interrupt that caused the VM exit is executed. The measurement code itself imposes a minimal overhead by storing the timestamps prior to executing `VMRUN`.

Differentiating Instructions

To empirically test the distinguishability of individual x86 instructions based on their latencies, i.e., the difference between the timestamp prior to and directly after `VMRUN`, we perform experiments in the form of microbenchmarks similar to those of Nemesis [15]. We execute an instruction slide of 1,000 assembly instructions and collect the latencies of each single-step. We repeat this procedure 100 times for a total of 100,000 measurements. Unlike with SGX-Step, we do not need to check the "accessed" bit in the page table entry to filter for zero-steps, but can directly use performance counters to evaluate the number of zero, single- and multi-steps, as described in Section 5.3. For our analysis, we pick instructions with a range of latencies based on benchmarks done by Abel et al. [1].

Figure 6a shows the latency distributions of a selection of x86 instructions. Using SEV-Step, we can distinguish low-latency instructions such as `add` or `mul` from high-latency instructions such as `rdrand` or `lar`. We also note that, while instructions such as `nop`, `add`

```
1  ; start APIC timer
2  movl %edx, (%r8)
3  ; timestamp before VMRUN
4  lfence
5  movl $1, %ecx
6  rdpru
7  shl $32, %rdx
8  or %rdx, %rax
9  lfence
10 ; save timestamp to stack
11 push %rax
12 ; Prepare VMCB arg
13 ; and enable interrupts
14 mov %rdi, %rax
15 sti
```

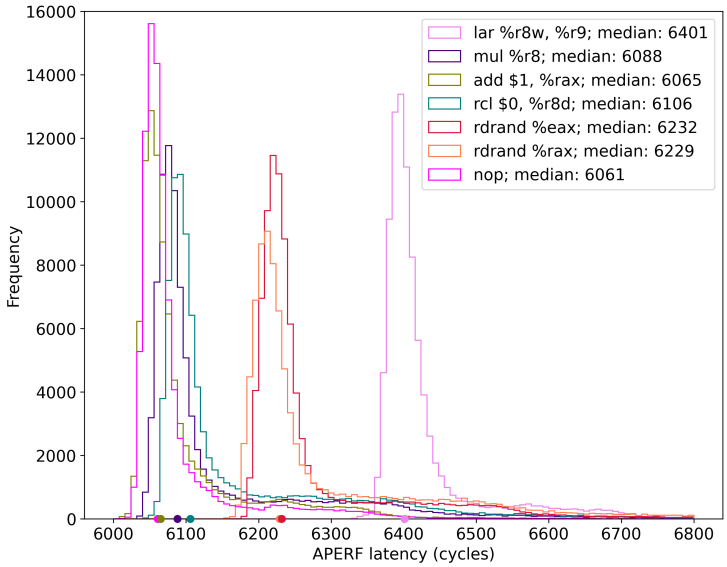
```
16 ; Enter VM
17 vmrun %rax
18 ; Execution resumes here
19 ; after VMEXIT
20 cli
21
22 ; timestamp after VMRUN
23 lfence
24 movl $1, %ecx
25 rdpru
26 shl $32, %rdx
27 or %rdx, %rax
28 lfence
29 ...
```

Figure 5: Assembly code for measuring the time for a single-step event. To reduce system noise to a minimum, we place the time measurement directly inside the kernel space hypervisor code and as close as possible to entering and leaving the VM.

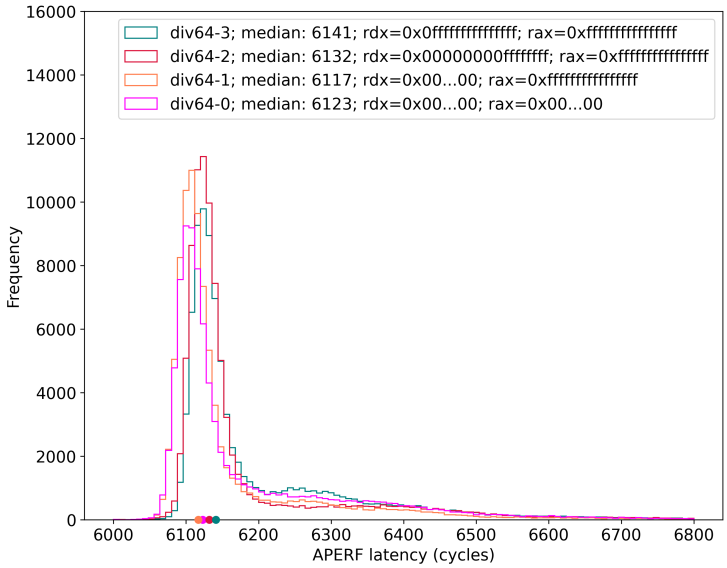
or `mul` are harder to discern due to their similar latencies and micro-ops, we can still distinguish the average execution time given sufficient repetitions.

6.2.1 Differentiating Data Operands

For determining the distinguishability of data dependent operations, we replicate the experiments of Nemesis regarding unsigned integer division `div` with the divisor `rbx` fixed to `0xffffffffffffffff` while varying the dividend `rdx` : `rax`. According to the AMD hardware optimization manual, the "hardware integer divider unit has a typical latency of 8 cycles plus 1 cycle for every 9 bits of quotient" [7, p. 36]. This suggests that there should also be a correlation between the significant bits of the dividend and the measured latencies. As can be seen in Figure 6b, while the median latencies for $\texttt{div64} - 1$, $\texttt{div64} - 2$ and $\texttt{div64} - 3$ increase with the size of the dividend, $\texttt{div64} - 0$ does not follow this trend. We also observe that the latency differences between the `divs` are significantly less prominent than those reported for SGX in the Nemesis paper.



(a) Latency distribution of a selection of x86 instructions.



(b) Latency distribution of the x86 `div` instruction with varying dividend operands and the divisor `rbx` fixed to `0xffffffffffffffff`.

Figure 6: Latency microbenchmarks with 100,000 executions of each instruction.

7 Discussion

7.1 Zero/Single-Step Countermeasures

There are several works that try to protect SGX enclaves against single-stepping-based attacks [17, 30, 31, 49, 55], but none of them found widespread adoption. In 2022 Intel in collaboration with researchers [18] from the academic community released the AEX-Notify extensions [24] [25, p. 199-204] for SGX that make the enclave interrupt-aware, allowing it to execute custom handler code before resuming at the interrupted instruction. The AEX-Notify paper [18] uses this interrupt awareness to execute a code gadget that aims to ensure that the first payload instruction of the enclave will execute fast by ensuring that the instruction as well as its operands are fully cached. This way they aim to prevent reliable zero-/single-stepping.

According to the Intel TDX Module Spec [27, Sec 17.3], TDX has been designed with countermeasures for zero-/single-stepping attacks. To prevent single-stepping attacks, a trusted domain (equivalent to SEV VM) can still execute a small randomized amount of instructions if it gets interrupted within approximately 4k cycles after being entered. To additionally prevent zero-steps via missing page table permissions, the TDX module limits the number of page faults that may occur without forward progress and thus forces the HV to ensure proper page table configuration before it can resume the trusted domain.

Given the novelty of single-stepping attacks against AMD SEV, we are not aware of any countermeasures. In contrast to the AEX-Notify countermeasure that has to cope with the architectural limitations of SGX, the TDX approach seems more principled. However, in contrast to the TDX design, for SEV there is no trusted layer between the HV and the VM that could e.g. prevent the VM from being entered after a certain amount of faults without forward progress. We leave the design of countermeasures to future work.

7.2 Preventing Vulnerable Algorithm Selection

As demonstrated in Section 6.1, an attacker can exploit the unmodified LUKS2 header in combination with the Linux kernel's expressive CAPI specification language, to trick the VM into decrypting its disk using cryptographic implementations vulnerable to side-channel attacks. One possible solution is to remove all vulnerable implementations from the Linux kernel, and replace them by constant-time code. If this is deemed unpractical, the API should flag all vulnerable implementations as such and provide a way to allow its users to explicitly disallow their usage. Another strategy would be to add a checksum preventing the LUKS2 header manipulation. However, that case would

require to explicitly specify an implementation for the crypto algorithm. Otherwise, the kernel's priority-based system might still select a vulnerable implementation under certain system configurations.

We used a side-channel leakage analysis tool [59] in combination with a custom QEMU plugin to analyze the Linux kernel's crypto primitives for the secret oblivious memory access and constant time properties. Due to limitations of QEMU, we were not able to analyze AVX-based implementations. We found significant leakages in many other symmetric ciphers, for example *aes-generic*, *aes-fixed-time*[1], *blowfish-asm*, *blowfish-generic*, *camellia-asm*, *camellia-generic*, *cast5-generic* and *cast6-generic*.

We disclosed our findings regarding the LUKS2 header manipulation and its impact on using LUKS2 in the context of confidential VMs to the cryptsetup/LUKS2 team[2]. As a result, they changed the CAPI parsing part of cryptsetup to disallow the selection of specific implementations. However, this does not help if all implementations known to the Linux kernel are vulnerable, as it is the case for the blowfish cipher.

8 Conclusion

In this paper, we have demonstrated that SEV-SNP VMs can be reliably single-stepped, which greatly increases their vulnerability against a wide range of microarchitectural side-channel attacks. In the hope to ease future research in this direction, we introduced SEV-Step, a reusable framework allowing the development of complex attacks from user space. We have demonstrated the framework's capabilities with two in-depth case studies. The cache attack against the Linux disk encryption infrastructure revealed that even with SEV-SNP, the implementation of protected VMs remains brittle due to continuing prevalence of vulnerable code. The clash between the attacker model for which these systems have been designed with their usage in the context of confidential VMs exposes them to powerful software-level attacks in virtualized environments. Given that not only AMD SEV but also Intel TDX [26] and ARM CCA [9] employ the confidential VM model, their security under this new threat model should be analyzed with more scrutiny. Finally, in the second case study, we have demonstrated that SEV is vulnerable to timing-based instruction classification. Like the Nemesis attack on SGX, we were able to confirm that instruction sequences can be reconstructed in SEV. While the timing variation is smaller than in SGX, repeat measurements can reveal even small variations due to data-dependent execution time of instructions such as `div`.

[1] Despite the name, this is simply the *aes-generic* implementation with interrupts disabled (from the VM's point of view). The external APIC timer interrupt used for single-stepping is not influenced by the VM's internal interrupt enablement status, so our attacks would still work.

[2] `https://gitlab.com/cryptsetup/cryptsetup/-/issues/809`

Acknowledgments

This work has been supported by the DFG under grant 456967092 and by the BMBF through projects ENCOPIA and SASVI.

References

[1] Andreas Abel and Jan Reineke. "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. 2019. DOI: 10.1145/3297858.3304062. URL: https://doi.org/10.1145/3297858.3304062.

[2] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. 2020-01.

[3] AMD. "AMD64 Architecture Programmer's Manual Volume 2: System Programming". In: *Manual* (2023-01). Rev 3.40.

[4] AMD. "AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions". In: *Manual* (2022-10). Rev 3.34.

[5] AMD. "Preliminary Processor Programming Reference (PPR) for AMD Family 19h Model 01h, Revision B1 Processors Volume 1". In: *Manual* (2021-05). Rev 0.50.

[6] AMD. *SEV Secure Nested Paging Firmware ABI Specification*. Tech. rep. 56860, Rev. 1.55. AMD, 2023-09.

[7] AMD. "Software Optimization Guide for AMD EPYC(TM) 7003 Processors". In: *Manual* (2020-11). Rev 3.00.

[8] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. "S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015. DOI: 10.1109/SP.2015.42. URL: https://doi.org/10.1109/SP.2015.42.

[9] ARM. *Introducing Arm Confidential Compute Architecture*. https://developer.arm.com/documentation/den0125/latest. Revision 0300-01. 2023-06.

[10] Milan Brož. *LUKS2 On-Disk Format Specification*. https://gitlab.com/cryptsetup/LUKS2-docs/blob/main/luks2_doc_wip.pdf. Version 1.1.1. 2022-07.

[11] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. "Fault Attacks on Encrypted General Purpose Compute Platforms". In: *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ , USA, March 22-24, 2017*. 2017. DOI: 10.1145/3029806.3029836. URL: https://doi.org/10.1145/3029806.3029836.

[12] Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean -Pierre Seifert. "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3484779. URL: https://doi.org/10.1145/3460120.3484779.

[13] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. "Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019. DOI: 10.1145/3319535.3354216. URL: https://doi.org/10.1145/3319535.3354216.

[14] Jo Van Bulck. *SGX-Step Repository*. https://github.com/jovanbulck/sgx-step. Accessed on 15.07.2023.

[15] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018. DOI: 10.1145/3243734.3243822. URL: https://doi.org/10.1145/3243734.3243822.

[16] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. 2017. DOI: 10.1145/3152701.3152706. URL: https://doi.org/10.1145/3152701.3152706.

[17] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. "Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*. 2017. DOI: 10.1145/3052973.3053007. URL: https://doi.org/10.1145/3052973.3053007.

[18] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. "AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves". In: *32nd USENIX Security Symposium, USENIX Security*

2023, Anaheim, CA, USA, August 9-11, 2023. 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/constable.

[19] Bernstein Daniel J. *Cache-timing attacks on AES*. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 2005.

[20] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. "Secure Encrypted Virtualization is Unsecure". In: *CoRR* abs/1712.05090 (2017). arXiv: 1712.05090. URL: http://arxiv.org/abs/1712.05090.

[21] Marcus Hähnel, Weidong Cui, and Marcus Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel.

[22] Felicitas Hetzelt and Robert Buhren. "Security Analysis of Encrypted Virtual Machines". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*. 2017. DOI: 10.1145/3050748.3050763. URL: https://doi.org/10.1145/3050748.3050763.

[23] "IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices". In: *IEEE Std 1619-2007* (2008), pp. 1–40. DOI: 10.1109/IEEESTD.2008.4493450.

[24] Intel. "Asynchronous Enclave Exit Notify and the EDECCSSA User Leaf Function". In: *White Paper* (2022-06). ID 736463.

[25] Intel. "Intel Architecture Instruction Set Extensions and Future Features". In: *Manual* (2022-09). 319433-046.

[26] Intel. *Intel Trust Domain Extensions*. https://cdrdv2.intel.com/v1/dl/getContent/690419. Accessed on 19.09.2024. 2023-02.

[27] Intel. *Intel Trust Domain Extensions (Intel TDX) Module Base Architecture Specification*. Revision 348549-004US. 2024-03.

[28] David Kaplan. *Protecting VM Register state with SEV-ES*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf. 2017-02.

[29] David Kaplan, Jeremy Powell, and Wolle. *AMD Memory Encryption*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf. 2021-10.

[30] Fan Lang, Wei Wang, Lingjia Meng, Jingqiang Lin, Qiongxiao Wang, and Linli Lu. "MoLE: Mitigation of Side-channel Attacks against SGX via Dynamic Data Location Escape". In: *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*. 2022. DOI: 10.1145/3564625.3568002. URL: https://doi.org/10.1145/3564625.3568002.

[31] David Lantz, Felipe Boeira, and Mikael Asplund. "Towards Self-monitoring Enclaves: Side-Channel Detection Using Performance Counters". In: *Secure IT Systems - 27th Nordic Conference, NordSec 2022, Reykjavic, Iceland, November 30-December 2, 2022, Proceedings*. 2022. DOI: 10.1007/978-3-031-22295-5_7. URL: https://doi.org/10.1007/978-3-031-22295-5%5C_7.

[32] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho.

[33] 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: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. 2022. DOI: 10.1109/SP46214.2022.9833768. URL: https://doi.org/10.1109/SP46214.2022.9833768.

[34] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. "CrossLine: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3485253. URL: https://doi.org/10.1145/3460120.3485253.

[35] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. "Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan.

[36] 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: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan.

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

[38] Moritz Lipp, Daniel Gruss, and Michael Schwarz. "AMD Prefetch Attacks through Power and Time". In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/lipp.

[39] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices". In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[40] Moritz Lipp, Vedad Hadzic, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. "Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors". In: *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*. 2020. DOI: 10.1145/3320269.3384746. URL: https://doi.org/10.1145/3320269.3384746.

[41] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015. DOI: 10.1109/SP.2015.43. URL: https://doi.org/10.1109/SP.2015.43.

[42] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. 2017. DOI: 10.1007/978-3-319-66787-4_4. URL: https://doi.org/10.1007/978-3-319-66787-4%5C_4.

[43] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat.

[44] Mathias Morbitzer, Manuel Huber, and Julian Horsch. "Extracting Secrets from Encrypted Virtual Machines". In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX , USA, March 25-27, 2019*. 2019. DOI: 10.1145/3292006.3300022. URL: https://doi.org/10.1145/3292006.3300022.

[45] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. "SEVered: Subverting AMD's Virtual Machine Encryption". In: *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*. 2018. DOI: 10.1145/3193111.3193112. URL: https://doi.org/10.1145/3193111.3193112.

[46] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. "SEVerity: Code Injection Attacks against Encrypted Virtual Machines". In: *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*. 2021. DOI: 10.1109/SPW53761.2021.00063. URL: https://doi.org/10.1109/SPW53761.2021.00063.

[47] Stephan Mueller and Marek Vasut. *Linux Kernel Crypto API*. https://www.kernel.org/doc/html/latest/crypto/index.html. Accessed on 10.07.2023.

[48] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00057. URL: https://doi.org/10.1109/SP40000.2020.00057.

[49] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. 2018. URL: https://www.usenix.org/conference/atc18/presentation/oleksenko.

[50] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*. 2006. DOI: 10.1007/11605805_1. URL: https://doi.org/10.1007/11605805%5C_1.

[51] Colin Percival. *Cache missing for fun and profit*. https://papers.freebsd.org/2005/cperciva-cache_missing.files/cperciva-cache_missing-paper.pdf. 2005.

[52] Martin Radev and Mathias Morbitzer. "Exploiting Interfaces of Secure Encrypted Virtual Machines". In: *Reversing and Offensive-oriented Trends Symposium (ROOTS)*. 2020.

[53] Phillip Rogaway. "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC". In: *Proceedings of the 10th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. 2004.

[54] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019. DOI: 10.1145/3319535.3354252. URL: https://doi.org/10.1145/3319535.3354252.

[55] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/.

[56] The Qubes Security Team. *Qubes Security Bulletin #19*. https://github.com/QubesOS/qubes-secpack/blob/master/QSBs/qsb-019-2015.txt. Accessed on 10.07.2023. 2015-07.

[57] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. "PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 20th International Conference, DIMVA 2023, Hamburg, Germany, July 12-14, 2023, Proceedings*. 2023. DOI: 10.1007/978-3-031-35504-2_3. URL: https://doi.org/10.1007/978-3-031-35504-2%5C_3.

[58] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. "The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*. 2019. DOI: 10.1145/3321705.3329820. URL: https://doi.org/10.1145/3321705.3329820.

[59] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. "Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. 2022. DOI: 10.1145/3548606.3560654. URL: https://doi.org/10.1145/3548606.3560654.

[60] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. "SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00080. URL: https://doi.org/10.1109/SP40000.2020.00080.

[61] Luca Wilke, Jan Wichelmann, Florian Sieck, and Thomas Eisenbarth. "undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation". In: *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*. 2021. DOI: 10.1109/SPW53761.2021.00064. URL: https://doi.org/10.1109/SPW53761.2021.00064.

[62] Rafael J. Wysocki. *CPU Performance Scaling*. https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html. Accessed on 10.07.2023. 2017.

[63] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

8

TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX

Publication

Luca Wilke*, Florian Sieck* and Thomas Eisenbarth. *TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX*. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

Contribution

Florian and I are co-first authors. The broad ideas for the two attack primitives presented in the paper were developed in collaboration. The implementation of the two attack primitives as well as the more fine-grained tweaks to make them work were mainly done by me. I also performed all the experiments for gathering the traces for the attack case studies. Florian Sieck performed the cryptanalysis presented in Section 6 and the key recovery part of the attack case studies.

Outline

TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX

Luca Wilke*, Florian Sieck* and Thomas Eisenbarth

University of Lübeck

Trusted Execution Environments are a promising solution for solving the data privacy and trust issues introduced by cloud computing. As a result, all major CPU vendors integrated Trusted Execution Environments (TEEs) into their CPUs. The biggest threat to TEE security are side-channel attacks, of which single-stepping attacks turned out to be the most powerful ones. Enabled by the TEE attacker model, single-stepping attacks allow the attacker to execute the TEE one instruction at a time, enabling numerous controlled- and side-channel based security issues. Intel recently launched Intel TDX, its second generation TEE, which protects whole virtual machines (VMs). To minimize the attack surface to side-channels, TDX comes with a dedicated single-stepping attack countermeasure.

In this paper, we systematically analyze the single-stepping countermeasure of Intel TDX and show, for the first time, that both, the built-in detection heuristic as well as the prevention mechanism, can be circumvented. We reliably single-step TDX-protected VMs by deluding the TDX security monitor about the elapsed processing time used as part of the detection heuristic. Moreover, our study reveals a design flaw in the single-stepping countermeasure that turns the prevention mechanism against itself: An inherent side-channel within the prevention mechanism leaks the number of instructions executed by the TDX-protected VM, enabling a novel attack we refer to as *StumbleStepping*. Both attacks, single-stepping and *StumbleStepping*, work on the most recent Intel TDX enabled Xeon Scalable CPUs.

Using *StumbleStepping*, we demonstrate a novel end-to-end attack against wolfSSL's ECDSA implementation, exploiting a control flow side-channel in its truncation-based nonce generation algorithm. We provide a systematic study of nonce-truncation implementations, revealing similar leakages in OpenSSL, which we exploit with our single-stepping primitive. Finally, we propose design changes to TDX to mitigate our attacks.

