Heracles: Chosen Plaintext Attack on AMD SEV-SNP

Benedict Schlüter
benedict.schlueter@inf.ethz.ch
ETH Zurich
Switzerland

Christoph Wech
chwech@student.ethz.ch
ETH Zurich
Switzerland

Shweta Shinde
shweta.shinde@inf.ethz.ch
ETH Zurich
Switzerland

Abstract

Confidential computing needs hardware support that stops privileged software from learning secrets of a guest virtual machine. AMD offers such hardware support in the form of SEV-SNP to create confidential virtual machines, such that hardware encrypts all the VM memory. Specifically, SEV-SNP uses the XEX encryption mode with address-dependent tweak values such that the same plaintext at different memory addresses yields different ciphertexts.

Heracles makes three observations: the hypervisor can move encrypted guest pages in DRAM using three APIs; when it moves the guest pages to a new DRAM address, pages are re-encrypted; re-encryption is deterministic. By re-encrypting guest data at precisely chosen DRAM locations, we can create a chosen plaintext oracle allowing us to leak guest memory at block granularity. We build four primitives that leverage the victim's access patterns to amplify Heracles's impact to not only leak data at block but at byte granularity. In our case studies, we leak kernel memory, crypto keys, and user passwords, as well as demonstrate web session hijacking.

CCS Concepts

• Security and privacy → Hardware attacks and countermeasures.

Keywords

SEV-SNP, confidential computing, ciphertext attack, virtualization

ACM Reference Format:
Benedict Schlüter, Christoph Wech, and Shweta Shinde. 2025. Heracles: Chosen Plaintext Attack on AMD SEV-SNP. In . ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Confidential computing has been adopted in several use-cases, especially cloud deployments [1, 2, 18, 36]. Departing from Intel SGX enclave model for sub-process isolation [24], modern deployments have shifted to a broader abstraction. Recent and upcoming hardware support for trusted execution environments that support confidential computing offer a confidential virtual machine (CVM) as the boundary of isolation. This is apt for cloud scenarios where customers can lift-and-shift their workloads from classic VMs to confidential VMs. Not only does this enforce hardware isolation, it shields the user workloads and data from a buggy or even a malicious hypervisor.

AMD has been offering support for such CVMs in the form of SEV [8], which has been followed up with subsequent enhancements. AMD SEV, which stands for Secure Encrypted Virtualization, initially performed memory encryption for VM memory to protect it from untrusted privileged software. It was followed by SEV-ES, which added encryption of guest state to enforce confidentiality [29], hence the suffix ES, which