1 Introduction

Data privacy concerns and legal regulations still hinder processing of sensitive data in the cloud. Such outsourced computation requires implicit trust in the cloud service

provider, that has full control over the machines that make up the cloud, and thus over the processed data. Trusted Execution Environments (TEE) are thriving due to the promise of protecting computations and data even from privileged adversaries with full control over the systems, e.g. over the hypervisor software. Effectively, TEEs lock out the cloud service provider and enable verifiably protected data processing on remote machines. Early designs like Intel SGX focused on protecting single processes and required the developer to adjust their application to the TEE. While the introduction of the library OS approach [6, 7, 49] partially solved this problem, a newer and more scalable approach is taken by the newest generation of TEEs, namely Intel Trust Domain Extensions (TDX) [21], AMD SEV [3, 25, 26], ARM Confidential Compute Architecture [5], as well as IBM Secure Execution [16]. This newest TEE generation protects entire virtual machines (VMs) and can thus be used as a drop-in solution to protect existing applications in the cloud with only minimal adjustments.

While removing the hypervisor from the trust base promises a wide range of use cases, it also comes with challenges and has severe implications for security. In fact, it introduces an attacker model in which the attacker has full system control. Thus, the adversary has a broad arsenal of mechanisms to gleam information from the protected code running inside the TEE. Since their introduction, TEEs have been extensively scrutinized by security researchers, revealing that microarchitectural side-channels remain an Achilles heel of current TEE designs: Soon after the release of SGX it was shown that control over the page table allows the hypervisor or OS to learn detailed information about the control flow of TEE-protected code [58]. Subsequent work demonstrated that numerous microarchitectural features such as caches [14, 32, 50] or branch prediction [13, 27] provide an even more fine-grained resolution of secret-dependent control flows or data accesses.

Interrupt driven single-stepping [11, 56], a powerful attack technique against Intel SGX and AMD SEV, greatly increases the temporal and spatial resolution of side-channel attacks on SGX [10, 31, 41, 45, 46, 47, 48] and on AMD SEV [43, 44, 51, 56, 59]. As such it poses a particularly severe threat for the security of TEEs. One especially powerful attack that builds on single-stepping is *instruction counting*. There, the attacker combines page fault information with single-stepping to reveal the target's control flow with intra-page precision, allowing them to exploit even the smallest secret-dependent control flow deviations [11, 32, 33].

Thus, a good defense mechanism against single-stepping attacks is an important building block for securing TEEs. Given the long line of attacks on SGX, Intel recently published AEX-Notify [12] in collaboration with academic researchers. AEX-Notify introduces a hardware-software co-design that makes the enclave interrupt-aware and allows it to prevent single-stepping attacks by providing a special interrupt handler.

To ensure resistance against similar attacks on Intel TDX, Intel early on conducted several security reviews for TDX [1, 19]. As part of this effort, TDX provides a built-in countermeasure against single-stepping attacks. In contrast to the AEX-Notify approach, the TDX single-stepping countermeasure does not depend on the software inside the TEE. Instead, the countermeasure is implemented inside the TDX module, TDX's security monitor. The countermeasure consists of a detection heuristic and a special single-stepping prevention mode that gets activated by the heuristic. We present the first systematic investigation of Intel's single-stepping countermeasure and show two attacks that overcome different aspects of the countermeasure.

The first attack on the Intel TDX single-stepping countermeasure exploits a weakness in its detection heuristic to prevent the activation of the single-stepping prevention mode. The heuristic is partially based on the elapsed time between entering and exiting the protected VM, which is very small if the VM is single-stepped. We manipulate the TDX module's sense of time, causing it to observe normal execution times, although the VM is single-stepped. While this vulnerability should be mitigatable by updating the detection logic, defining a safe and sound rule set is not trivial.

The second attack, *StumbleStepping*, exploits the inherent side-channel attack surface of the TDX single-stepping prevention. The intended effect of the single-stepping prevention mode is to stop the hypervisor from obtaining fine-grained insights into the protected VM's progress. *StumbleStepping*, however, exploits the prevention mode's inherent cache side-channel to leak the number of instructions executed by the protected VM. As such, *StumbleStepping* exploits a systematic issue that is not easily fixable, meanwhile providing a somewhat weaker leakage than single-stepping.

To demonstrate the capabilities of *StumbleStepping* in exploiting small leakages, we target a minuscule leakage found in the current ECDSA implementations of wolfSSL. The leakage was already identified in [52], but was deemed unexploitable by the authors. We show that this leakage can actually be captured by *StumbleStepping* and is exploitable for select choices of elliptic curves. We provide an extensive analysis of the ECDSA leakage, revealing similar problems in OpenSSL, which we exploit with our single-stepping primitive.

In summary, we:

- Demonstrate an attack that renders the TDX single-stepping countermeasure inoperative and enables single-stepping on TDX
- Introduce the *StumbleStepping* attack, an inherent cache side-channel in the single-stepping countermeasure of TDX that leaks the number of instructions executed by the TD

- Use *StumbleStepping* and our single-stepping primitive, to leak ECDSA keys in a novel nonce truncation-based attack against wolfSSL and OpenSSL
- Provide an extensive analysis of nonce truncation leakages in ECDSA implementations including wolfSSL and OpenSSL

The code to reproduce our results is available at `https://github.com/UzL-ITS/tdxdown`.

The remainder of this paper is structured as follows: Section 2 introduces required background information. Next, Section 3 analyzes the TDX single-stepping countermeasure in detail. Section 4 and Section 5 introduce and evaluate the two main attack primitives of this paper. Section 6 analyzes nonce truncation-based control flow leakages in ECDSA implementations. Afterwards, in Section 7 we present two attack case studies, exploiting wolfSSL's ECDSA leakage using *StumbleStepping* and OpenSSL's ECDSA leakage via our single-stepping primitive. Finally, we discuss limitations and countermeasures in Section 8.

Responsible Disclosure We officially reported our findings to Intel's PSIRT team on October 11, 2023. Using our proof of concept code they reproduced our attacks and issued CVE 2024-27457. Intel is working on a countermeasure against the single-stepping attack and states that future TDX module versions after 1.5.0.6 should no longer be affected. Intel will not provide countermeasures against instruction counting attacks like *StumbleStepping* as part of the TDX module and instead refers to their Software Security Guidance information [24] to solve this issue on the application level.

We contacted wolfSSL and OpenSSL with our findings concerning leaking nonce bits in their ECDSA implementations. The findings were acknowledged for both libraries. At the time of submission, wolfSSL fixed the vulnerability in version 5.6.6 and assigned CVE 2024-1544. OpenSSL does not assign CVEs for "same physical system side-channel" [40] vulnerabilities but acknowledged it and is working on a fix.

2 Background

2.1 TDX

Intel *Trust Domain Extensions (TDX)* [21] is a Trusted Execution Environment (TEE) that protects whole VMs. The protected VMs are called *Trust Domains (TD)*. Figure 1 shows an overview of the TDX architecture. The so-called *TDX module* forms the core of the design. In contrast to regular VMs, the hypervisor needs to invoke the TDX module's *SEAMCALL API* to manage TDs. Crucially, only the TDX module can enter TDs. Exits

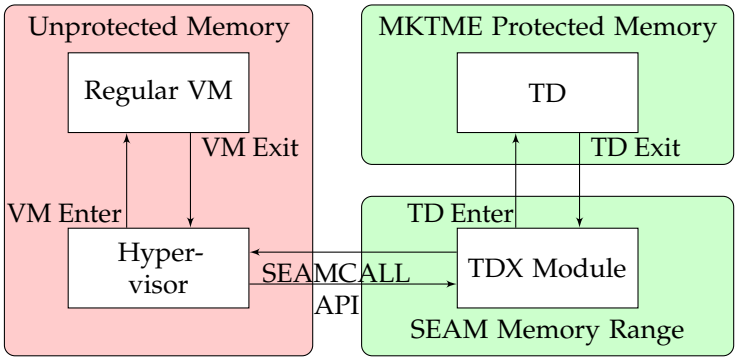


Figure 1: Unlike with regular VMs, the hypervisor does not have direct access to TDs but has to manage them via the TDX module. Based on Figure 5.2 from [21].

from a TD return control to the TDX module instead of to the hypervisor. Thus, the TDX module forms a trusted layer between the untrusted hypervisor and the TD.

To protect the TDX module, it resides in a newly-added, protected memory range. Furthermore, TDX introduces a new CPU mode called *Secure Arbitration Mode (SEAM)* that is split into two sub modes *VMX root* and *VMX non-root*. The TDX module runs in the SEAM VMX root mode, while TDs run in the SEAM VMX non-root mode. To protect against physical attackers the memory used by the TDs is encrypted using *Multi Key Total Memory Encryption (MKTME)* [18]. MKTME allows the use of different encryption keys based on the *KeyID*, an identifier that is encoded by re-purposing the upper bits of the physical address. With TDX, the KeyID range is split into shared and private. Using private KeyIDs is restricted to the new SEAM CPU mode and thus to the TDX module and TDs. In addition, an access right-based mechanism is used for additional security when the CPU is outside the SEAM mode. Reading protected memory returns a fixed pattern to guard against ciphertext side-channel attacks [28, 30]. Writing taints the memory location, leading to a fatal error the next time the TD tries to access it.

2.2 TDX Control Data Structures

The data structures describing a TD are managed by the TDX module [22, Sec. 6]. One such control data structure is the *Trust Domain Virtual Processor State (TDVPS)*, which describes the state of each virtual CPU of a given TD. The memory for the data structure is initially allocated by the hypervisor and then passed to the TDX module, which encrypts the memory with a private MKTME KeyID. Thus, while knowing the memory addresses of the TDVPS, the hypervisor is forced to use the TDX module's API to interact with the content of the data structure.

2.3 Cache Attacks on Intel TDX

As explained in Section 2.1, TDX encrypts the TD's memory with MKTME which has a severe impact on the applicable cache side-channels. Since the MKTME KeyID is encoded in the physical address bits, it is part of the cache tag. As a result, accessing the same data with different KeyIDs would, in theory, lead to different cache tags and thus in different decryptions of the same physical data residing in the cache at the same time. However, a coherency mechanism ensures that an existing entry using a different KeyID is flushed prior to loading the data with the new KeyID. This behavior enables *Flush+Reload* style cache attacks where the attacker accesses TD memory with a different KeyID, to evict it from the cache [1, 23]. Without this mechanism, an attacker could not perform *Flush+Reload* attacks, as they can neither perform flushes nor memory accesses with the TD's KeyID.

2.4 Instruction Counting Attacks

Instruction counting is a single-stepping-based side-channel attack against TEEs that aims to infer fine-grained control flow information. The attacker is assumed to know the executed binary. Then, they combine the coarse grained page fault controlled-channel [58] with a single-stepping attack, to reconstruct the victim's control flow with instruction granularity. Instruction counting attacks can even detect if two code branches of equal length execute memory accesses at different points in the instruction sequence [11, 33]. To guard against such attacks, security critical code, should employ the data oblivious constant-time paradigm: Neither the execution path nor any memory accesses may depend on secret data. While hard to implement, these properties defeat all known side-channel attacks.

2.5 Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a variant of DSA on elliptic curves [38]. In order to sign a message, one chooses an elliptic curve $E(\mathbb{F}_p)$ over a finite field $\mathbb{F}_p$ and a generator G of order n. Next, the signer generates a long-term secret key $d \in [1, n-1]$ and the corresponding public key $Q = d \cdot G$, which is a point on the elliptic curve. Finally, for every signature, the signer generates a new unique nonce $k \in [1, n-1]$ which they must keep secret. The signature is then computed over the hash h of the message and comprises the tuple (r, s). The signer calculates $r = x_r \bmod n$ with $(x_r, y_r) = k \cdot G$ and $s \equiv k^{-1}(h + r \cdot d) \bmod n$.

The security of the long-term key d in ECDSA depends heavily on the choice of the nonce k. A slight bias in the randomness of k allows to recover the secret key. While

the sampling algorithm used by most ECDSA implementations produces sufficiently random numbers, an attacker can also obtain such a bias via side-channel information [4, 42, 52]. To reconstruct the secret key d from signatures for which the attacker has partial information about k, there are two common approaches. Both first recover k and then use it to compute d. The first approach formulates the problem as a shortest vector problem (SVP) and solves it via lattice reduction techniques like LLL or BKZ [15, 39]. Recently, Albrecht et al. [2] improved the performance of the lattice reduction approach by combining it with enumeration and sieving with predicate, allowing to exploit smaller biases while also requiring fewer biased signatures. The second approach by Bleichenbacher [8] is based on Fourier analysis.

2.6 Attacker Model

We assume the default TEE attacker model, where an attacker with root privileges on the system tries to attack a program running inside the TEE [11, 29, 35, 55]. Most importantly for this work, the attacker is capable of using the page fault controlled-channel, programming the APIC timer and controlling the processor frequency. For our experiments, we disabled all hardware cache prefetchers, Intel *SpeedStep*, as well as hardware controlled P-states. The latter is important to allow the Linux *cpufreq* driver to control the CPU frequency. We do not assume physical access.

This is in line with the Intel TDX attacker model [21]. A real world example is a malicious or compromised cloud service provider that tries to leak data from a customer's TD.

3 TDX Single-Stepping Countermeasure

Single-stepping is a powerful attack mechanism that has been successfully deployed against major TEEs like SGX [11, 31, 33, 41, 45, 46] and AMD SEV [43, 44, 51, 56, 59]. It uses the attacker's control over the APIC timer to interrupt the TEE after every instruction. Besides enhancing existing side-channel attacks, it can also be used for so-called instruction counting attacks, to reveal the TEE's control flow at the instruction level, allowing the exploitation of minuscule leakages in e.g. cryptographic libraries. Given the devastating effects of single-stepping attacks, Intel TDX comes with a built-in countermeasure which consists of a heuristic to detect single-stepping interrupt patterns and subsequently activates a prevention mode. In the remainder of this section we explain both mechanisms in detail. An overview of the single-stepping countermeasure is given in Figure 2.

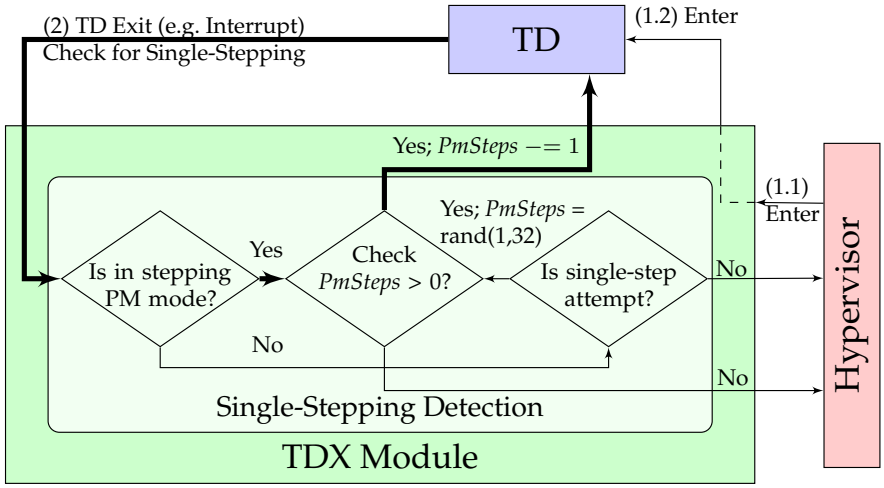


Figure 2: The hypervisor starts the TD by calling the TDX module (1.1) which in turn enters the TD (1.2). When the TD exits (2), the TDX module applies a heuristic to check for single-stepping. If yes, it activates the prevention mode (PM) and re-enters the TD for a randomized number of times (*PmSteps*), before disabling the PM mode, as indicated by the circle in the top left part. On each entry, a special configuration is used to ensure that the TD only executes a single instruction.

TDX Interrupt Architecture On both, Intel SGX and AMD SEV, single-stepping attacks exploit the fact that external interrupts, like the APIC timer interrupt, abort the execution of the TEE and return control to the attacker controlled OS or hypervisor. Intel TDX, however, has a different design. Neither can the attacker controlled hypervisor directly enter the TD nor do exits from the TD immediately return control to the hypervisor, as discussed in Section 2.1. Instead, there is a trusted intermediate layer, called the TDX module, which runs in a special CPU mode and offers so-called *SEAMCALLs* to the hypervisor for managing TDs. While this design still allows a malicious hypervisor to program the APIC timer such that it interrupts the TD shortly after it is entered, the resulting exit is now handled by the TDX module, as shown in Figure 2. Thus, the malicious hypervisor cannot immediately observe if the TD was interrupted. Since the TDX module is not intended to replace the hypervisor, there are many instances in which the TDX module eventually needs to notify the hypervisor about the TD interruption to allow the expected virtualization behavior. To prevent interrupt-based single-stepping attacks, the TDX module makes an attempt on deciding whether a certain interrupt pattern is benign or if it is part of an attack. In the former case, the TDX module immediately returns to the hypervisor, while in the case of a detected attack it continues to execute the TD for a randomized amount of cycles via the single-stepping prevention mode, before eventually handing back control to the hypervisor.

Single-Stepping Detection Whenever the TD is exited due to an interrupt, the TDX module measures and evaluates two properties to decide if the current interrupt behavior is benign or if the hypervisor tries to perform a single-stepping attack. In Figure 2 both checks are summarized as "*Is single-step attempt?*".

The first analyzed property is the number of bytes the TD's instruction pointer (RIP) has advanced since the last exit. If RIP has advanced by more than two times the number of bytes required for the longest x86 instruction, the TDX module can be sure that at least two instructions have been executed, i.e., the TD has not been single-stepped [20].

The second analyzed property is the time that has elapsed since the TD was entered via the TDX module. To obtain the elapsed time, the TDX module stores a *rdtsc* timestamp t_B before entering and a timestamp t_A after exiting the TD. If "sufficient" time $t_d = t_A - t_B$ has passed, the current interrupt behavior is classified as benign. In version 1.5 of the TDX module the threshold is set to 4096 *rdtsc* increments [20]. If either at least two instructions have been executed or sufficient time has passed between entries, the behavior is classified as normal. Otherwise, the TDX module activates the single-stepping *prevention mode*.

Prevention Mode The core idea of the prevention mode is to execute a randomized number of instructions *PmSteps* inside the TD before finally informing the hypervisor about the initial interrupt. To implement this, after detecting a potential single-stepping attempt, the TDX module first disables all interrupts to block the hypervisor from further interfering with the execution of the prevention mode. Since the TDX module runs in the privileged SEAM VMX root mode, the prevented interrupts even include NMIs and SMIs [36, Sec. 1.3.4]. Next, the TDX module enters the TD with the Monitor Trap Flag, causing it to exit after executing exactly one instruction [17, Sec 26.5.2, Vol 3C]. This process is repeated in a loop until *PmSteps* instructions have been executed, as depicted by the circle in the top left of Figure 4.

In essence, the single-stepping prevention mode single-steps the TD *PmSteps* times from the TDX module while preventing the hypervisor from architecturally observing or interrupting the TD. When the control is eventually returned to the hypervisor, it is unaware of the TD's exact progress.

4 Single-Stepping Trust Domains

In this section, we demonstrate how to circumvent the single-stepping detection heuristic from the previous section, re-enabling single-stepping attacks on TDX. In essence, we delude the TDX module about the elapsed time between entering and exiting the TD by

reducing the frequency of the CPU core running the TD. Thereby, we exploit that the *rdtsc* timestamp counter's frequency is unaffected by CPU frequency scaling.

4.1 Attack Primitive Description

The TDX module classifies a TD interruption as benign if sufficient time has passed since entering the TD, as explained in Section 3. More precisely, the time span t_d between entering and exiting the TD is measured inside the TDX module by comparing the *rdtsc* value t_B shortly before entering and t_A shortly after exiting the TD. The goal of the attacker is to trick the TDX module into measuring $t_d \geq 4096$ while only executing one instruction in the TD.

On modern Intel CPUs the *rdtsc* timestamp counter is incremented at a constant frequency instead of being tied to the current CPU frequency [17, Sec 18.17, Vol 3B]. We combine this *rdtsc* behavior with the malicious hypervisor's ability to configure the current operating frequency of the core on which the TDX module and the TD are running: Using a constant frequency for *rdtsc* implies that the demanded 4096 timestamp counter increments always take the same wall clock time. Meanwhile, lowering the frequency of the victim core slows down its execution speed. By setting the frequency low enough, entering the TD, executing one instruction and leaving the TD already takes more than 4096 timestamp counter increments. Consequently, with the modified CPU frequency in place, we are able to use the APIC timer to interrupt the TD after every instruction while still ensuring that the TDX module measures $t_d \geq 4096$ and does not trigger the single-stepping prevention.

Reliable Single-Stepping To use the APIC timer for single-stepping, we need to ensure that the corresponding interrupt hits during the execution of the first instruction. As described in [11, 12, 56], we flush the TD's Translation Lookaside Buffer (TLB) to prolong the execution time of the first instruction and therefore increase the timing window that leads to single-stepping. By choosing the timer such that it arrives rather at the start of the single-stepping window than at the end, we prevent multi-stepping.

While this causes occasional zero-steps, we can detect them by running a cache attack against the code page currently executed by the TD. We use the KeyID-based *Flush+Reload* mechanism from [1, 23], which results in a very strong signal with access timing differences higher than DRAM reads. The hypervisor flushes all cache lines corresponding to the code page before entering the TD and reloads all lines after exiting the TD. A long reload time signals the execution of an instruction within the TD. In the next paragraph, we describe how we obtain the address of the code page.

Finally, similar to SEV-Step [56], we have to modify the hypervisor to suppress virtual APIC timer interrupts while single-stepping the TD. Otherwise, the TD would jump to the corresponding interrupt handler, instead of executing the targeted code.

Adding Spatial Information For a meaningful interpretation of the single-stepping data, we need to correlate it with the currently executing code in the TD. To achieve this, we use the well known page fault side-channel [29, 33, 35, 37, 45, 55], that leaks both code and data accesses with page granularity, in order to detect the currently executing application via template attacks. In contrast to other TEEs like SGX or SEV, the page tables for the TD's private memory are inaccessible to the attacker, since they are managed by the TDX module. As a result, we cannot modify the access permission bits to force page faults. However, the TDX module still offers a dedicated API that allows the hypervisor to temporarily block access to any TD page, albeit without the ability to only remove individual access permissions from the page.

Zero-Stepping While we try to minimize zero-stepping for instruction counting attacks, another line of research has shown that it can be used to boost microarchitectural side-channels by allowing to repeatedly measure the effect of the same instruction [48]. For this work, we consider further exploration of zero-stepping attack primitives on TDX out of scope.

4.2 Attack Primitive Evaluation

In this section, we evaluate our primitive for single-stepping TDX with a synthetic target. The experiments were performed on a 5th generation Intel Xeon Gold 6526Y with a base frequency of 2800 MHz. The processor runs the TDX module in version 1.5. We ran the evaluation in a default Ubuntu 23.10 environment and we implemented the code of the attack primitive on top of the Ubuntu Linux kernel in version 6.5 with the official TDX patches. To break the *rdtsc* based time check in the single-stepping heuristic, we configure the CPU frequency of the core running the TD and the TDX module to the lowest possible value of 800 MHz.

To validate that our single-stepping primitive works reliably, we verify that we do not multi-step and that we dependably detect zero-steps. Therefore, we run the loop from Listing 8.1 in the TD and measure 3 different scenarios: Executing the loop once (8 instructions), nine times (56 instructions) and ten times (62 instructions). We repeat the measurement for every scenario 10,000 times. The code for the single-stepping evaluation purposely contains `NOP`s as these are the shortest instructions and do not load any parameters from memory. For the evaluation we assume that the address of the code

Listing 8.1: Evaluation target for single-stepping.

```
  mov  qword ptr[r8], 42
loop_label:
  dec rax
  nop
  nop
  nop
  nop
  jnz loop_label
  mov qword ptr [r9], 42
```

Listing 8.2: Evaluation target for *StumbleStepping*.

```
  mov  qword ptr[r8], 42
loop_label:
  dec rax
  jnz loop_label
  mov qword ptr [r9], 42
```

Figure 3: Synthetic programs for evaluating the stepping primitives from Section 4 and Section 5. We block access to the memory locations pointed to by *r8* and *r9*, using the resulting page faults as the start and end trigger for the attack.

page of the target program as well as the addresses pointed to by *r8* and *r9* are known to the attacker.

We evaluate the attack in the release TDX mode as well as in debug mode. In debug mode, the TDX module offers additional API calls, e.g. reading the current instruction counter. Thereby, we can immediately check that no multi-steps occur and that all zero-steps are detected by the cache attack. In release mode, we check that after filtering zero-steps, the remaining event count matches the expected number of instructions. Again, we do not encounter any errors. We did not observe meaningful differences between the two modes with regard to single-stepping. When running the three evaluation scenarios we execute 1,260,000 instructions and observe only 0.8% zero-step events.

5 StumbleStepping: Leaking Instruction Counts

In this section, we describe a second attack primitive, that we dub *StumbleStepping*, allowing an attacker to perform instruction counting attacks against the TD. As discussed in Section 2.4, instruction counting attacks are commonly used to exploit secret-dependent control flow in, e.g., cryptographic libraries. *StumbleStepping* works, even if the single-stepping attack from the previous section is mitigated, i.e., the TDX module correctly activates the single-stepping prevention mode. In contrast to our single-stepping attack, it exploits a conceptual weakness of the countermeasure instead of a flawed checking logic. In essence, *StumbleStepping* turns the prevention mode upon itself and employs a cache attack to leak the number of instructions executed by the TD. An overview of the attack procedure is shown in Figure 4.

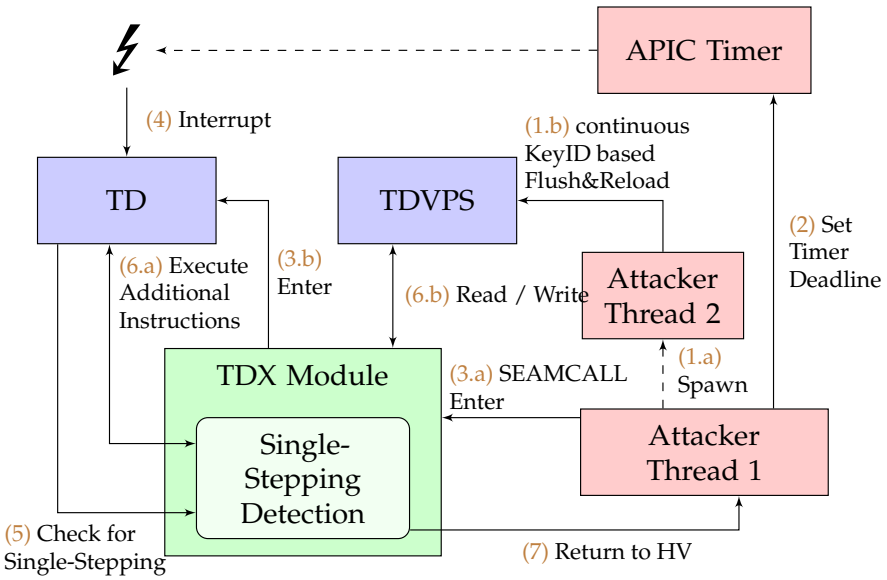


Figure 4: Overview of the steps required for APIC timer-based *StumbleStepping* attacks against TDX. The *Attacker* spawns a new thread to concurrently probe the pages with the victim's TDVPS using the KeyID-based *Flush+Reload* attack. The double ended arrow in step (6.a) represents the prevention mode (c.f. Section 3) and means that the TDX module re-enters the TD several times, before eventually resuming with step (7).

5.1 Attack Primitive Description

The core idea of *StumbleStepping* is to employ a side-channel attack (1.b) against the single-stepping prevention mode implemented inside the TDX module. The side-channel attack runs in parallel to the execution of the TDX module, on a separate core (1.a). In contrast to the single-stepping primitive, we do not want to avoid detection but deliberately trigger the countermeasure. After detecting a potentially malicious interrupt pattern (4 and 5), the single-stepping prevention mode of the TDX module re-enters the TD several times (6.a), before returning control to the hypervisor (7). The TDX module configures the TD such that on each entry the TD executes only a single instruction (6.a). For *StumbleStepping*, we exploit that the TDX module leaks the number of times it re-enters the TD via a cache side-channel (1.b). For each TD entry (6.a), the TD's TDVPS pages are accessed (6.b). This data structure describes the state of the TD, e.g., the vCPU's register file. Note that the information stored in the TDVPS pages is encrypted and thus inaccessible to the hypervisor. However, by running a cache attack in parallel to the execution of the countermeasure inside the TDX module, the hypervisor can leak the number of accesses to the TDVPS pages and thus the number of instructions executed by the TD. To continuously leak the number of executed instructions, the hypervisor sends APIC timer interrupts such that the countermeasure mode is always active (4). For the

cache attack, we again use the KeyID-based *Flush+Reload* mechanism.

Improving Temporal Resolution To obtain reliable information, we require a high temporal resolution for our cache attack, such that we do not miss any of the TDX module's accesses to the *TDVPS* structure. Thus, we only monitor a single cache line of the *TDVPS*. In addition, we again decrease the CPU frequency of the core running the TDX module while setting the frequency of the attacker's core to the highest possible value, increasing the effective sampling rate of our attack.

However, in order to ensure that we do not accidentally trigger the single-stepping attack from the previous section, circumventing the activation of the single-stepping prevention mode, we cannot clock down the TD's core to the lowest possible value of 800 MHz. Instead, we have to keep it running above 1.6 GHz. Thus, once the single-stepping detection heuristic has been fixed, the temporal resolution could be doubled by using the lowest frequency.

Adding Spatial Information As with the single-stepping primitive in Section 4, we use the page fault controlled-channel to correlate the information from *StumbleStepping* with the currently executing code page for a meaningful interpretation. The randomized bursts in which *StumbleStepping* executes the TD prevent the attacker from terminating the attack at a arbitrary instruction. However, we can exploit that the TDX module aborts the single-stepping prevention mode upon page faults to precisely stop the execution at a defined code location.

In summary, combined with page fault information, *StumbleStepping* reveals the TD's control flow with intra-page resolution, allowing to exploit minuscule secret-dependent control flow leakages. In contrast to single-stepping, it does not allow to pause the execution after every instruction.

5.2 Attack Primitive Evaluation

For evaluating the *StumbleStepping* primitive, we performed all experiments remotely on an Intel provided machine with a TDX enabled 4th generation Xeon Platinum 8480CTDX processor. Furthermore, we verified that the attack primitive still works on a 5th generation Intel Xeon Gold 6526Y which introduces public availability of Intel TDX. The 4th generation CPU used the TDX module software version 1.0 and the 5th generation CPU used version 1.5. For the 5th generation Intel Xeon processor, we ran the evaluation in a default Ubuntu 23.10 environment and implemented the code of the attack primitive on top of the Ubuntu Linux kernel in version 6.5 with the official TDX patches. The evaluation on the 4th generation CPU was conducted on Ubuntu 22.04 with kernel version 5.19. We evaluate *StumbleStepping* with a synthetic target.

Profiling TDVPS Accesses For *StumbleStepping*, we exploit that each TD entry leads to accesses to the TDVPS data structure which we want to observe via a cache attack in parallel to the execution of the TDX module. We again use the KeyID-based *Flush+Reload* mechanism and measure between 600 and 1000 cycles when accessing a cache line that has previously been accessed by the TDX module, which is much higher than the DRAM access time caused by a regular cache attack. To maximize the temporal resolution, we only observe one cache line of the TDVPS structure. We observe, that the number of observed cache misses per TD entry varies depending on the monitored offset inside the TDVPS pages. In an offline profiling step, we determine the offset with the lowest amount of noise, by running *StumbleStepping* against a calibration target several times, sweeping over every cache line aligned offset. On our machine, offset *0x128* in the third TDVPS page gives a stable correlation, with two accesses per TD entry.

Accuracy To evaluate the accuracy of our counting primitive, we use the synthetic code snippet from Listing 8.2. We choose a loop with only one instruction instead of an if-else construct as it allows us to easily scale the number of executed instructions while still allowing differences as small as two executed instructions between two runs. For the evaluation, we assume that the memory locations pointed to by *r8* and *r9* are known to the attacker.

Figure 5 shows the resulting data for 1 to 5 loop iterations which corresponds to 4 to 12 executed instructions. Figure 6 shows the data for 100 to 105 loop repetitions which corresponds to 202 to 212 instructions. The results clearly show that the measurement noise increases when we observe longer program sequences. However, the distributions for different iteration counts only partially overlap and the means are easily distinguishable. Thus, when using only a single measurement, there is a certain error probability when trying to distinguish events with almost the same amount of executed instructions. However, repeating the measurement multiple times eliminates the error. For events with larger instruction differences, a single measurement is sufficient.

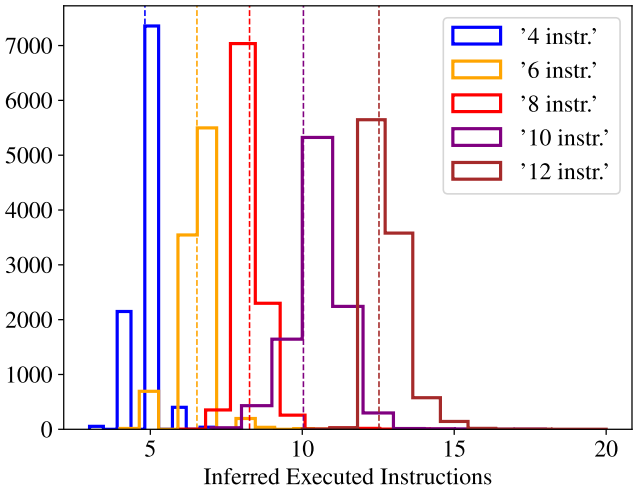


Figure 5: Inferred instruction count for 1 to 5 repetitions of the loop from Listing 8.2 repeating using 10,000 measurements. The dotted lines show the mean value.

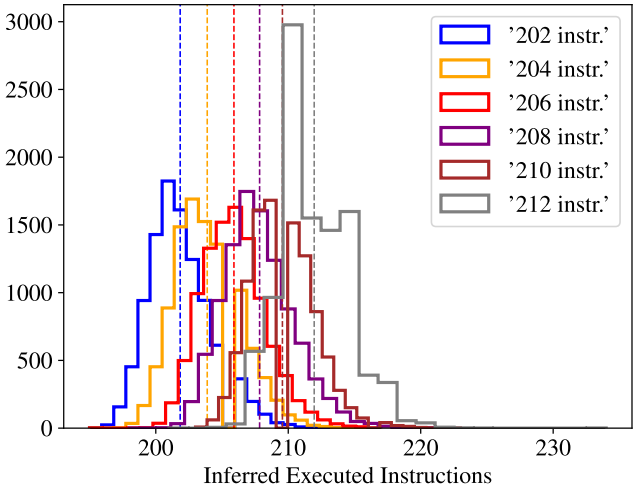


Figure 6: Inferred instruction count for 100 to 105 repetitions of the loop from Listing 8.2 using 10,000 measurements. The dotted lines show the mean value. We removed a total of 17 outliers above 300 inferred instructions.

6 Leaking ECDSA Keys from Biased Nonce Truncation

As discussed in the preceding sections, single-stepping attacks are frequently used to leak secret-dependent control flow. To protect against such attacks without relying on countermeasures employed by the TEE, cryptographic libraries should use the data oblivious constant-time programming paradigm. However, developing constant-time code at the instruction level is a challenging task. In this section we present, in detail, a control flow-based leakage during the derivation of the random and secret nonce k of the ECDSA signing process.

In essence, there are two established ways to generate a random nonce mod n: A modular reduction-based truncation of the randomly generated value or rejection sampling of random values until a value $k < n$ is drawn. Implementations of the latter method usually do not leak a nonce bias. However, implementations of the former are more prone to leak information, as they require a division which is more complex to implement in a side-channel resistant manner. Both methods are listed in the FIPS digital signature standard [38, Sec. A.3.1, A.3.2].

While Weiser et al. [52] already discussed this leakage in modular reduction-based truncation, they deemed it negligible and did not further investigate it. We analyze this leakage in full detail and show that, depending on the curve, in fact up to 15 bits of the nonce are leaked. Additionally, we systematically investigate the usage of truncation for nonce computation in multiple cryptographic libraries, finding leakages in wolfSSL and OpenSSL. We evaluate the introduced leakage and its exploitability depending on the curve and the curve's modulus.

Root Cause To ensure that the nonce k is smaller than the curve's modulus n, both wolfSSL and OpenSSL use truncation via modular reduction of a random byte string. The byte string has a bit length greater than the bit length of the curve order n. Next, both libraries perform a modular reduction, reducing the random byte string to a value smaller than n. Therefore, on a high level, both libraries consider the two top most words of the numerator k_{top} and the top most word of the denominator n_{top}. Next, they compute $q_{top} = \frac{k_{top}}{n_{top}}$ to estimate the quotient $q = \frac{k}{n}$. Afterwards, they check whether $n \cdot q_{top} > k$. If this condition is true, q_{top} is decremented. This decrementation is implemented as a loop, meaning the number of times q_{top} has to be decremented is reflected in the execution count of the loop. The number of loop iterations in turn can be observed by a side-channel attacker and leaks information about the nonce's most significant bits.

Investigated Libraries and Curves Table 1 lists all libraries we investigated during this work and whether they use truncation or rejection sampling. Of the analyzed libraries, only wolfSSL and OpenSSL use truncation. We initially found the leakage by analyzing

Table 1: ECDSA nonce generation in different libraries.

Library	Version	Nonce Derivation	Vuln.	c'time version inc.
wolfSSL	5.6.4	truncation	yes	limited[1]
OpenSSL	3.2.0	truncation	yes	no[2]
Nettle	3.9.1	rejection sampling	no	N/A[3]
Mbed TLS	3.5.1	rejection sampling	no	N/A[3]
botan	3.2.0	rejection sampling	no	N/A[3]
nss	3.9.4	rejection sampling	no	N/A[3]

[1] Only for curves secp256,384,521; not enabled by default
[2] Constant-time variant not yet implemented
[3] Not applicable; the default is rejection sampling

wolfSSL with Microwalk [53, 54]. Using the obtained knowledge, we were able to analyze the remaining libraries manually.

The remainder of this section is structured as follows. First, we give details on our analysis methodology. Next, we present the discovered leakages in wolfSSL and OpenSSL in more detail and discuss their exploitability.

6.1 Analysis Methodology

Before giving the results on the individual implementations in the analyzed libraries, we describe our analysis workflow.

Simulated Side-Channel Traces In order to calculate the maximum obtainable information and plot the bias introduced to the nonce k, we simulate side-channel traces by adding counters to the targeted code, to observe the occurrence of certain control flow events. We give more details on these events in the next sections. For each curve and library we collect 10 million signatures. Per signature, we store the values of the injected event counters, the signed hash h, the ECDSA signature values r and s, as well as the nonce k. For the latter, we again modify the libraries as the nonce is not usually exported. We stress that these modifications *do not* introduce secret-dependent changes to the control flow and thus do not influence the code's leakage properties. Afterwards, we divide the collected samples into sets, one set per observable control flow event combination. Within each set, we analyze the distribution of the bit values of k. We refer to a distribution of nonce bit values simply as distribution.

Table 2: Maximum obtainable leakage in terms of mutual information (MI) and fully leaked bits (FB) for different curves in wolfSSL and OpenSSL. The MI values are rounded. The full bit (FB) value reports on those bits which have the same value for all nonces in a distribution. The event column specifies the event combination which corresponds to the leakage and the probability of the event combination.

	wolfSSL		OpenSSL	
Curve	Event	MI / FB	Event	MI / FB
	(W_1, W_2) $Pr[A = a]$	[bit / bit]	(O_1, O_2) $Pr[A = a]$	[bit / bit]
bp224r1	(2, *) 0.09	1.6 / 1	(1, 0) $1.6 \cdot 10^{-4}$	7 / 6
bp320r1	(3, *) < 0.002	3 / 3	(2, *) $1.7 \cdot 10^{-3}$	3 / 3
bp384r1	(2, *) 0.05	3.5 / 0	(1, *) 0.05	3.5 / 0
secp160r1	(2, *) $1.5 \cdot 10^{-5}$	15.6 / 15	(1, *) $1.3 \cdot 10^{-5}$	15.8 / 15

Leakage Quantification To quantify the leakage, we calculate I, the mutual information (MI) per distribution. Therefore, we use $I = \sum_{i=0}^{i<\text{bitlen}(G)} H(B) - H(B|A = a)$ with B being the random variable describing a single bit value over the alphabet $\{0, 1\}$, A the random variable describing the distribution of nonces, and G the generator of the curve. The number of distributions per curve depends on the number of discernible events. The probability $Pr[A = a]$ of a nonce falling into one of the distributions is calculated by dividing the number of samples with a specific event combination by the total number of signatures collected for the curve.

Since we subdivide all nonces recorded during sampling into disjoint sets, we are interested in the overall information gain on all nonce bits per distribution rather than the gain over all distributions. Thus, we do not sum over all distributions when calculating the MI but only consider the distribution of the considered event combination.

While the MI precisely captures the leakage from an information theoretic point of view, most key reconstruction algorithms require knowing the value of individual bits with high certainty. Thus, we also analyzed which bits of each nonce always have the same value for a given event combination. We call these *full bits*. Comparing MI and *full bits* gives an insight on how much of the MI is distributed over small biases in different bits. All leaking bits are most significant bits (MSB). We analyzed the curves secp128r1, secp160r1 and secp192r1 as well as the R1 Brainpool curves for 160, 192, 224, 256, 320 and 384 bits for wolfSSL and OpenSSL.

Listing 8.3: Simplified version of the leaking _sp_div_impl (wolfssl/wolfcrypt/src/sp_int.c) function which divides a by d and is called during the nonce generation. The snippet is not self-contained and only intended to highlight the control flow. The colored *Event* comments mark points in the control flow that leak information about the nonce.

```
1   int _sp_div_impl(sp_int* a, d, r, trial) {
2
3    for (i = a->used - 1; i >= d->used; i--) {
4     //Calculate trial quotient
5     t = sp_div_word(a->dp[i], a->dp[i-1], dt);
6     do {
7       for (j = 0; j < d->used; j++) {...}
8       for (j = d->used; j > 0; j--)
9           //Event W2
10          if (trial->dp[j] != a->dp[j + o])
11              break;
12
13      if (trial->dp[j] > a->dp[j + o]) { t--; }
14      //Event W1
15    }while (trial->dp[j] > a->dp[j + o]);
16    }
17  };
```

The most important findings are summarized in Table 2 and the results for the remaining curves and control flow events can be found in Table 3 in Section A. Per curve and library, we specify the control flow events which cause leakage in the event column. In the next two sections, we describe the leakages and the corresponding events in more detail.

6.2 Nonce Leakage in wolfSSL

For analyzing the leakage in wolfSSL, we use the default compile configuration with additional hardening parameters and options to enable smaller ECC curves as well as Brainpool curves. While the default implementation of wolfSSL's math functionality is supposedly constant-time [57], the default ECC sign functionality makes use of truncation for generating k. With additional, non-default compiler flags, wolfSSL includes implementations which use rejection sampling, but these are only available for the curves secp256, secp384 and secp521.

Leaking Control Flow Events A simplified version of the algorithm for nonce truncation in wolfSSL is shown in Listing 8.3. Event W_1 in line 14 describes the number of times the do-while loop was executed and thus how often the estimated quotient was decremented. As a shorthand, we use $W_1 = x$ if W_1 occurred x times. The event W_2 counts which words of the estimated quotient and nominator are relevant for the comparison to decide on the decrementation of the variable `t`.

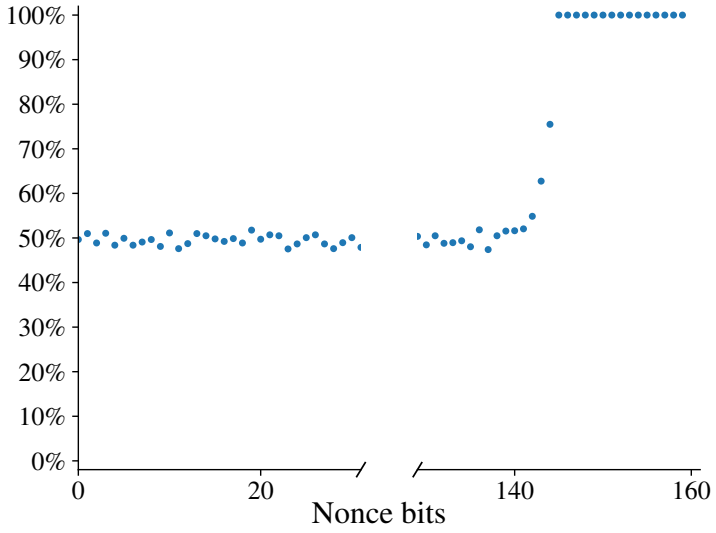


Figure 7: Distribution of the nonce bits for curve secp160r1 in wolfSSL given event $(W_1 = 2, W_2 = *)$. The y-axis shows the percentage of nonces for which the value of the corresponding bit is 1. The 15 most significant bits are always 1.

Leakage Quantification

Figure 7 shows the bias introduced to the nonce when $W_1 = 2$, i.e. there are two iterations of the do-while loop. While this event only happens with a probability of approximately $1.5 \cdot 10^{-5}$, it reveals that 15 MSBs of the nonce are 1. Note, the bit length of the curve order of secp160r1 is 161 bit, contrary to what the name suggests. However, the order's MSBs are all 0, except for the most significant bit. Thus, the order of secp160r1 is only slightly larger than 2^{160}, meaning the likelihood of a k with 161 bits is very small. Within the 10 million samples we collected, most (about 50%) have a k of size 160 bit, but there was no sample with a k of size 161 bit. Consequently, we assume the most significant bit of k to be 0 and known by default.

The curve brainpoolp320r1 and brainpoolp384r1 show a leakage of 3 bit and 3.5 bit. The distributions are shown in Figure 10 in the Appendix. Though the brainpoolp384r1 curve does not leak any bit without error, i.e. no *full bits* (c.f. Section 6.1), there is less than 2% error in each of the biases of the 4 top most significant bits. For the brainpoolp224r1 curve, wolfSSL only shows negligible leakage.

Leakage Exploitability The leakage observed for secp160r1 is exploitable with conventional LLL reduction techniques. In Section 7.3 we demonstrate the key reconstruction for secp160r1 from real side-channel traces as a case study for *StumbleStepping*.

For evaluating the exploitability of the leakages for the curves brainpoolp320r1 and brainpoolp384r1, we use the predicate with sieving technique from Albrecht et al. [2] and

extend their implementation to also support MSB prefixes containing bits other than 0, as the leaked MSB prefixes are `0b110` and `0b1000`, respectively. The work of Albrecht et al. suggests, that 4 bits are required to reconstruct keys for 384 bit curves. Thus, our 3.5 bit leakage in the brainpoolp384r1 curve is a borderline case. However, our data shows that we can also use the 4 MSBs as *full bits*, accepting a small additional error. The error can be countered by resampling the subsets of the obtained signatures used for reconstruction and running the key reconstruction multiples times with different subsets.

While the key reconstruction terminates in a reasonable time, it never succeeds. To verify, that the error is not due to the small error probability of the individual bits, we also performed additional key reconstruction experiments on simulated data: We simulate the leading 4 bit `0b1000` leakage from our side-channel experiments without errors. However, the reconstruction does not succeed either. Using a simulated, error free 5 bit leakage, the reconstructions succeeds. To verify the correctness of our changes to the implementation of Albrecht et al. [2] we validate that the reconstruction for a simulated 4 bit non-zero MSB leakage for the NISTP384 curve, which they used as a benchmark, succeeds. Since this validation was successful, we assume that more than 4 bits are required for brainpoolp384r1, in contrast to NISTP384.

The 3 bit leakage for the brainpoolp320r1 was too small to be exploited with the methods of Albrecht et al. in our experiments.

6.3 Nonce Leakage in OpenSSL

For analyzing OpenSSL, we compile it with the default configuration, which uses truncation with modular reduction to compute the nonce k. From the OpenSSL code and corresponding comments, we could infer that it is envisaged to implement the computation of the estimated quotient in constant-time. However, this feature is not used and during the course of the responsible disclosure we learned that it is not implemented.

Leaking Control Flow Events We show a simplified version of the procedure used for the division during nonce truncation in Listing 8.4. It is comparable to the procedure used in wolfSSL, however, contains slightly different observable side-channel events. Event O_1 in line 9 describes how often the estimated quotient is decremented. Additionally, we observe the event O_2 in line 14 which describes whether the remainder of the division overflows during the procedure of decrementing the variable q. We do not consider the if-clause following event O_2. Though it changes the control flow we did not observe any differences in the resulting bit distributions when using it as a differentiator.

Leakage Quantification In Figure 8, we show that there is a 7 bit leakage for curve brainpoolp224r1. As detailed in Table 2, only 6 of these 7 bits are *full bit* leakages. However,

Listing 8.4: Simplified version of the leaking bn_div_fixed_top (openssl/crypto/bn/bn_div.c) function which divides *num* by *divisor* and is called during the nonce generation. The snippet is not self-contained and only intended to highlight the control flow. The colored *Event* comments mark points in the control flow that leak information about the nonce.

```
1  int bn_div_fixed_top(BIGNUM* dv, rm, num, divisor,
2    BN_CTX *ctx) {
3
4  for (i = 0; i < loop; i++, wnumtop--) {
5    for (;;) {
6      if ((t2h < rem) ||
7          ((t2h == rem) && (t2l <= n2)))
8        break;
9      //Event O1
10     q--;
11     rem += d0;
12     if (rem < d0) //don't let rem overflow
13       break;
14     //Event O2
15     if (t2l < d1)
16       t2h--;
17     t2l -= d1;
18   }}};
```

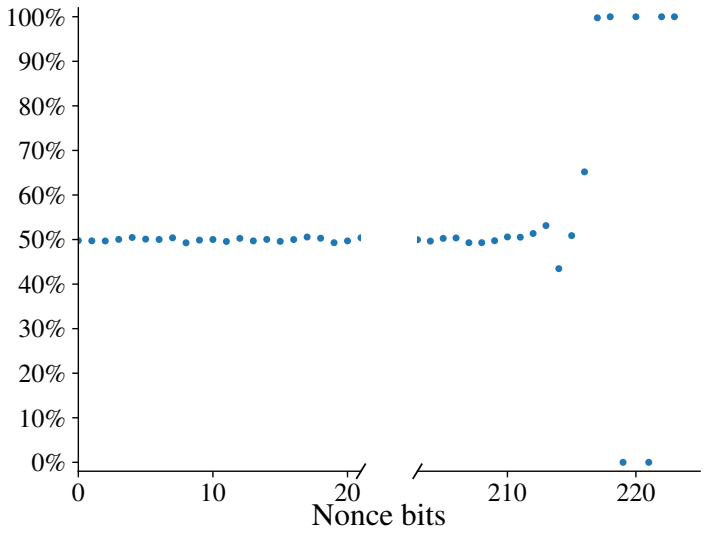


Figure 8: Distribution of nonce bits for brainpoolp224r1 curve in OpenSSL given event $(O_1 = 1, O_2 = 0)$. The y-axis shows the percentage of nonces for which the value of the corresponding bit is 1. The 6 most significant bits are 0b110101 for all samples. The 7th bit is 1 for more than 99% of all samples.

since the error for the 7th bit is small, we can integrate it into the key reconstruction as well. This leakage is only observable in the OpenSSL implementation as a differentiation by the event O_2 is required. For the secp160r1, brainpoolp320r1 and brainpoolp384r1 curve, OpenSSL shows similar leakage as wolfSSL.

Leakage Exploitability The 7 bit leakage for curve brainpoolp224r1 is exploitable. To reconstruct the key, we use our extended variant of the predicate with sieving technique from Albrecht et al. [2]. We present the result in our single-stepping case study in Section 7.2. The maximum leakage and exploitability for the curves secp160r1, brainpoolp320r1 and brainpoolp384r1 corresponds to the analysis in Section 6.2.

6.4 Leakage Analysis Summary

We observe that the truncation of the secret nonce k leaks a varying number of most significant bits, depending on the order of the ECDSA curve. The order of the curve serves as denominator during nonce truncation. While we observe only small leakages for curves with an order that consist of only 1 valued bits in the MSBs, we see large leakage in the opposite case, i.e., few 1 valued bits in the MSBs of the curve's order.

In contrast to what is reported in previous work [52], we find that the bias introduced through modular reduction during nonce creation is not always negligible, but depends on the order n of the curve. We show that in certain situations, a substantial bias is introduced and observable through side-channels. Additionally, note that FIPS 186.5 [38] states in A3.1 that implementations which use truncation during nonce creation shall use an additional 64 bit of randomness to avoid a bias to k. While wolfSSL is following this advice, k is still biased. We assume that the advice in FIPS 186.5 refers to the overall distribution of k, but does not take into account additional side-channel information.

7 Case Studies

In the following we evaluate both our attack primitives on real-world cryptographic libraries and demonstrate their ability to leak the ECDSA nonce k, allowing us to reconstruct the private key. Our attack targets are the nonce leakages described in the previous section. We first explain our attack approach in general and then give details for the specific primitives and attacks targets.

7.1 Attack Approach

The general attack approach is the same for both attacks and splits into an online and an offline phase.

Offline Phase In the offline phase, we build the mapping from the number of observed instructions per trace to the occurrence count of the events. Additionally, to be able to use single-stepping and *StumbleStepping* we need to find page fault trigger points, such that the number of instructions executed between the trigger points allows us to infer the number of times the control flow passes the observed event. To infer when the victim is about to be executed, we generate a page fault based template. For all tasks, we use a semi-automated approach combining static binary analysis and dynamic binary instrumentation.

Online Phase In the online phase, the attacker first needs to determine the guest physical addresses of certain functions inside the target in order to instantiate the page fault sequence template from the offline phase. Afterwards, they can use the template to start single-stepping or *StumbleStepping* for tracing the TD when the targeted code is about to be executed. This allows us to count the executed instructions between the trigger points. For the evaluation, we streamlined the attack scenario by calling the target libraries from a self-written program, that triggers the signature generation and supplies the attacker with the guest physical address of the target library. As several works [28, 29, 30, 35] against AMD's confidential VM solution SEV, as well as confidential VM like systems in general [9], have already demonstrated that an attacker can locate applications in memory by observing access patterns, it is a valid assumption that the attacker can infer the guest physical addresses for the page fault template. We want to stress that the addresses used for the template are only from the target library, not from the calling application. To maximize the performance, we implemented our attack logic inside the Linux KVM hypervisor kernel module.

7.2 Single-Stepping brainpoolp224r1

The first case study shows the reliability and high resolution of our single-stepping primitive. We extract the private ECDSA key from the side-channel leakage in the nonce generation process for the brainpoolp224r1 curve in OpenSSL as described in Section 6. The attack was executed on the same platform as the single-stepping evaluation. The possible event combinations for brainpoolp224r1 in OpenSSL are (O_1, O_2): $(0, *)$, $(1, 0)$, $(1, 1)$. The event $(1, 0)$ corresponds to signatures with the nonce bias required for our attack. This event corresponds to leaving the inner for-loop in Listing 8.4 before event $O_2 = 0$ in Line 14 but only after executing event $O_1 = 1$ in Line 9 once.

Offline Phase Due to the code structure, we cannot use page accesses to distinguish the events. Instead, we use page accesses shortly before and after the loop to trigger single-stepping. Using our trigger points, we measure 32 steps for the event $(0, *)$, either 38 for 39 steps for the nonce bias event $(1, 0)$, and 42 or 43 steps for the event $(1, 1)$. The variable amount of steps for the events $(1, 0)$ and $(1, 1)$ is caused by the *or*-condition in Listing 8.4 before the event O_1 and some code restructuring by the compiler.

Online Phase As explained in the attack approach, we first obtain the guest physical address for the attacked code sequence in OpenSSL to instantiate the page fault template which we use to single-step only the execution of the nonce truncation. Our attack code requires on average 32.98 ms per signature. Without an ongoing attack, a signature requires on average 0.33 ms. To recover the key, we need 33 signatures with the nonce bias event (O_1, O_2) = $(1, 0)$, i.e. 38 or 39 counted steps. Given the low probability of the event, we need to observe 170,000 signatures. Collecting all signature traces takes approximately 94 minutes. The reconstruction of the long-term key is conducted as described in Section 6.3 and requires 1.5 seconds on an Intel Xeon E-2286M.

7.3 StumbleStepping secp160r1

In this section, we exploit the nonce leakage in the secp160r1 curve from Section 6 with *StumbleStepping*. We choose wolfSSL as target for the attack and run the experiments on the same platform used for the evaluation of the *StumbleStepping* primitive.

The possible event combinations for secp160r1 are (W_1, W_2): $(1, 1)$, $(2, 1)$, $(2, 2)$. The event W_1 describes the number of times the do-while loop in Listing 8.3 is executed. The events $(2, 1)$, $(2, 2)$ correspond to signatures with the nonce bias required for our attack.

Offline Phase Due to the code structure, we cannot use page accesses to distinguish the events. Instead, we use page accesses shortly before and after the outer loop to trigger *StumbleStepping*. Using our trigger points, the events $(1, 1)$, $(2, 1)$, $(2, 2)$ correspond to 178, 230 and 239 executed instructions.

Online Phase As explained in the attack approach, we first determine the required guest physical address for the attacked code sequence in wolfSSL, instantiate the page fault template and then start the *StumbleStepping* primitive to trace only the execution of the nonce truncation. Our attack code requires on average 4.77 ms per secp160r1 signature. Without an ongoing attack, a signature requires on average about 0.01 ms.

To recover the key, we require 12 signatures with a biased nonce, i.e. two occurrences of the event W_1. Since the probability for two occurrences of event W_1 is very low, we need to observe 700,000 signatures. Collecting all signature traces takes approximately

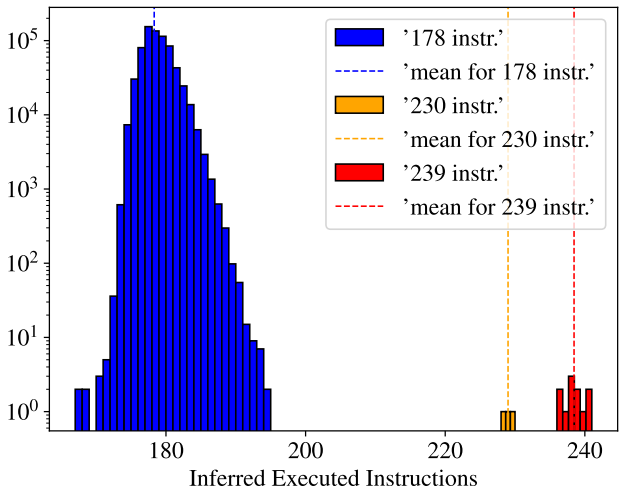


Figure 9: Side-channel data for the *StumbleStepping* attack on the secp160r1 curve in wolfSSL. The legend states the actual number of executed instructions while the x-axis shows the inferred number of instructions. In total, we collected data for 700,000 signatures.

56 minutes. The measurement results are shown in Figure 9. As expected from the evaluation of the toy examples in Section 5.2, the measurements contain some noise. Still, we are able to distinguish the two relevant event groups, which differ by 52 instructions, without errors. Using the LLL approach described in Section 2.5, the key recovery finishes in 1.7 seconds on an Intel Xeon E-2286M.

8 Discussion

In this section we suggest improvements to the current single-stepping countermeasure design and discuss limitations of our *StumbleStepping* primitive.

8.1 Improved Single-Stepping Detection

We propose to only rely on the progress of the instruction counter to detect single-stepping. As discussed in AEX-Notify [12], single-stepping requires the attacker to artificially slow down the execution of the first instruction, e.g. via a TLB flush. The authors further show that consequently the attacker cannot reliably interrupt the SGX enclave after the second instruction, i.e. the attacker cannot "two-step". Based on these results, changing the heuristic to enforce that at least two instructions have been executed

prevents repeatedly interrupting the TD after x instructions. If less than two instructions have been executed the prevention mode gets activated.

8.2 Improved Single-Step Prevention Mode

In this section, we propose changes to the single-stepping prevention mode, to mitigate the instruction count leakage. For our *StumbleStepping* attack in Section 5, we observe memory accesses to the TD's TDVPS management data structure to infer the number of executed instructions. Thus, one could consider adding additional accesses to this data structure from the TDX module to introduce noise to any potential side-channel measurements relying on these accesses. However, the fact that each iteration of the invocation of the TDX single-stepping prevention mode requires a TD entry and exit exposes a large microarchitectural attack surface, potentially allowing for other measurable effects. For example, simply measuring the time between the APIC timer interrupt firing and control being handed back to the hypervisor already reveals coarse grained information about the number of instructions executed by the TD.

Thus, as a more profound solution, we propose to extend the Monitor Trap Flag mechanism currently used when executing the TD in single-stepping prevention mode. Instead of trapping after one instruction, the mechanism could directly support to execute a randomized number of instructions in one burst. As a result, only a single TD entry is required regardless of the number of instructions executed by the single-stepping prevention mode, mitigating the instruction count leakage at its root.

8.3 AEX-Notify based Countermeasure

Orthogonal to the TDX single-stepping countermeasure, that is split into detection and prevention, it should also be possible to port the countermeasure from AEX-Notify[12] to VM-based TEEs like TDX. They execute a special interrupt handler that prefetches the first instruction to undo any artificial slowdowns that would be required for single-stepping. Since VMs are already interrupt aware, it should be possible to simply execute this interrupt handler every time the TD is entered. With the original AEX-Notify design, the security of the TEE depends on the runtime inside the protected enclave to use their interrupt handler. With TDX, this could be improved by moving the prefetching step to the TDX module, instead.

8.4 Limitations of StumbleStepping

Compared to instruction counting attacks that use single-stepping, as e.g. CopyCat [33], the *StumbleStepping* attack from Section 5 provides slightly weaker leakage. Since we cannot pause the TD after every instruction, we cannot distinguish balanced if-else branches that only differ in the relative order of their memory accesses. This is only possible with single-stepping, as it allows removing page access rights after every memory access instruction.

8.5 Attack Overhead

The case studies in Section 7.2 and Section 7.3 show different relative overheads introduced to the signature computation time by the attack code. While the single-stepping attack on OpenSSL in Section 7.2 slows down the execution approximately by a factor 100, the signature creation with a running StumbleStepping attack on wolfSSL in Section 7.3 is roughly 500 times slower.

These differences can be attributed to multiple factors. First, we use different single-stepping mechanisms in both primitives. While StumbleStepping implicitly single-steps the TD by setting the MTF flag, the single-stepping primitive uses the APIC timer. Furthermore, the observed instruction sequences have different lengths and finally, the page fault sequences required to trigger the attack have different lengths.

9 Related Work

We start this section, by reviewing existing security flaws found in TDX before giving a summary of existing attacks on ECDSA.

9.1 TDX

To the best of our knowledge, this is the first academic paper attacking the Intel TDX single-stepping countermeasure. However, Intel commissioned several security reviews [1, 19] to assess and improve the security of TDX.

Single-Stepping In Intel's security review [19], a straightforward single-stepping attack against an early TDX version was developed. However, Intel states that this is mitigated since TDX module version 1.0. Our evaluation targets use version 1.0 and version 1.5 and thus break Intel's countermeasure. During the disclosure process, Intel stated TDX module versions after 1.5.0.6 will contain additional security measures.

Page Fault Controlled-Channel The authors of [1] discuss that the memory blocking feature of the TDX API can be used to implement page fault controlled-channel attacks, recovering the TD's control flow with page granularity.

Cache Attacks In [1] they also describe how the MKTME KeyID in combination with the cache coherency protocol enables *Flush+Reload*-style cache attacks on TDX. In addition, they state that the *monitor* and *mwait* instruction can be used to implement cache attacks, similar to [60].

9.2 ECDSA Key Recovery

Weiser et al. [52] perform a systematic study of ECDSA nonce leakages. They already discovered the leakage described in this work, but classify it as negligible and do not further investigate it. However, our results show that the leakage depends on the curve order and that it introduces large biases for some curves. CopyCat [33] uses instruction counting with SGX to exploit side-channels in modular inversion and elliptic curve scalar multiplication. In TPM-Fail [34], the authors also exploit the elliptic curve scalar multiplication, however in the context of TPM implementations. In Ladderleak [4], the authors use a timing side-channel in combination with roughly half a billion signatures for a 163 bit curve to exploit nonce leakages smaller than 1 bit, building on Bleichenbacher [8]. Moreover, Ryan [42] investigates leakages introduced through non-constant-time implementations of the modular reduction of $r \cdot d$ and $r \cdot d + h$.

10 Conclusion

Intel's most recent TDX TEE comes with a built-in countermeasure against single-stepping attacks. In this work, we have demonstrated the first attacks against this countermeasure. We developed two attack primitives: Single-stepping TDs by outwitting the detection heuristic and counting the TD's instructions with *StumbleStepping*. The former fully breaks the countermeasure by manipulating the CPU frequency to pass a time check in the single-stepping detection heuristic. The second attack exploits the side-channel properties of the single-stepping countermeasure, revealing a systematic flaw in the current design that leaks the number of executed instructions via a cache side-channel. We propose design changes to mitigate both attacks. As a second major contribution, we have performed an extensive analysis of nonce truncation-based leakages in ECDSA signatures, revealing vulnerable implementations in wolfSSL and OpenSSL. We exploit our findings in two attack case studies: one against curve secp160r1

in wolfSSL using *StumbleStepping* and one against curve brainpoolp224r1 in OpenSSL using our single-stepping primitive.

Acknowledgements

The authors thank Intel for providing them with access to a TDX enabled machine. Additionally, we thank Anja Rabich for fruitful discussions as well as Anna Pätschke and Jan Wichelmann for proofreading and valuable feedback. This work was supported by the BMBF projects SASVI and AnoMed as well as by Deutsche Forschungsgemeinschaft (DFG) under grant 439797619 (HaSPro).

References

[1] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. *Intel Trust Domain Extensions (TDX) Security Review*. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf. Accessed on 07.10.2023. 2023-04.

[2] Martin R. Albrecht and Nadia Heninger. "On Bounded Distance Decoding with Predicate: Breaking the "Lattice Barrier" for the Hidden Number Problem". In: *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*. 2021. DOI: 10.1007/978-3-030-77870-5_19. URL: https://doi.org/10.1007/978-3-030-77870-5%5C_19.

[3] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. 2020-01.

[4] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. "LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage". In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. 2020. DOI: 10.1145/3372297.3417268. URL: https://doi.org/10.1145/3372297.3417268.

[5] ARM. *Introducing Arm Confidential Compute Architecture*. https://developer.arm.com/documentation/den0125/latest. Revision 0300-01. 2023-06.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. "SCONE: Secure Linux Containers with Intel SGX". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[7] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *ACM Trans. Comput. Syst.* 33.3 (2015), 8:1–8:26. DOI: 10.1145/2799647. URL: https://doi.org/10.1145/2799647.

[8] Daniel Bleichenbacher. "On the generation of one-time keys in DL signature schemes". In: *Presentation at IEEE P1363 working group meeting*. 2000.

[9] Robert Buhren, Felicitas Hetzelt, and Niklas Pirnay. "On the Detectability of Control Flow Using Memory Access Patterns". In: *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. Toronto, Canada, 2018. ISBN: 9781450359986. DOI: 10.1145/3268935.3268941. URL: https://doi.org/10.1145/3268935.3268941.

[10] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018. DOI: 10.1145/3243734.3243822. URL: https://doi.org/10.1145/3243734.3243822.

[11] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. 2017. DOI: 10.1145/3152701.3152706. URL: https://doi.org/10.1145/3152701.3152706.

[12] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. "AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves". In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/constable.

[13] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. "Branchscope: A new side-channel attack on directional branch predictor". In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 693–707.

[14] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. "Cache Attacks on Intel SGX". In: *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*. 2017. DOI: 10.1145/3065913.3065915. URL: https://doi.org/10.1145/3065913.3065915.

[15] Nick Howgrave-Graham and Nigel P. Smart. "Lattice Attacks on Digital Signature Schemes". In: *Des. Codes Cryptogr.* 23.3 (2001), pp. 283–290.

[16] IBM. *Introducing IBM Secure Execution for Linux 1.3.0*. https://www.ibm.com/docs/en/linuxonibm/pdf/l130se03.pdf. Revision SC34-7721-03. 2022-11.

[17] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1 to 4*. Revision 325462-080. 2023-06.

[18] Intel. *Intel Architecture Memory Encryption Technologies*. Revision 336907-004US. 2022-10.

[19] Intel. *Intel TDX Documentation*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/tdx-security-research-and-assurance.html. Accessed on 30.11.2023. 2023-04.

[20] Intel. *Intel TDX Module - Code for Single-Step Detection and Single-Step Prevention*. https://github.com/intel/tdx-module/blob/tdx_1.5/src/td_transitions/td_exit_stepping.. Accessed on 18.04.2024. 2024.

[21] Intel. *Intel Trust Domain Extensions*. https://cdrdv2.intel.com/v1/dl/getContent/690419. Accessed on 19.09.2024. 2023-02.

[22] Intel. *Intel Trust Domain Extensions (Intel TDX) Module Base Architecture Specification*. Revision 348549-001. 2021-09.

[23] Intel. *MKTME Side Channel Impact on Intel TDX*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/mktme-side-channel-impact-on-intel-tdx.html. Accessed on 07.10.2023. 2023.

[24] Intel. *Software Security Guidance - Best Practices*. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/best-practices.html. Accessed on 28.08.2024. 2024.

[25] David Kaplan. *Protecting VM Register state with SEV-ES*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf. 2017-02.

[26] David Kaplan, Jeremy Powell, and Wolle. *AMD Memory Encryption*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf. 2021-10.

[27] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho.

[28] 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: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. 2022. DOI: 10.1109/SP46214.2022.9833768. URL: https://doi.org/10.1109/SP46214.2022.9833768.

[29] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. "Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan.

[30] 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: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan.

[31] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. "PLATYPUS: Software-based Power Side-Channel Attacks on x86". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. 2021. DOI: 10.1109/SP40001.2021.00063. URL: https://doi.org/10.1109/SP40001.2021.00063.

[32] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. 2017. DOI: 10.1007/978-3-319-66787-4_4. URL: https://doi.org/10.1007/978-3-319-66787-4%5C_4.

[33] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-copycat.

[34] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. "TPM-FAIL: TPM meets Timing and Lattice Attacks". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-tpm.

[35] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. "SEVered: Subverting AMD's Virtual Machine Encryption". In: *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*. 2018. DOI: 10.1145/3193111.3193112. URL: https://doi.org/10.1145/3193111.3193112.

[36] Stephan Mueller and Marek Vasut. *Intel Trust Domain CPU Architectural Extensions*. https://www.kernel.org/doc/html/latest/crypto/index.html. Revision 343754-002. 2021-05.

[37] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00057. URL: https://doi.org/10.1109/SP40000.2020.00057.

[38] National Institute of Standards and Technology. *FIPS 186-5 - Digital Signature Standard (DSS)*. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf. 2023-02.

[39] Phong Q. Nguyen and Igor E. Shparlinski. "The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces". In: *Des. Codes Cryptogr.* 30.2 (2003), pp. 201–217. DOI: 10.1023/A:1025436905711. URL: https://doi.org/10.1023/A:1025436905711.

[40] OpenSSL. *OpenSSL Security Policy*. https://www.openssl.org/policies/general/security-policy.html. Accessed on 18.04.2024. 2024.

[41] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CrossTalk: Speculative Data Leaks Across Cores Are Real". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. 2021. DOI: 10.1109/SP40001.2021.00020. URL: https://doi.org/10.1109/SP40001.2021.00020.

[42] Keegan Ryan. "Return of the Hidden Number Problem. A Widespread and Novel Key Extraction Attack on ECDSA and DSA". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.1 (2019), pp. 146–168. DOI: 10.13154/TCHES.V2019.I1.146-168. URL: https://doi.org/10.13154/tches.v2019.i1.146-168.

[43] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. "WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP". In: *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. 2024. DOI: 10.1109/SP54263.2024.00262. URL: https://doi.org/10.1109/SP54263.2024.00262.

[44] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. "HECKLER: Breaking Confidential VMs with Malicious Interrupts". In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/schl%20%5C%C3%5C%BCter.

[45] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019. DOI: 10.1145/3319535.3354252. URL: https://doi.org/10.1145/3319535.3354252.

[46] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. "Util: : Lookup: Exploiting Key Decoding in Cryptographic Libraries". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3484783. URL: https://doi.org/10.1145/3460120.3484783.

[47] Florian Sieck, Zhiyuan Zhang, Sebastian Berndt, Chitchanok Chuengsatiansup, Thomas Eisenbarth, and Yuval Yarom. "TeeJam: Sub-Cache-Line Leakages Strike Back". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024.1 (2023-12), pp. 457–500. DOI: 10.46586/tches.v2024.i1.457-500. URL: https://tches.iacr.org/index.php/TCHES/article/view/11259.

[48] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. "MicroScope: Enabling Microarchitectural Replay Attacks". In: *IEEE Micro* 40.3 (2020), pp. 91–98. DOI: 10.1109/MM.2020.2986204. URL: https://doi.org/10.1109/MM.2020.2986204.

[49] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[50] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[51] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. "PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 20th International Conference, DIMVA 2023, Hamburg, Germany, July 12-14, 2023, Proceedings*. 2023. DOI: 10.1007/978-3-031-35504-2_3. URL: https://doi.org/10.1007/978-3-031-35504-2%5C_3.

[52] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. "Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/weiser.

[53] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "MicroWalk: A Framework for Finding Side Channels in Binaries". In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. 2018. DOI: 10.1145/3274694.3274741. URL: https://doi.org/10.1145/3274694.3274741.

[54] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. "Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. 2022. DOI: 10.1145/3548606.3560654. URL: https://doi.org/10.1145/3548606.3560654.

[55] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. "SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00080. URL: https://doi.org/10.1109/SP40000.2020.00080.

[56] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. "SEV-Step A Single-Stepping Framework for AMD-SEV". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 180–206. DOI: 10.46586/TCHES.V2024.I1.180-206. URL: https://doi.org/10.46586/tches.v2024.i1.180-206.

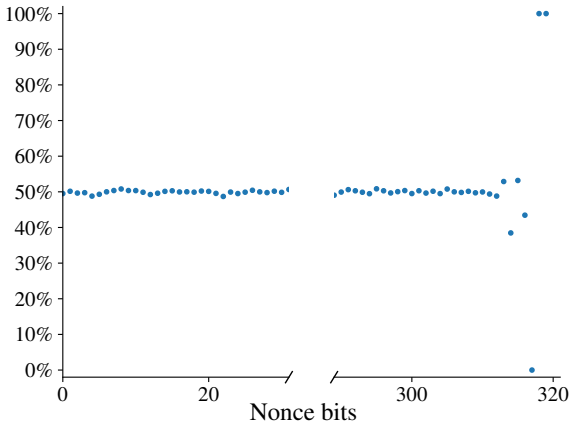
[57] wolfSSL. *wolfSSL Manual*. https://www.wolfssl.com/documentation/manuals/wolfssl/chapter02.html. Accessed on 30.11.2023. 2023.

[58] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015. DOI: 10.1109/SP.2015.45. URL: https://doi.org/10.1109/SP.2015.45.

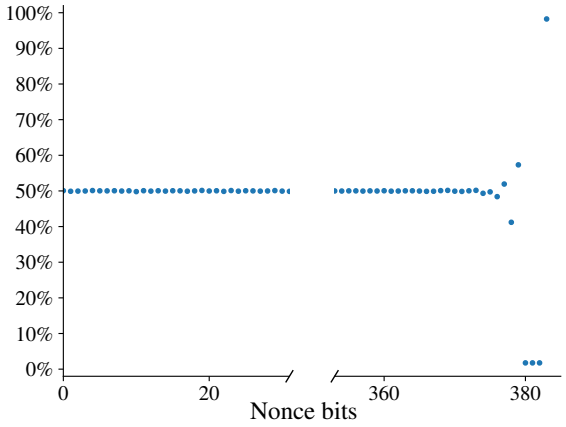
[59] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. "CacheWarp: Software-based Fault Injection using Selective State Reset". In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-ruiyi.

[60] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. "(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels". In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-ruiyi.

A ECDSA Leakage Analysis



(a) Distribution of the nonce bits for curve brainpoolp320r1 in wolfSSL given the event $W_1 = 3$. The most significant bits are 0b110 in all cases.



(b) Distribution of the nonce bits for curve brainpoolp384r1 in wolfSSL given the event $W_1 = 2$. The most significant bits are 0b1000 in 98% of all cases.

Figure 10: Distribution of nonce bits for different Brainpool curves. Events are specified in the subcaptions. The y-axis shows the percentage of nonces with a 1 in the corresponding bit position.

Table 3: Maximum obtainable leakage in terms of mutual information (MI) and fully leaked bits (FB) for different curves in wolfSSL und OpenSSL. This table complements Table 2. Data collection described in Section 6.1.

wolfSSL			OpenSSL		
Event	MI/FB	Pr[A=a]	Event	MI/FB	Pr[A=a]
(W_1, W_2)	[bit / bit]		(O_1, O_2)	[bit / bit]	
bp160r1					
-	-	-	(0, *)	0.1 / 0	0.81
			(1, 0)	1.3 / 1	0.11
			(1, 1)	1 / 1	0.08
bp192r1					
-	-	-	(0, *)	0.4 / 0	0.64
			(1, 0)	0.7 / 0	0.13
			(1, 1)	<0.1 / 0	0.23
bp256r1					
(1, 1)	0.2 / 0	0.24	(0, *)	0.4 / 0	0.85
(1, 2)	0.6 / 0	0.61	(1, *)	1.1 / 0	0.15
(2, *)	1.1 / 0	0.15			
bp224r1					
(1, 1)	<0.1 / 0	0.35	(0, *)	0.1 / 0	0.92
(1, 2)	0.3 / 0	0.56	(1, 0)	7 / 6	$1.6 \cdot 10^{-4}$
bp320r1					
(1, 1)	<0.1 / 0	0.16	(0, *)	0.3 / 0	0.54
(1, 2)	0.6 / 0	0.38	(1, 0)	0.1 / 0	0.28
(2, 3)	<0.1 / 0	0.38	(2, *)	3 / 3	$1.7 \cdot 10^{-3}$
(2, 4)	<0.1 / 0	0.8			
bp384r1					
(1, 1)	0.6 / 0	0.25	(0, *)	0.7 / 0	0.95
(1, 2)	0.8 / 0	0.70	(1, *)	3.5 / 0	0.05
secp128r1					
(1, 1)	<0.1 / 0	0.30	(0, *)	<0.1 / 0	0.77
(1, 2)	0.3 / 0	0.47	(1, *)	1.3 / 1	0.23
(2, *)	1.3 / 1	0.23			
secp192r1					
(1, *)	0.2 / 0	0.5	(0, *)	0.2 / 0	0.5
(2, *)	0.2 / 0	0.5	(1, *)	0.2 / 0	0.5

9

BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments

Publication

Jesse De Meulemeester*, **Luca Wilke***, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede and Jo Van Bulck. *BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments*. In *IEEE Symposium on Security and Privacy (SP)*, 2025.

Contribution

Jesse and I are co-first authors. Jesse conceived the idea for the SPD manipulation and performed the corresponding experiments. I worked on making the resulting aliases useable from software and developed the attacks against SEV-SNP.

Outline

BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments

Jesse De Meulemeester[1]*, **Luca Wilke**[2]*, David Oswald[3], Thomas Eisenbarth[2], Ingrid Verbauwhede[1] and Jo Van Bulck[4].

[1]COSIC, KU Leuven
[2]University of Lübeck
[3]University of Birmingham
[4]DistriNet, KU Leuven

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

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

1 Introduction

Cloud computing has become an important paradigm that takes advantage of the economy of scale by sharing platform resources among mutually distrusting tenants. Traditionally, a privileged hypervisor software layer orchestrates and isolates different guest Virtual Machines (VMs). However, in this paradigm, cloud users must assume

the hypervisor to be free from exploitable vulnerabilities, as well as trust the cloud service provider's administrators, staff with physical access, and local law enforcement. In response to these concerns, Trusted Execution Environments (TEEs) have been developed, including AMD's Secure Encrypted Virtualization (SEV) [48], Intel's Software Guard Extensions (SGX) [46, 58] and Trusted Domain Extensions (TDX) [31], and Arm's Confidential Compute Architecture (CCA) [6]. TEEs aim to facilitate private computations even in the presence of an untrusted hypervisor, guarding against both privileged software-level and hardware-level attacks.

To this end, TEEs implement strong, hardware-enforced access control mechanisms to protect data in use within the trusted CPU package while transparently encrypting all data before writing it to untrusted off-chip DRAM. Therefore, TEEs safeguard data confidentiality against advanced physical adversaries employing cold boot attacks [84] or DRAM interposers [50]. Initial TEE designs, like Intel SGX, prioritized additional strong cryptographic integrity and freshness protection to thwart data modification and replay attacks. However, ensuring freshness requires secure on-chip storage for the root of the integrity tree, which does not scale effectively with larger memory sizes. Consequently, these initial designs are limited to a relatively small memory region (*i.e.*, 128 or 256 MB) [26]. A clear industry shift, exemplified by Scalable SGX, TDX, SEV, and CCA, has since extended protection to full VMs, thereby scaling memory encryption to encompass the entire DRAM. However, this expansion may come at the cost of theoretically reducing the strength of cryptographic integrity guarantees against physical adversaries [46].

While it has been established that data encryption cannot conceal address access patterns [19, 50], and scaling up memory encryption may introduce powerful ciphertext side channels [51, 54] in code that is otherwise constant time, no practical integrity breaches have been demonstrated to date. One reason for this is that the cost of such advanced physical attacks is considered to be exceedingly high. For instance, the sole previous hardware attack that managed to extract access patterns from SGX's memory encryption engine required a prohibitive investment of $170,000 for a DDR4 DRAM interposer [50]. It is worth noting that this interposer requires continuous physical access and may not even be fast enough to manipulate or replay data, let alone target more recent DDR5 technologies. Thus, in this paper, we analyze the remaining DRAM trust assumptions, considering the following fundamental questions:

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

Exploring a new research direction, we focus our attention on the memory subsystem's

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

In our attacks, we double the apparent size of the Dual Inline Memory Module (DIMM) installed in the system to trick the CPU's memory controller into using additional "ghost" addressing bits. These addressing bits will be unused within the virtually enlarged DIMM, creating an interesting *aliasing* effect where two different physical addresses now refer to the same DRAM location. We develop a practical reverse-engineering method to locate these aliases and show that they can be exploited to bypass access control restrictions, including those implemented by TEEs. Most impactful, in the case of AMD SEV-SNP, we show that BadRAM attackers can tamper with or replay ciphertexts and even manipulate the crucial reverse map table data structure, thereby re-introducing potent page-remapping attacks [61, 62, 63] that initially prompted the development of SEV-SNP. Building upon these primitives, we construct a comprehensive, end-to-end attack that allows replaying the cryptographic launch digest used in SEV-SNP's attestation process. We experimentally demonstrate that this capability permits the launching of arbitrarily modified VM images without altering their attestation report, consequently undermining all trust in the SEV-SNP ecosystem.

Next, we analyze the effect of our BadRAM aliasing primitive on the security of other popular TEEs beyond AMD SEV-SNP. We find that "classic" Intel SGX incorporates suitable cryptographic integrity protections that effectively thwart ciphertext replay or corruption attacks but still allow BadRAM adversaries to discern precise, noiseless write access patterns at a fraction of the cost of prior work [50]. Conversely, Scalable SGX and TDX include a trusted code module that explicitly checks the physical memory space for aliases during boot time, preventing our attacks at present. Following our responsible disclosure, AMD plans to introduce a similar countermeasure through a firmware update for SEV-SNP.

Contributions

Our main contributions are as follows:

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

Responsible Disclosure

We disclosed the SPD aliasing attacks with proof-of-concepts to break SEV-SNP to AMD on February 26, 2024. AMD acknowledged our findings, which they are tracking under CVE-2024-21944 and AMD-SB-7022, and requested an embargo until December 10, 2024. Notably, AMD's official CVSS assessment (`AV:L/AC:H/PR:H/UI:N/S:C/C:N/I:H/A:N`) acknowledges that BadRAM attacks can be mounted by local, software-only attackers without physical access (e.g., via SSH). For the issue in classic Intel SGX, we did not deem disclosure necessary at this point, as the underlying problem of write-pattern leakage has been demonstrated before. To mitigate our findings, AMD will interactively check the DRAM configuration using "AMD Secure Boot loader firmware". Section A contains AMD's verbatim response.

Open Science

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

2 Background

2.1 AMD Secure Encrypted Virtualization

AMD's Secure Encrypted Virtualization (SEV) [48] is a TEE that protects virtual machines against privileged software attackers, such as a malicious hypervisor. The intended use case is to run VMs in the cloud without needing to trust the cloud service provider. SEV uses a combination of access rights and memory encryption to protect the VM's data. Before writing data to DRAM, it is encrypted with AES XOR-Encrypt-XOR (AES-XEX) using a tweak value derived from the physical address [48, 67, 80]. The encryption keys are managed by the AMD Secure Processor (SP), which forms the hardware root of trust for the system. The SP offers an API to the hypervisor to manage encrypted VMs. To protect the VM's plaintext data inside the system-on-chip, e.g., while it is in the cache, the data is tagged with a VM-specific identifier to restrict access to the corresponding VM. The updated version SEV Encrypted State (SEV-ES) [47] added encryption to the previously unprotected VM register file. The latest version, SEV Secure Nested Paging (SEV-SNP) [1, 5], mainly adds integrity protection to both the VM's memory content and its memory layout, preventing an attacker from writing to an encrypted VM's memory from software and restricting their ability to remap the VM's secure memory pages. Both mechanisms are implemented via an additional, hardware-managed data structure called the Reverse Map Table.

2.2 DRAM Organization

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

Accessing a certain memory location first requires activating the corresponding row. As only one row can be open at a time within a bank, switching between rows incurs a performance penalty. The translation of physical address to DRAM address bits, performed by the memory controller, is, therefore, not a one-to-one mapping. For instance, the channel, rank, and bank bits are typically obtained by XORing different physical address bits together [65, 76]. This spreads consecutive addresses over different channels, ranks, and banks, reducing the performance overhead.

2.3 Serial Presence Detect

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

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

Write Protection

To protect against accidental overwrites, each block can be optionally write protected. As per JEDEC specification, this write protection must be reversible, though doing so requires physical access to the DIMM. For DDR4, reverting the write protection requires connecting I^2C addressing pin SA0 to V_{HV} (7–10 V) and issuing the Clear all Write Protection (CWP) command [44]. For DDR5, the protection status is stored in registers MR12 and MR13, which can be modified by tying the HSA pin to ground [45].

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

3 BadRAM Memory Aliasing Primitive

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

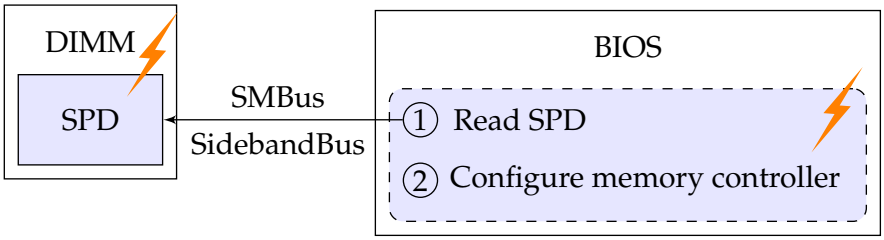


Figure 1: High level overview of the memory configuration steps performed by the BIOS. An incorrect DIMM topology can be either the result of a malicious SPD, or a malicious BIOS.

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

3.1 Attacker Model

For our attacks, we assume an attacker with *(i)* root privileges on the target system and *(ii)* one-time physical access to a DIMM module installed in the system. Assumption *(i)*, *i.e.*, software root access, is the standard adversary model in the TEE context and in principle also includes arbitrary code execution in the BIOS. However, as BIOSes are notoriously inaccessible to end users, we introduce assumption *(ii)*, recognizing that TEEs generally assert a degree of protection against physical DRAM attacks. Intel SGX and TDX explicitly consider the DRAM subsystem untrusted [26, 35]. Similarly, AMD SEV-SNP considers certain DRAM attacks, such as cold boot attacks, as part of their threat model, while "on-line DRAM integrity attacks, such as attacking the DDR bus while the VM is actively running" are considered out of scope as they are deemed "very complex and require a significant level of local access and resources to perform" [1].

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

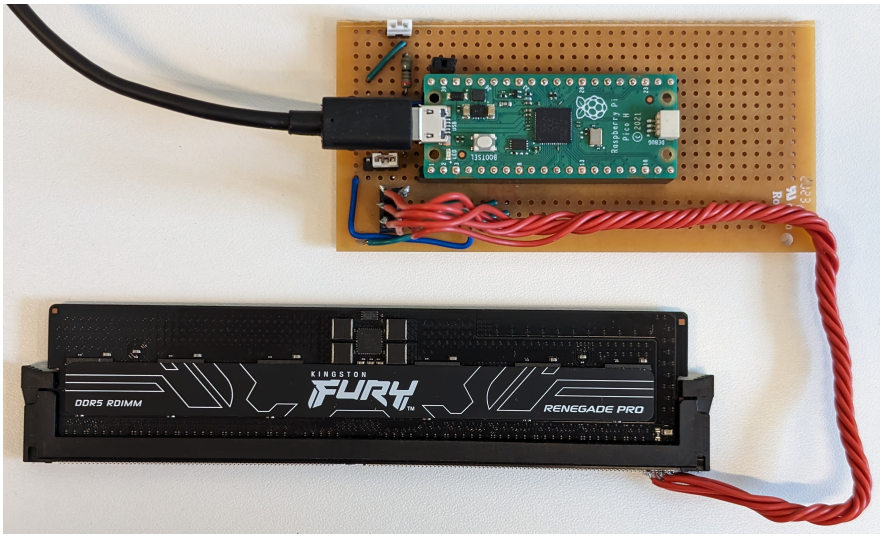


Figure 2: Raspberry Pi Pico setup to unlock and modify DDR4 and DDR5 SPDs.

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

Figure 3: Encoding of MR12 and MR13 for DDR5 [45]. BadRAM SPD modification attacks require bits 0 and 7 of MR12 to be clear.

3.2 Modifying SPD Contents

Experimental Setup

To physically interface with the SPD chip, we connected a standard Raspberry Pi Pico to the I^2C interface exposed on the DIMM, as shown in Figure 2. This offers a direct connection to the EEPROM, enabling read and write access, and allows to disable any potential write protection. As the SPD lacks authentication measures, its contents can be altered without detection by the system. This has been used before to adjust the DIMM's frequency [27, 49], or to create counterfeit memory modules [73]. Note that while DDR5 supports I3C, it remains backward compatible with I^2C and defaults to I^2C on power on [45]. The total setup cost to perform SPD modifications, which includes the Raspberry Pi Pico and DDR sockets, is approximately $10. Section B includes a full parts list.

DRAM Vendor Analysis

We analyzed several off-the-shelf DDR4 and DDR5 memory modules, including RDIMMs, UDIMMs, and SODIMMs, as summarized in Table 1. Performing a BadRAM attack requires modifications to block 0 for DDR4 and blocks 0 and 7 for DDR5 (as DDR5 stores

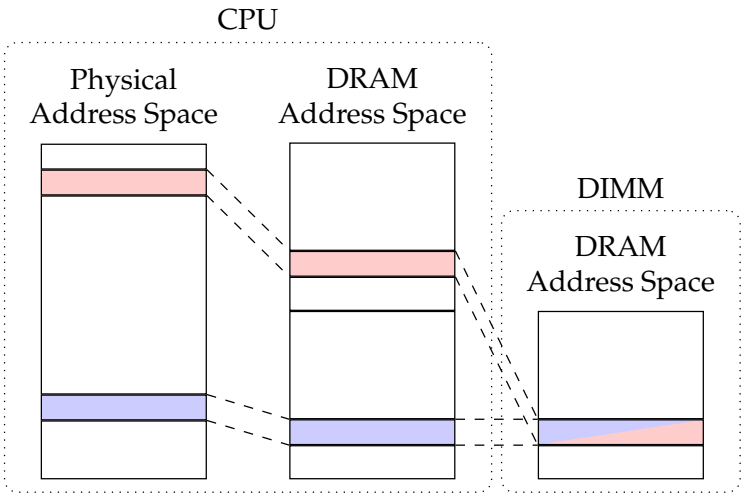


Figure 4: An incorrect configuration of the memory controller can result in unused address bits. Two addresses that only differ in the ghost bit alias to the same physical location inside the DRAM memory as this bit is ignored.

the CRC in a separate block). The protection of these blocks is defined by WP0 for DDR4 and bits 0 and 7 of register MR12 for DDR5 (cf. Figure 3).

We found that most, though not all, memory modules lock the base configuration by default, as required by JEDEC, though they do not all set the remaining protection bits equally. We experimentally verified the ability to remove this protection with physical access as per the JEDEC specification. This operation can be performed entirely using the Raspberry Pi, with DDR4 only requiring an additional 7–10 V source, such as a boost converter or a 9 V battery. Notably, we found at least two off-the-shelf DDR4 DIMMs (Corsair_1 and Corsair_2) that leave the base configuration entirely unprotected, possibly exposing them to software-only BadRAM attacks. In specific cases, this may even lead to accidental corruption of the SPD, a known problem for some motherboards [56]. However, as these particular modules are UDIMMs, they are not compatible with our test server systems.

3.3 Creating Memory Aliases

With the ability to modify the SPD data, we can change the base configuration information of the DIMM. While not changing the underlying physical properties of the DIMM, this will change the properties as perceived by the host system. For instance, we can change the addressing that is used for the DIMM. The memory controller relies on this information to construct its physical-to-DRAM address mapping. This mapping depends, for instance, on the number of bits that are required to address each level of the DIMM's

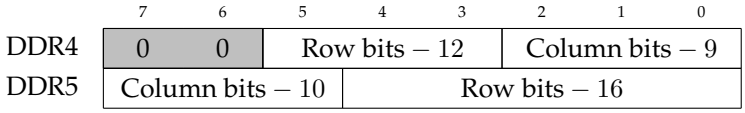


Figure 5: Encoding of byte 5 of the SPD's content, representing the SDRAM addressing [41, 42]. This byte is part of block 0 in the SPD.

hierarchy [65, 76]. An incorrect configuration, as reported by the SPD, can create an inconsistent memory view between the CPU and DIMM. For instance, if the SPD reports more addressing bits than are actually used by the module, effectively increasing the size of the DIMM, the CPU will accordingly incorporate these bits into its mapping. However, since these "ghost" bits are not used by the DIMM, they are effectively ignored, creating aliases in the CPU's memory view.

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

SPD Encoding

As mentioned before, the required information to correctly address the DIMM is stored in the SPD. For instance, byte 5 encodes the number of row and column bits in use by the DIMM, as shown in Figure 5. To introduce an additional DRAM address bit, we can modify this byte to either increment the number of rows or columns. This necessarily also requires a modification to the DRAM density per die (byte 4, bits 0–3 for DDR4 and bits 0–4 for DDR5), which has to be updated to reflect the doubled capacity due to the additional addressing bit. Finally, the CRC bytes must be updated to match the modified content.

When booting a system containing a DIMM with a modified SPD, we found that some BIOSes may cache the SPD contents of the DIMM based on its serial number. This is, for instance, the case in coreboot, an open-source BIOS implementation [17, `src/include/spd_cache.h`]. The changes in the SPD may, therefore, not be applied if the module was already part of the system before. Modifying the serial number (bytes 325–328 for DDR4 or 517–520 for DDR5) simulates inserting a different module and thus

forces the system to re-read the modified addressing information in the EEPROM. This behavior highlights that these memory mapping manipulations could also be performed by a malicious BIOS, which we discuss further in Section 6.1.

In our experiments, we opted to increment the number of row address bits. These bits are typically mapped to the highest physical address bits [65, 76], making unintended aliases, which impact system stability, less likely. Additionally, the column bits are typically combined in linear functions to determine the channel, rank, and bank, making the search for aliases more complicated. Assuming common DIMM properties (*i.e.*, a 64-bit interface, an 8 kB row size, and 16 (32) banks for DDR4 (DDR5)), all single-rank DIMMs with a capacity up to 16 GB for DDR4 and 32 GB for DDR5 will have at least one free row address bit. This capacity increases with additional ranks; the maximal capacity for DIMMs susceptible to row-based BadRAM attacks for common rank configurations is given in Table 2. In practice, however, this restriction does not significantly limit the attack surface as servers typically have many DIMM slots and only one modified DIMM is required for our attacks.

Finding Memory Aliases

In contrast to the simple example from the previous paragraph, physical address bits need not map one-to-one to DRAM address bits. The ghost bit, therefore, does not necessarily correspond to a single physical address bit. Thus, finding two aliasing addresses may require scanning the entire physical memory space. However, this step needs to be done only once per memory configuration, as the mapping is deterministic. On a high level, we can search for the alias of address $\mathcal{A}$ by writing a marker value to it and scanning the remaining memory for another appearance of this marker. The process is slightly complicated by memory scrambling [84], where the memory controller XORs a randomized scramble pattern to the payload data before writing it to DRAM to even out the electrical load on the memory bus. As the scramble pattern is based on the physical address, a different pattern will be applied when reading from address $\mathcal{A}$ and its alias, hiding that they are, in fact, containing the same marker value.

To find the aliased address for $\mathcal{A}$, we use the approach shown in Algorithm 2. The flush operations are required because there is no cache coherency for aliased physical addresses. By comparing the XOR values on line 7, we ensure that the effect of memory scrambling cancels out. Note that for this one-time search, we temporarily disable memory encryption features like AMD SME [3, §7.10] or Intel TME-MK [33], as they encrypt the memory contents using AES with an address-based tweak. This is no longer a linear operation, and thus cannot be canceled out by an XOR-based comparison. For both SME and TME-MK, the encryption status can be configured with page granularity at

Algorithm 2 Search alias for physical address $\mathcal{A}$.

for each page-aligned physical address $\mathcal{B} \neq \mathcal{A}$ **do**
  $m_1, m_2 \leftarrow$ 64 random bytes;
  $flush\,(\mathcal{B})\,;\ write\_mem\,(\mathcal{A}, m_1)\,;\ flush\,(\mathcal{A})$
  $g_1 \leftarrow read\_mem(\mathcal{B});\ flush\,(\mathcal{B})$
  $write\_mem\,(\mathcal{A}, m_2)\,;\ flush\,(\mathcal{A})$
  $g_2 \leftarrow read\_mem(\mathcal{B})$
  if $m_1 \oplus m_2 = g_1 \oplus g_2$ **then**
    return $\mathcal{B}$
  end if
end for

runtime or for the whole system via BIOS settings. Some mainboards also allow memory scrambling to be disabled in the BIOS, simplifying the alias scanning to just looking for a second appearance of the marker value.

Evaluation

We evaluated our memory aliasing primitive on three different systems, which we refer to as AMD_1, Intel_1, and Intel_2 (cf. Table 3). On all systems, we modified the SPD to report one additional row and thus twice the actual memory size. To ensure stable system operation, we must prevent the kernel and all applications from using the introduced ghost memory regions to avoid accidental overwrites. We achieve this via the Linux kernel command line parameter `memmap` $=$ `nn$ss`, which marks the memory region ss to $ss + nn$ as reserved, preventing the system from using it [55]. On all systems, booting with the upper half of the memory blocked by `memmap` resulted in a largely stable system. Our alias search implementation consists of a user space application implementing the core logic in Algorithm 2, assisted by a small Linux kernel module that enables direct access to arbitrary physical addresses.

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

"memory hoisting" [4, 40], which allows applying offset-based modifications to the way physical addresses map to DRAM.

Table 1: Write protection and addressing (highlighted) for various DIMMs. Full version in Appendix (Tables 7 and 8).

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

Table 2: Maximal DIMM capacity susceptible to row-based BadRAM attacks.

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

Table 3: Overview of evaluation systems used in this paper.

System	TEE	Mainboard	CPU	DIMM(s)	DRAM
AMD_1	SEV-SNP	ASRock ROMED8-2T	EPYC 7313P	$1\times Micron_1$	DDR4
$Intel_1$	Classic SGX	Intel NUC7i3BNH	i3-7100U	$1\times Crucial_1$	DDR4
$Intel_2$	Scalable SGX	Supermicro X12DPi-NT6	Xeon 6330	$16\times Micron_3$	DDR4
$Intel_3$	TDX	ProLiant DL320 Gen11	Xeon 5515+	$8\times Kingston_7$	DDR5

4 Breaking AMD SEV-SNP

In this section, we show how the BadRAM primitive can be used to break SEV-SNP's newly introduced central memory integrity claim: "[...] if a VM is able to read a private (encrypted) page of memory, it must always read the value it last wrote" [1]. To this end, SEV-SNP imposes additional restrictions on the untrusted hypervisor to protect the integrity of the VM's memory layout and prohibit writing to its encrypted pages. In this section, we first explain how these features are implemented and then show how the protection can be broken using the BadRAM primitive. Finally, we demonstrate an end-to-end attack that breaks SEV-SNP's attestation, allowing an attacker to make arbitrary changes to a protected VM without changing its attestation report, breaking *all trust* in SEV-SNP.

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

SEV-SNP's integrity features are implemented via the newly introduced Reverse Map Table (RMP). The RMP is a linear table that contains one entry for each HPA page that should be assignable to SEV-SNP VMs. Each RMP entry records various attributes. The most important ones are whether the page is used by an SEV-SNP VM and, if so, the GPA at which the page is supposed to be mapped within the VM. Following AMD's nomenclature, we will call pages used by SEV-SNP VMs *guest-owned* and all other pages *hypervisor-owned*. The RMP has to be allocated before architecturally enabling SEV-SNP, by specifying its physically contiguous memory range via the `RMP_BASE` and `RMP_END` MSRs [3, §15.26.4]. Afterward, the hypervisor can no longer write to this memory region. Instead, it has to use a newly introduced set of instructions that grant it restricted access to the RMP. Thus, in contrast to the NPT, the information in the RMP is trustworthy. In the following, we show how we can use the BadRAM primitive to break both the memory layout integrity and the memory content integrity introduced by the RMP.

4.1 Breaking Memory Layout Integrity

Prior to SEV-SNP, there was no mechanism for a VM to detect changes in the GPA to HPA mapping performed by the hypervisor-controlled NPT. Morbitzer et al. exploited this for their SEVered attacks [61, 62, 63], which use the ability to swap the memory mapping of

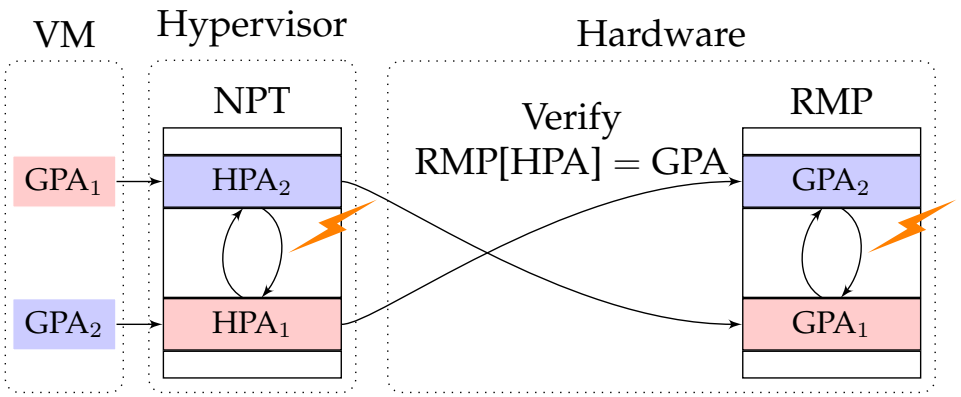


Figure 6: Mapping manipulation required for remapping attacks on SEV-SNP. The hardware checks that the HPA, as translated by the NPT, matches the expected GPA specified in the RMP. This enables the hardware to detect malicious mapping changes through the hypervisor controlled NPT. Thus, the attacker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

two pages in conjunction with a service running inside the VM, to decrypt arbitrary VM memory, inject hypervisor chosen plaintext and execute arbitrary code in the VM.

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

Using the BadRAM primitive, the hypervisor can circumvent the write protection of the RMP itself and, thus, make arbitrary changes to the stored GPA values without clearing the valid bit. Crucially, we find that the RMP's content is not encrypted. Thus, the hypervisor can directly write to the RMP and does not have to resort to replaying previously captured ciphertexts. As a result, the hypervisor can trivially swap the GPA-to-HPA mapping of two pages, by swapping both the GPA-to-HPA mapping in the NPT and the expected GPA in the corresponding RMP entry, as shown in Figure 6. This re-enables SEVered attacks [61, 62, 63], which, in conjunction with a service running

in the VM, allow both decryption and encryption of SEV-SNP VM memory as well as code execution. Note that, as each GPA is inherently associated with exactly one RMP entry at a time, the hypervisor still cannot map two GPAs to the same HPA at the same time. However, such aliasing is not required for the cited attacks. If desired, adversaries can use single-stepping, e.g., SEV-Step [81], to precisely manipulate RMP entries at a maximal, instruction-level temporal resolution.

Proof-of-Concept

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

4.2 Breaking Memory Content Integrity

Safeguarding the integrity of encrypted memory content in SEV-SNP is not enforced via cryptographic measures, but solely relies on the RMP. If SEV-SNP is enabled, the RMP is consulted for each memory write performed by the hypervisor [3, Table 15-39]. If the targeted page is guest-owned, the write attempt is blocked. Crucially, using the BadRAM primitive, we can circumvent this RMP protection by ensuring that the alias is hypervisor-owned. While the hypervisor can now write to guest-owned pages, their content is still encrypted with AES-XEX. As AES-XEX does not offer integrity, the guest cannot detect if a ciphertext has been manipulated. Instead, a modified ciphertext simply decrypts to a randomized value, i.e., it is not possible to make controlled semantic changes to the plaintext. Nonetheless, manipulating the ciphertext can be used as a capable fault primitive, e.g., against cryptographic schemes [11].

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

Proof-of-Concept

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

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

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

SEV-SNP Attestation

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

1. In `GSTATE_LAUNCH`, the hypervisor creates the VM and prepares its memory content by using the corresponding launch API functions of the Secure Processor (SP), SEV's hardware root-of-trust. First, the hypervisor donates memory for the guest context data structure to the SP. The hypervisor has to mark the donated memory page as a firmware page in the RMP, preventing future writes. Next, the SP encrypts the donated memory using its memory encryption key and initializes the guest context. The guest context is the central data structure describing the SEV VM. Among other information, the guest context stores the launch digest, a cryptographic hash representing both the initial memory content and the initial memory layout of the VM.

2. Next, the initial memory content of the SEV VM is loaded by the hypervisor via the `SNP_LAUNCH_UPDATE` command offered by the SP. This command encrypts the memory with the VM's memory encryption key, without revealing the key to the hypervisor. On each invocation, the SP updates the launch digest inside the guest context accordingly. Eventually, the hypervisor uses the `SNP_LAUNCH_FINISH` command to transition the VM into the `GSTATE_RUNNING` state, which disables the launch commands API. The `GSTATE_RUNNING` state marks the VM as "runnable", allowing the hypervisor to start the VM via the `VMRUN` instruction.

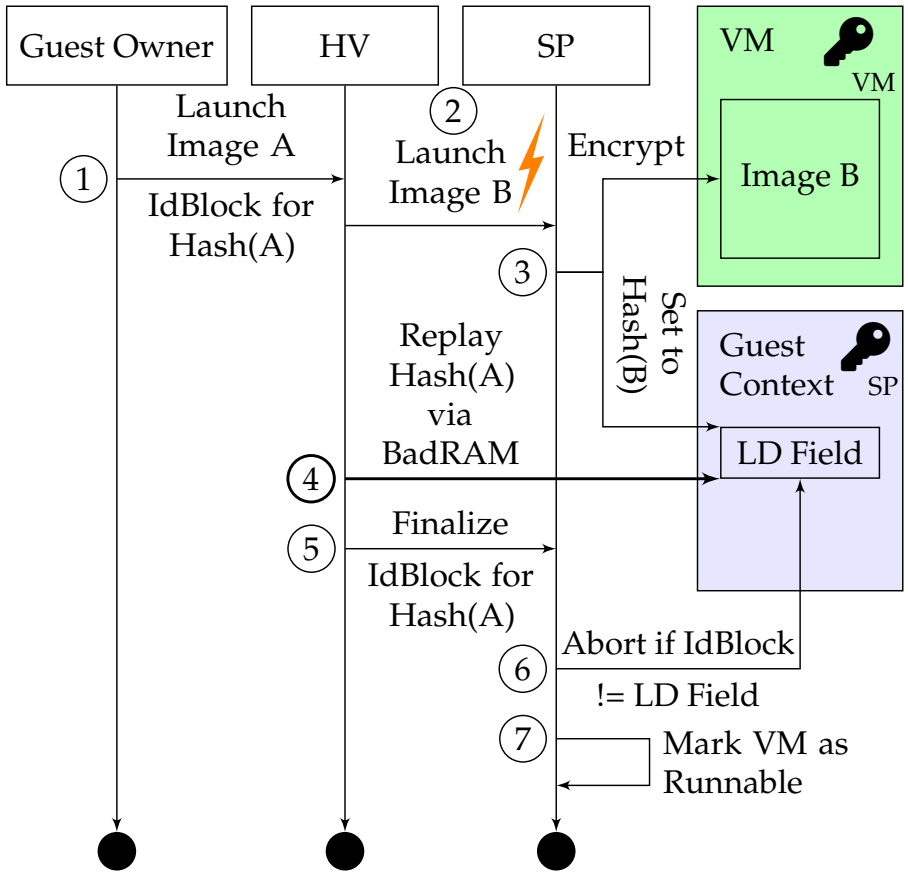


Figure 7: Online phase of the launch digest replay attack that breaks SEV-SNP's attestation. In step ① the correct image, together with a signed representation of the expected launch digest, is transferred to the hypervisor. In step ②, the hypervisor modifies the requested VM image to contain a backdoor.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

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

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

Replaying the Launch Digest

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

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

In the online phase, the hypervisor donates the same physical memory page as in the offline phase to be used as the guest context page. Every time an SEV VM is created, the SP assigns it a fresh memory encryption key. Thus, the hypervisor cannot use any of the VM's ciphertext from the offline phase for replay attacks. However, the memory encryption key of the SP itself is only regenerated when the system reboots. Crucially, the guest context pages of all SEV VMs are encrypted with the SP's single memory encryption key. Since both the physical memory location of the guest context page and the memory encryption key are the same between the online and the offline phase, the hypervisor can replay the ciphertext of the benign launch digest from the offline phase in the online phase. Thus, after receiving the requested VM image in step ① the hypervisor

can make arbitrary changes in step ②, as it can simply replay the previously captured benign launch digest in step ③ before finalizing the VM in step ④, which triggers a check of the launch digest in step ⑤. Due to the replay, the check succeeds, and the SP marks the VM as runnable in step ⑥. Note that the alternative, VM-triggered attestation procedure can only take place *after* step ⑥ and is, thus, also rendered useless by the launch digest replay.

To determine the exact offset of the 48-byte launch digest inside the encrypted guest context page, we use an empirical approach and dump the ciphertext of the guest context page between calls to `SNP_LAUNCH_UPDATE`. Next, we compute the difference between these dumps, revealing that only the 64 bytes from offset `0x460` to `0x4A0` change every time. Due to the 16-byte block size of SEV's memory encryption, the difference is only 16-byte granular. We correlate this information with the publicly available source code of the SP's firmware [2, `sev_rmp.h : 226`], indicating that the array for the measurement is not 16-byte aligned, causing our replay to also overwrite the `IMIEn` field as well as parts of the 16-byte `GOSVW` field [5, Table 6]. However, both fields represent configuration options that cannot change between capturing and replaying, since we have to start the SEV-SNP VM with the same configuration options both times anyway.

End-to-End Attack

We analyze the use case where SEV-SNP is used together with a disk image, to bring up a fully-fledged VM, as described in [24, 57, 66, 79]. The disk image needs to use regular Linux full disk encryption to ensure the confidentiality and integrity of its content, as SEV-SNP does not offer any protection for virtual disks. The VM first boots into a minimal environment that runs a small server to provide the attestation report to the guest owner and subsequently establishes a secure communication channel. This minimal environment corresponds to the image that gets loaded in step ① in Figure 7 and thus is protected by the SEV-SNP. After verifying the report, the guest owner uses the secure channel to send the disk decryption key to the VM, which subsequently unlocks the disk and boots into the rich Linux environment contained inside the now-unlocked disk.

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

Table 4: Vulnerability of popular TEEs to BadRAM attacks.

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

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

to the guest owner, making the attack undetectable. Alternatively, the malicious code could also be enhanced to delete itself before allowing the guest owner to log into the VM.

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

5 Analyzing DRAM Trust in Popular TEEs

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

5.1 Classic Intel SGX

Memory Encryption

The original Intel SGX architecture considered the DRAM as entirely untrusted storage, featuring a dedicated Memory Encryption Engine (MEE) to encrypt and integrity-protect all enclave data stored in memory [26]. The MEE uses an AES-CTR-based cryptographic scheme that ensures the confidentiality, integrity, and freshness of the ciphertexts. It

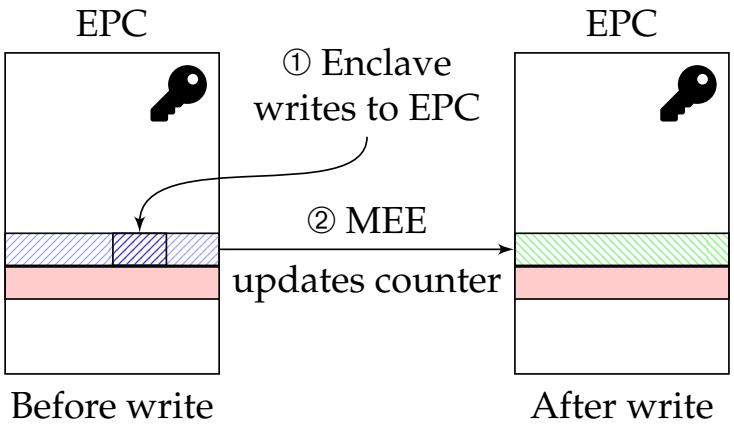


Figure 8: Using the BadRAM aliases, an adversary can monitor the EPC of classic SGX for changes in ciphertexts, revealing the write pattern. As SGX uses a fresh counter on every write, the ciphertext changes on every write, independent of the plaintext.

features a Merkle tree that guarantees the freshness of the counter values by storing the root of the tree in on-chip memory, inaccessible to malicious DRAM, with a fresh counter value generated on every memory write. Additionally, a 56-bit Message Authentication Code (MAC) ensures the integrity of the ciphertext. Therefore, both replayed and manipulated ciphertexts can be detected, locking the processor upon such violations.

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

Write Access Patterns

Crucially, BadRAM attackers can circumvent SGX's contiguous EPC check, enabling read and write access to enclave ciphertexts from software. While ciphertext modifications would be detected by the MEE, BadRAM attackers may still monitor the ciphertext for changes as a capable side-channel. Figure 8 shows that, when the enclave writes to its private memory, the corresponding ciphertext *always* changes, regardless of whether the underlying plaintext has changed. This is because the MEE is explicitly designed to assign a fresh counter on every write, protecting against the content-based ciphertext side-channels demonstrated on SEV platforms [51, 54].

Therefore, although the contents of memory accesses are effectively protected by the MEE, BadRAM adversaries who monitor relative changes in ciphertext over time can

precisely deduce the location of write operations. As the CPU always writes back a whole cache line at a time, the obtained write address pattern has a spatial resolution of 64 bytes, which can be observed at a maximal, instruction-level temporal granularity using a single-stepping framework like SGX-Step [14]. This provides a more fine-grained leakage compared to an adversary that is only able to deterministically monitor page faults [83]. Additionally, in contrast to cache side channels [60], the obtained write pattern is deterministic and noiseless.

The sole previous study [50] that showcased DRAM address leakage on SGX necessitated continuous physical access and equipment costs reaching $170,000, making it unfeasible for most adversaries. Notably, however, such a full interposer-based setup can also reveal the read pattern, which is not possible with the BadRAM aliasing attack since reading does not alter the ciphertext.

Evaluation

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

5.2 Scalable SGX

Memory Encryption

The requirement to maintain a dedicated Merkle tree in the classic SGX design does not scale well to large EPC sizes, prompting Intel to transition to a new, “Scalable” SGX design for Xeon server processors [46]. Scalable SGX abandoned the MEE and instead repurposes Intel Total Memory Encryption (TME).

TME uses AES-XTS to encrypt the entire physical memory range, but does not offer cryptographic integrity or replay protection. Thus, an adversary that can write to the EPC memory range, for instance, through an alias, would be able to corrupt or replay ciphertexts similar to our attacks on SEV-SNP. Indeed, similar to SEV, the tweak values used for AEX-XTS solely depend on the physical address and do not change during

runtime. Thus, adversaries capable of reading encrypted EPC data would also be able to perform ciphertext side-channel attacks [51, 54].

Alias Checks

Notably, we found that Scalable SGX comes with dedicated architectural countermeasures against BadRAM aliasing attacks [38, 46]. Particularly, Intel differentiates between "outside-in" and "inside-in" aliasing.

First, outside-in aliasing refers to the situation where an EPC page has an alias that itself is not part of the EPC. To protect against these kinds of attacks, Scalable SGX repurposes one of the DRAM Error Correcting Code (ECC) metadata bits as an "ownership" bit to specify whether the cache line is part of the EPC [46]. If the CPU is not currently executing an SGX enclave, the memory controller returns a fixed pattern for read accesses to cache lines that have the ownership bit set. This mitigates ciphertext side-channel attacks from software. Writing to EPC memory while the CPU is not executing an SGX enclave clears the ownership bit, which will be detected the next time an enclave tries to access the corrupted memory location.

Second, inside-in aliasing refers to the situation where an EPC page has an alias that itself is also part of the EPC. To protect against these, a trusted code module explicitly checks the physically contiguous EPC range for aliases before enabling SGX at boot time. On Xeon CPUs with Scalable SGX, the EPC can be as large as 512 GB, totaling 1 TB for a dual-socket system [34]. Initially, this check was part of SGX's `MCHECK` authenticated code module [39]. However, starting with 4th generation Xeon scalable processors, alias checking is now handled by a dedicated Alias Checking Trusted Module (ACTM) [38] that builds on Intel TXT [37] to run without trusting the BIOS.

Evaluation

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

5.3 Intel TDX

Intel Trusted Domain Extensions (TDX) is Intel's latest TEE, moving away from the enclave-based paradigm to support confidential VMs, similar to AMD SEV, which are also referred to as Trusted Domains (TDs) [31]. Similar to SGX on Xeon scalable processors, TDX relies on Total Memory Encryption-Multi-Key (TME-MK) to provide memory confidentiality by encrypting its contents with AES-XTS. In contrast to SGX, however, TDX does not come with the concept of a fixed-size EPC, lifting any artificial limits on the total amount of memory used for TDs. Instead, TDX only uses the approach introduced with Scalable SGX to provide logical integrity protection and to prevent outside-in aliasing via a dedicated "TD-owner" ECC bit. Additionally, TDX introduces optional cryptographic integrity protection by storing a 28-bit MAC in the ECC bits for each cache line. This MAC is computed over the ciphertext, address-based tweak, TD-owner bit, and MAC key, and ensures the integrity of the cache line. If the integrity check fails, the memory location is marked as "poisoned" to prevent an attacker from brute-forcing the MAC [36, §16.2.1.1].

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

Evaluation

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

5.4 Arm CCA

Similar to SEV and TDX, Arm CCA is a VM-scoped TEE, though it is not yet commercially available. Like the aforementioned TEEs, CCA considers some attacks on DRAM to be in scope [6, 8] and features memory encryption as its primary defense. Additionally, the most critical data structures—belonging to the EL3 monitor that warrants CCA's security—receive extra protection [6]. They are stored either in on-chip memory, inaccessible to an attacker controlling external memory, or in external memory, but with

additional integrity guarantees. The only exception is the Granule Protection Table (GPT) [7, §4], which is stored entirely in external memory and enforces memory isolation against software attackers, similar to SEV's RMP.

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

6 Discussion and Mitigations

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

6.1 Software-Only Adversaries

Up to this point, we have assumed an attacker with one-time physical access to the DIMMs, allowing the attacker to disable the module's write protection and overwrite the contents of the SPD. However, the SPD EEPROM may also be exposed over the SMBus or SidebandBus, allowing read and write access to the EEPROM by a privileged software-based adversary. Additionally, as the initialization of the memory controller performed by the BIOS is based on the reported SPD values, a malicious BIOS may spoof these values for a similar effect.

SMBus & SidebandBus

The SPD chip is connected to the rest of the system via the SMBus or SidebandBus for DDR4 and DDR5, respectively. This interface may be exposed to software, like the `decode-dimms` utility from Linux's `i2c-tools` that provides comprehensive information on the connected memory. Performing BadRAM attacks by leveraging this interface requires the ability to write to the SPD base parameter section to change the addressing information. However, a DIMM may set its SPD write protection to disable writes to this

section. Additionally, some memory controllers have protections in place that prevent writes to the SPD chip from software (e.g., through the SPD Write Disable (SPDWD) bit on the Intel PCH [32]), though some manufacturers allow this protection to be disabled in the BIOS, for instance to support on-DIMM RGB lighting [18].

BIOS

The BIOS reads out the SPD contents and configures the system based on the reported values, as shown in Figure 1. An adversary in control of the BIOS could change these values before configuring the system. This effectively enables BadRAM attacks without the requirement for physical access. However, as the BIOS is a complex, proprietary component (which might be cryptographically authenticated [20, 64]), this attack vector is significantly more complicated than modifying the SPD directly. While there are some efforts to create open-source firmware, such as coreboot, they currently do not support the newest TDX and SEV-SNP platforms.

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

6.2 Countermeasures

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

Improving SPD Security

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

Validating Memory Layout

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

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

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

Strong Cryptography

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

protected memory and thus making large memory sizes impractical. For instance, Classic Intel SGX—one of the few commercial TEEs providing cryptographically secure memory integrity protection—supports only up to 128 MB or 256 MB of protected memory while incurring a performance overhead of up to 14 % [26].

Protecting large amounts of memory with both strong cryptography and acceptable overhead is a challenging research question. Recent academic works provide more scalable designs by employing skewed [71] or mountable Merkle trees [23], increasing the arity of the integrity tree by reducing the counter size [68, 72], dynamically adjusting the tree's height and arity [75], and changing the underlying cryptographic primitives [30]. While these approaches enable integrity protection for larger memory sizes, they have not been adopted by industry. Furthermore, even with strong integrity and freshness guarantees, attacks on the external memory may still enable high precision, sub-page access pattern leakage, as discussed in Section 5.1. Thus, software still needs to follow oblivious, constant-time programming paradigms to avoid such leakages.

6.3 Unverified Trust in DRAM Hardware

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

The SPD manipulation from this paper essentially shows a data-driven attack against an otherwise benign BIOS. However, another attack angle would be a full DRAM interposer [50] or a DRAM module with manipulated hardware that, e.g., allows arbitrary read and write to the memory content via a second interface that is only available to the attacker. Such a hardware attacker could easily hide any manipulations from a boot time alias check, like the one performed by Intel. Hopkins et al. [29] discuss such modified memory modules for DDR3 and also implemented an FGPA-based prototype that attaches to the DRAM slot and acts as an interposer. It allows the attacker to redirect memory accesses to protected regions. We found one company that sells memory modules that come with data processing units, allowing to execute custom code directly on the DIMM [74]. However, their DIMM is currently restricted to a few mainboards, none of which support TDX.

7 Related Work

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

Memory Aliasing

Breuer et al. consider memory aliasing in a security context [10] and define two types: "software aliasing," where multiple logical addresses map to one physical address, and "hardware aliasing," where multiple physical addresses map to one logical address. They observe that the latter case can, e.g., arise when the number of address lines exceeds the bit width of the CPU arithmetic and develop methods to "certify" the safety of machine code in this scenario. More recently, Intel contributed a vulnerability class, "CWE-1257: Improper Access Control Applied to Mirrored or Aliased Memory Regions" [59], to MITRE's Common Weakness Enumeration list.

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

Software Attacks on AMD SEV

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

A long line of attacks [28, 61, 62, 63] exploits the hypervisor's control over nested page tables, breaking the integrity of the VM's memory layout. The SEVered attack [62] uses this attack primitive to trick services inside the VM to encrypt and decrypt arbitrary data. SEV-SNP was designed to mitigate this class of attacks by introducing the RMP that provides integrity to the VM's memory layout. With BadRAM, we break the RMP, re-enabling these attacks.

In [22, 80], the authors unveil the details of SEV's tweaked encryption mode, showing flaws that allow reverse engineering the tweak values. Exploiting the known tweaks,

[53, 80] show that by adjusting for the tweak differences, moving ciphertexts in memory allows building mechanisms to encrypt and decrypt arbitrary data. All SEV-SNP-enabled CPUs use strong tweak values, mitigating these attacks. Buhren et al. [11] exploit the missing integrity protection to perform fault attacks by flipping ciphertext bits, which is mitigated with SEV-SNP by the RMP's write protection. Our BadRAM primitive re-enables this attack. Li et al. [51, 54] introduce ciphertext side-channel attacks, showing that the boot-time fixed tweak values used by SEV allow leaking access patterns of the executing code. This attack is not mitigated on SEV-SNP. Starting with revision 1.55 from September 2023, the SEV-SNP spec [5] mentions a "ciphertext hiding" feature but does not provide further details.

Schlüter et al. [69, 70] exploit the hypervisor's ability to inject unexpected interrupts. In combination with insufficient sanitization by the Linux kernel running in the VM, they are able to change the VM's register values, allowing them to eventually read/write memory and execute arbitrary code. While SEV-SNP does provide additional hardware features to mitigate these attacks, there is currently no software support. Wilke et al. [81] show that the external APIC timer interrupt can be used to single-step SEV-SNP VMs, enhancing the resolution of side-channel attacks. There is no mitigation for SEV-SNP. Single-stepping is also used by [70, 85]. Cachewarp [85] exploits a microcode bug to drop cache write-backs, which has been fixed by an update. CrossLine [52] exploits improper ASID checks prior to SNP.

In summary, SEV-SNP is currently vulnerable to the following software attacks: interrupt injection [69, 70], ciphertext side-channel [51, 54], and single-stepping [81]. Using the BadRAM primitive from this paper, we re-enable fault attacks [11] as well as SEVered [62, 63] attacks, essentially downgrading SEV-SNP back to SEV-ES. With the attack on the attestation presented in this paper, we break all trust in the SEV-SNP ecosystem.

Physical Attacks on DRAM and TEEs

An attacker with physical access to the CPU can, for example, manipulate the CPU voltage, which may introduce faults within the code running on the system [9]. The VoltPillager and PMFault attacks show how an attacker can inject faults into SGX enclaves by sending packets on the various voltage regulator interfaces used on modern Intel CPUs [15, 16]. Similarly, Buhren et al. glitch the AMD Secure Processor over the same interface to break the confidentiality and attestation features of SEV-SNP [12, 13].

For DRAM specifically, Hopkins et al. [29] present a DDR3 interposer that remaps attacker-controlled addresses to protected ones when inserted between a DIMM and the CPU. They discuss placing the interposer directly on the DIMM's IC but opt for an FPGA-based implementation for their prototype that only supports DDR3 up to 800 MHz.

Similarly, Lee et al. use a commercial interposer, with a purchase price of $170,000, to capture the addresses on the DRAM bus [50]. While SGX encrypts the EPC contents, it does not protect the addresses. Their attack, dubbed Membuster, uses the captured access pattern as a side-channel to uncover secrets in non-constant-time code. On simpler, embedded systems, merely shorting or connecting an address line with tweezers or a sewing needle may, in certain cases, suffice to overcome security functionality, such as the memory protection in the Nintendo Wii [78] or the boot process of an embedded Linux system [21].

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

8 Conclusion

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

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

Acknowledgements

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

References

[1] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. `https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf`. 2020-01.

[2] AMD. *AMD-ASPFW*. `https://github.com/amd/AMD-ASPFW`. Commit 3ca6650.

[3] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Manual 24593, Rev. 3.42. AMD, 2024-03.

[4] AMD. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors*. Tech. rep. 42301, Rev. 3.14. AMD, 2013.

[5] AMD. *SEV Secure Nested Paging Firmware ABI Specification*. Tech. rep. 56860, Rev. 1.55. AMD, 2023-09.

[6] Arm. *Arm CCA Security Model*. Tech. rep. Arm DEN 0096, Version 1.0. 2021-08.

[7] Arm. *Learn the architecture - Introducing Arm Confidential Compute Architecture*. Tech. rep. Arm DEN 0125, Version 3.0. 2023-06.

[8] Michael Bartock, Murugiah Souppaya, Ryan Savino, Timothy Knoll, Uttam Shetty, Mourad Cherfaoui, Raghuram Yeluri, Akash Malhotra, and Karen Scarfone. *Hardware-Enabled Security: Enabling a Layered Approach to Platform Security for Cloud and Edge Computing Use Cases*. Tech. rep. NIST IR 8320. National Institute of Standards and Technology, 2021.

[9] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)". In: *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*. 1997. DOI: 10.1007/3-540-69053-0_4. URL: https://doi.org/10.1007/3-540-69053-0%5C_4.

[10] Peter T. Breuer and Jonathan P. Bowen. "Certifying Machine Code Safe from Hardware Aliasing: RISC is Not Necessarily Risky". In: *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers*. 2013. DOI: 10.1007/978-3-319-05032-4_27. URL: https://doi.org/10.1007/978-3-319-05032-4%5C_27.

[11] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. "Fault Attacks on Encrypted General Purpose Compute Platforms". In: *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ , USA, March 22-24, 2017*. 2017. DOI: 10.1145/3029806.3029836. URL: https://doi.org/10.1145/3029806.3029836.

[12] Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean -Pierre Seifert. "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3484779. URL: https://doi.org/10.1145/3460120.3484779.

[13] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. "Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019. DOI: 10.1145/3319535.3354216. URL: https://doi.org/10.1145/3319535.3354216.

[14] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. 2017. DOI: 10.1145/3152701.3152706. URL: https://doi.org/10.1145/3152701.3152706.

[15] Zitai Chen and David F. Oswald. "PMFault: Faulting and Bricking Server CPUs through Management Interfaces Or: A Modern Example of Halt and Catch Fire". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.2 (2023), pp. 1–23. DOI: 10.46586/TCHES.V2023.I2.1-23. URL: https://doi.org/10.46586/tches.v2023.i2.1-23.

[16] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David F. Oswald, and Flavio D. Garcia. "VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai.

[17] coreboot. *coreboot*. https://review.coreboot.org/coreboot.git. Git Repository.

[18] Corsair. *RAM: How to enable SPD Write on your ASUS Z690 motherboard*. URL: https://help.corsair.com/hc/en-us/articles/5718039999117-RAM-How-to-enable-SPD-Write-on-your-ASUS-Z690-motherboard.

[19] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: *IACR Cryptol. ePrint Arch.* (2016), p. 86. URL: http://eprint.iacr.org/2016/086.

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

[21] Brad Dixon. *pin2pwn: How to Root an Embedded Linux Box with a Sewing Needle*. DEF CON 24. 2016.

[22] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. "Secure Encrypted Virtualization is Unsecure". In: *CoRR* abs/1712.05090 (2017). arXiv: 1712.05090. URL: http://arxiv.org/abs/1712.05090.

[23] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. "Scalable Memory Protection in the PENGLAI Enclave". In: *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. 2021. URL: https://www.usenix.org/conference/osdi21/presentation/feng.

[24] Anna Galanou, Khushboo Bindlish, Luca Preibsch, Yvonne- Anne Pignolet, Christof Fetzer, and Rüdiger Kapitza. "Trustworthy confidential virtual machines for the masses". In: *Proceedings of the 24th International Middleware Conference*. 2023.

[25] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. "Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures". In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 2016. DOI: 10.1109/SP.2016.11. URL: https://doi.org/10.1109/SP.2016.11.

[26] Shay Gueron. "A Memory Encryption Engine Suitable for General Purpose Processors". In: *IACR Cryptol. ePrint Arch.* (2016), p. 204. URL: http://eprint.iacr.org/2016/204.

[27] Hannu Hartikainen. *Hacking DDR3 SPD*. 2018-05. URL: https://hannuhartikainen.fi/blog/hacking-ddr3-spd/.

[28] Felicitas Hetzelt and Robert Buhren. "Security Analysis of Encrypted Virtual Machines". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*. 2017. DOI: 10.1145/3050748.3050763. URL: https://doi.org/10.1145/3050748.3050763.

[29] Bradley D. Hopkins, John Shield, and Chris North. "Redirecting DRAM memory pages: Examining the threat of system memory Hardware Trojans". In: *2016 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016, McLean, VA, USA, May 3-5, 2016*. 2016. DOI: 10.1109/HST.2016.7495582. URL: https://doi.org/10.1109/HST.2016.7495582.

[30] Akiko Inoue, Kazuhiko Minematsu, Maya Oda, Rei Ueno, and Naofumi Homma. "ELM: A Low-Latency and Scalable Memory Encryption Scheme". In: *IEEE Trans. Inf. Forensics Secur.* 17 (2022), pp. 2628–2643. DOI: 10.1109/TIFS.2022.3188146. URL: https://doi.org/10.1109/TIFS.2022.3188146.

[31] Intel. *Architecture Specification: Intel Trust Domain Extensions (I ntel TDX) Module*. Specification 344425-005US. Intel, 2023-02.

[32] Intel. *Intel 500 Series Chipset Family On-Package Platform Controller Hub*. Datasheet Volume 2 of 2, Revision 002. Intel, 2023-02.

[33] Intel. *Intel Architecture Memory Encryption Technologies*. Revision 336907-004US. 2022-10.

[34] Intel. *Intel Processors Supporting Intel SGX*. https://web.archive.org/web/20240515140432/https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions-processors.html. Accessed on May 15th, 2024. 2024.

[35] Intel. *Intel Trust Domain Extensions*. https://cdrdv2.intel.com/v1/dl/getContent/690419. Accessed on 19.09.2024. 2023-02.

[36] Intel. *Intel Trust Domain Extensions (Intel TDX) Module Base Architecture Specification*. Revision 348549-004US. 2024-03.

[37] Intel. *Intel Trusted Execution Technology (Intel TXT)*. Manual 315168-017, Rev. 017.4. Intel, 2023-04.

[38] Intel. *Intel Xeon Scalable Processors: NEX Eagle Stream Platform, Intel Platform Security*. Tech. rep. 784473. Intel, 2023-08.

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

[40] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. "ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms". In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/jattke.

[41] JEDEC. *Annex L: Serial Presence Detect (SPD) for DDR4 SDRAM Modules*. Standard 21-C, Section 4.1.2.L-6. JEDEC, 2020-11.

[42] JEDEC. *DDR5 Serial Presence Detect (SPD) Contents*. Standard JESD400-5B. JEDEC, 2023-10.

[43] JEDEC. *Definition of the EE1002 and EE1002A Serial Presence Detect (SPD) EEPROMS*. Standard 21-C, Section 4.1.3. JEDEC, 2022-05.

[44] JEDEC. *Definitions of the EE1004-v 4 Kbit Serial Presence Detect (SPD) EEPROM and TSE2004av 4 Kbit SPD EEPROM with Temperature Sensor (TS) for Memory Module Applications*. Standard 21-C, Section 4.1.6. JEDEC, 2022-05.

[45] JEDEC. *SPD5118 Hub and Serial Presence Detect Device Standard*. Standard JESD300-5B.01. JEDEC, 2023-05.

[46] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. *Supporting Intel SGX on Multi-Socket Platforms*. White Paper. Intel, 2021.

[47] David Kaplan. *Protecting VM Register state with SEV-ES*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf. 2017-02.

[48] David Kaplan, Jeremy Powell, and Wolle. *AMD Memory Encryption*. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf. 2021-10.

[49] Zak Kemble. *Modifying RAM SPD Data*. 2016-03. URL: https://blog.zakkemble.net/modifying-ram-spd-data/.

[50] 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: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lee-dayeol.

[51] 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: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. 2022. DOI: 10.1109/SP46214.2022.9833768. URL: https://doi.org/10.1109/SP46214.2022.9833768.

[52] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. "CrossLine: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3485253. URL: https://doi.org/10.1145/3460120.3485253.

[53] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. "Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan.

[54] 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: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan.

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

[56] Zhiye Liu. *Gigabyte Motherboard Firmware Update: Saving Your DDR5 RAM From Corruption*. 2023-09. URL: https://www.tomshardware.com/news/gigabyte-motherboard-firmware-update-saving-your-ddr5-ram-from-corruption.

[57] Sergio López. *libkrun*. https://github.com/containers/libkrun. GitHub Repository.

[58] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative instructions and software model for isolated execution". In: *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*. 2013. DOI: 10.1145/2487726.2488368. URL: https://doi.org/10.1145/2487726.2488368.

[59] Mitre. *CWE-1257: Improper Access Control Applied to Mirrored or Aliased Memory Regions*. https://cwe.mitre.org/data/definitions/1257.html. 2020.

[60] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. 2017. DOI: 10.1007/978-3-319-66787-4_4. URL: https://doi.org/10.1007/978-3-319-66787-4%5C_4.

[61] Mathias Morbitzer, Manuel Huber, and Julian Horsch. "Extracting Secrets from Encrypted Virtual Machines". In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX , USA, March 25-27, 2019*. 2019. DOI: 10.1145/3292006.3300022. URL: https://doi.org/10.1145/3292006.3300022.

[62] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. "SEVered: Subverting AMD's Virtual Machine Encryption". In: *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*. 2018. DOI: 10.1145/3193111.3193112. URL: https://doi.org/10.1145/3193111.3193112.

[63] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. "SEVerity: Code Injection Attacks against Encrypted Virtual Machines". In: *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*. 2021. DOI: 10.1109/SPW53761.2021.00063. URL: https://doi.org/10.1109/SPW53761.2021.00063.

[64] Krzysztof Okupski. *Exploring AMD Platform Secure Boot*. https://labs.ioactive.com/2024/02/exploring-amd-platform-secure-boot.html. 2024.

[65] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[66] Davi Pontes, Fernando Silva, Eduardo De Lucena Falcão, and Andrey Brito. "Attesting AMD SEV-SNP Virtual Machines with SPIRE". In: *12th Latin-American Symposium on Dependable and Secure Computing (LADC)*. 2023.

[67] Phillip Rogaway. "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC". In: *Proceedings of the 10th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. 2004.

[68] Gururaj Saileshwar, Prashant J. Nair, Prakash Ramrakhyani, Wendy Elsasser, José A. Joao, and Moinuddin K. Qureshi. "Morphable Counters: Enabling Compact Integrity Trees For Low-Overhead Secure Memories". In: *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. 2018. DOI: 10.1109/MICRO.2018.00041. URL: https://doi.org/10.1109/MICRO.2018.00041.

[69] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. "WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP". In: *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. 2024. DOI: 10.1109/SP54263.2024.00262. URL: https://doi.org/10.1109/SP54263.2024.00262.

[70] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. "HECKLER: Breaking Confidential VMs with Malicious Interrupts". In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/schl%20%5C%C3%5C%BCter.

[71] Jakub Szefer and Sebastian Biedermann. "Towards fast hardware memory integrity checking with skewed Merkle trees". In: *HASP 2014, Hardware and Architectural Support for Security and Privacy, Minneapolis, MN, USA, June 15, 2014*. 2014. DOI: 10.1145/2611765.2611774. URL: https://doi.org/10.1145/2611765.2611774.

[72] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. 2018. DOI: 10.1145/3173162.3177155. URL: https://doi.org/10.1145/3173162.3177155.

[73] Bashir M. Sabquat Bahar Talukder, Vineetha Menon, Biswajit Ray, Tempestt J. Neal, and Md. Tauhidur Rahman. "Towards the Avoidance of Counterfeit Memory: Identifying the DRAM Origin". In: *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*. 2020. DOI: 10.1109/HOST45689.2020.9300125. URL: https://doi.org/10.1109/HOST45689.2020.9300125.

[74] UPMEM. *UPMEM*. https://www.upmem.com. May 21, 2024.

[75] Saru Vig, Rohan Juneja, and Siew-Kei Lam. "DISSECT: Dynamic Skew-and-Split Tree for Memory Authentication". In: *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*. 2020. DOI:

10.23919/DATE48585.2020.9116548. URL: https://doi.org/10.23919/DATE48585.2020.9116548.

[76] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. "DRAMDig: A Knowledge-assisted Tool to Uncover DRAM Address Mapping". In: *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. 2020. DOI: 10.1109/DAC18072.2020.9218599. URL: https://doi.org/10.1109/DAC18072.2020.9218599.

[77] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. "The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*. 2019. DOI: 10.1145/3321705.3329820. URL: https://doi.org/10.1145/3321705.3329820.

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

[79] Luca Wilke and Gianluca Scopelliti. "SNPGuard: Remote Attestation of SEV-SNP VMs Using Open Source Tools". In: *IEEE European Symposium on Security and Privacy Workshops, EuroS&PW 2024, Vienna, Austria, July 8-12, 2024*. 2024. DOI: 10.1109/EUROSPW61312.2024.00026. URL: https://doi.org/10.1109/EuroSPW61312.2024.00026.

[80] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. "SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00080. URL: https://doi.org/10.1109/SP40000.2020.00080.

[81] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. "SEV-Step A Single-Stepping Framework for AMD-SEV". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.1 (2024), pp. 180–206. DOI: 10.46586/TCHES.V2024.I1.180-206. URL: https://doi.org/10.46586/tches.v2024.i1.180-206.

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

[83] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015. DOI: 10.1109/SP.2015.45. URL: https://doi.org/10.1109/SP.2015.45.

[84] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd M. Austin. "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors". In: *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*. 2017. DOI: 10.1109/HPCA.2017.10. URL: https://doi.org/10.1109/HPCA.2017.10.

[85] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. "CacheWarp: Software-based Fault Injection using Selective State Reset". In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-ruiyi.

Table 5: DIMM connections to interface with the SPD on DDR4 and DDR5.

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

Table 6: Bill of materials for the DDR4/DDR5 setup.

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

†Only for unlocking DDR4.

A AMD Response

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

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

Table 7: Write protection status and addressing information for various DDR4 modules.

			Write Protection				Addressing Bits				
Manufacturer	Type	Serial Number	WP0	WP1	WP2	WP3	Rank	Bank	Row	Col.	Capacity [GB]
Corsair_1	UDIMM	CMV4GX4M1A2133C15	✗	✗	–	–	0	4	15	10	4
Corsair_2	UDIMM	CMK16GX4M2E3200C16	✗	✗	✗	✗	0	4	16	10	8
Crucial_1	SODIMM	CT16G4SFD824A.M16FE	✓	✓	✓	✗	1	4	16	10	16
Kingston_1	RDIMM	KSM32RS8/8HDR	✓	✓	✗	✗	0	4	16	10	8
Kingston_2	RDIMM	KSM32RS8L/16MFR	✓	✓	✗	✗	0	4	17	10	16
Kingston_3	RDIMM	KTH-PL432/16G	✓	✓	✗	✗	0	4	17	10	16
Kingston_4	UDIMM	KVR26N19S8/8	✓	✓	–	–	0	4	16	10	8
Micron_1	RDIMM	MTA36ASF4G72PZ-3G2R1	✓	✓	✓	✗	1	4	17	10	32
Micron_2	RDIMM	MTA36ASF8G72PZ-3G2F1	✓	✓	✓	✗	1	4	18	10	64
Micron_3	RDIMM	MTA9ASF1G72PZ-3G2E2	✓	✓	–	–	0	4	16	10	8
Micron_4	RDIMM	MTA18ASF2G72PZ-2G9J3R	✓	✓	✓	✗	0	4	17	10	16
Samsung_1	UDIMM	M378A2K43DB1-CTD	✓	✓	✗	✗	1	4	16	10	16
Samsung_2	SODIMM	M471A2K43EB1-CWE	✓	✓	✗	✗	1	4	16	10	16
SK hynix_1	RDIMM	HMA82GR7DJR4N-XN	✓	✓	✗	✗	0	4	17	10	16
SK hynix_2	RDIMM	HMAA4GR7AJR8N-XN	✓	✓	–	–	1	4	17	10	32
SK hynix_3	UDIMM	HMA41GU6AFR8N-TF	✓	✓	✗	✗	1	4	15	10	8
SK hynix_4	UDIMM	HMA82GU6JJR8N-VK	✓	✓	✗	✗	1	4	16	10	16
SK hynix_5	SODIMM	HMA41GS6AFR8N-TF	✓	✓	✗	✗	1	4	15	10	8

Table 8: Write protection status and addressing information for various DDR5 modules.

			Write Protection		Addressing Bits				
Manufacturer	Type	Serial Number	MR12	MR13	Rank	Bank	Row	Col.	Capacity [GB]
Kingston_5	RDIMM	KF548R36RB-16	`0xff`	`0x3c`	0	5	16	10	16
Kingston_6	RDIMM	KSM48R40BS8KMM-16HMR	`0xff`	`0x00`	0	5	16	10	16
Kingston_7	RDIMM	KSM48R40BD8KMM-32HMR	`0xff`	`0x00`	1	5	16	10	32
Samsung_3	RDIMM	M321R2GA3BB6-CQK	`0xff`	`0x01`	0	5	16	10	16
SK hynix_6	RDIMM	HMCG78MEBRA115N	`0xff`	`0x01`	0	5	16	10	16
SK hynix_7	UDIMM	HMCG78AEBUA084N	`0xff`	`0x01`	0	5	16	10	16

Table 9: Write protection status and addressing information for the DDR3 module used in Section C.

					Addressing Bits				
Manufacturer	Type	Serial Number	SWP	PSWP	Rank	Bank	Row	Col.	Capacity [GB]
SK hynix_8	UDIMM	HMT351U6BFR8C-H9	–	✓	1	3	15	10	4

B SPD Setup

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

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

When connecting the addressing pins to ground, the EEPROM will be assigned I^2C peripheral address `0x50`. Any write protection on DDR4 can be cleared by connecting SA0 (pin 139) to V_{HV} (*i.e.*, 7–10 V) and issuing the Clear all Write Protection (CWP) command by writing to peripheral address `0x33` [44]. For DDR5, connecting HSA (pin 148) to ground allows modifications to be made to MR12 and MR13, the registers holding the protection status [45] (cf. Figure 3). In both cases, these changes are persistent.

C BadRAM attacks on DDR3

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

We evaluated the DDR3 BadRAM primitive on a Dell OptiPlex 990 DT with a CN-0VNP2H mainboard and an Intel Core i7-2600 with a single DDR3 UDIMM memory

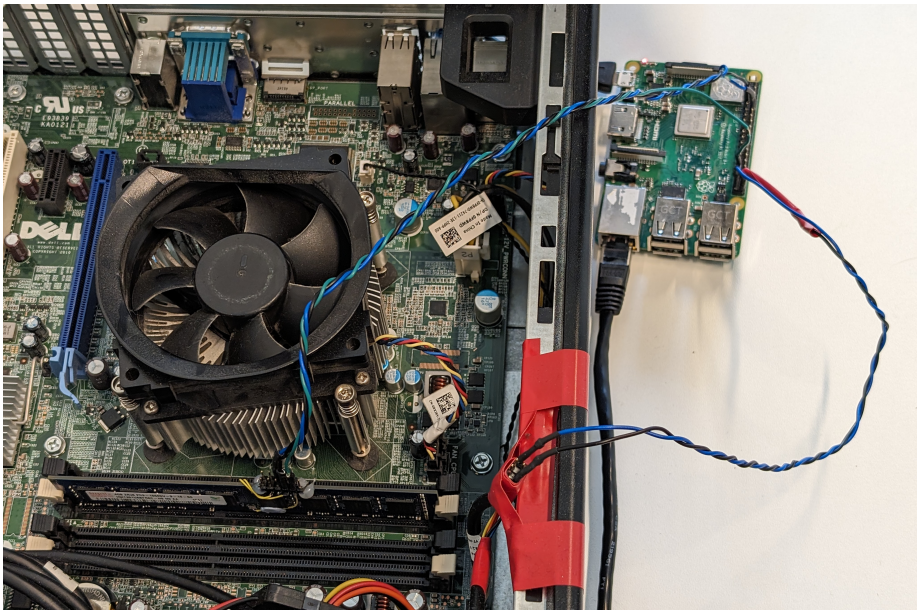


Figure 9: DDR3 setup to perform BadRAM attacks. The DDR3 DIMM has its SPD chip removed and is instead connected to a Raspberry Pi. The additional wires are connected to the power button, but are not required for the attack.

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

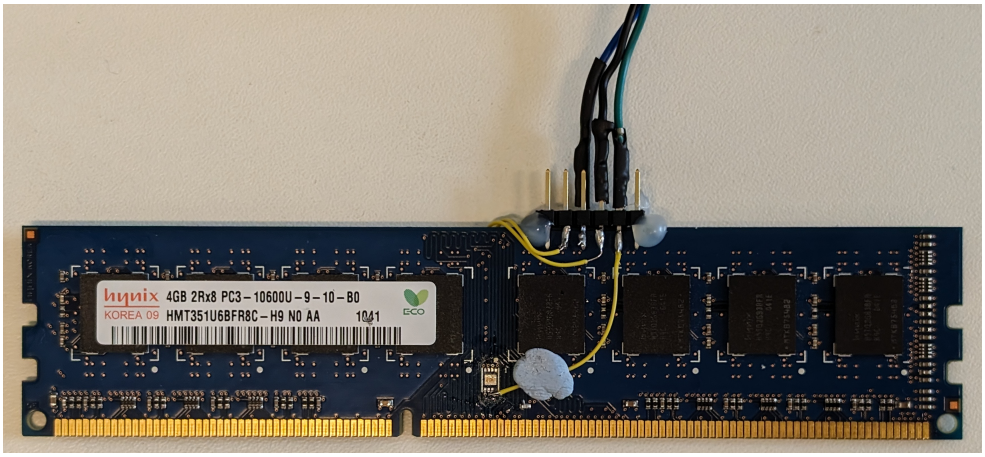


Figure 10: Closeup of the removed SPD chip from the DIMM in Figure 9. Note that some connections are made to the vias on the backside.

D Meta-Review

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

D.1 Summary

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

D.2 Scientific Contributions

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

D.3 Reasons for Acceptance

1. This paper identifies an impactful vulnerability. This paper demonstrates an architectural gap in AMD SEV-SNP that allows bypassing memory integrity protections for confidential VMs. The paper exploits the SPD interface available on DIMMs that allow re-programming of DIMM metadata (in-compliance with the JEDEC standard) to present the host SoC with a 'bad' view of memory that leads to memory aliasing. The memory aliasing is then used to manipulate the RMP table that holds security metadata for individual memory pages and thereby, bypass the protections affered by the RMP. The paper demonstrates and end-to-end attack where the measurement in the attestation doesn't match the measurement of the actual memory content of a confidential VM.

2. This paper provides a valuable step forward in an established field and independently confirms results with limited prior research. Memory aliasing attacks by manipulation of DIMM metadata have been known for a while. Intel SGX and TDX provide protections against such attacks through a proprietary set of checks (whose details are not public). This paper actually confirms that the checks implemented by these two technologies are effective against the attacks outlined in the paper which provides for the first time, independent verification of the security claims. The paper also explores for the first time, the SPD interface, requirements underlying the SPD interface that allow reprogramming of DIMM metadata by the JEDEC standard as well as effects of manipulating the DIMM metadata via the SPD interface. Since there are no existing mechanisms/standardized ways to protect this DIMM metadata, the identified vulnerabilities will need a workaround (just like TDX and SGX do). The paper also demonstrates the viability of affecting the security posture of a TEE whose trust boundary is confined to the SoC via corruption of a system/platform component that is likely vulnerable to supply chain attacks.

3. This paper establishes a new research direction. Existing countermeasures to the outlined attacks have been proprietary (and emerging ones will likely also be so) to detect memory aliasing. Changing DIMM standards to include more systematic countermeasures will likely have a long tail. So, this paper highlights the need to devise mechanisms that can work with existing DIMM standards in the public domain—ones that lend themselves to systematic analysis instead of a heuristic that relies on randomly checking for aliases but still scale for use in cloud settings.

4. The paper creates a new tool for future science: The authors are committed to making their attack framework available for other researchers to explore other offensive and defensive countermeasures.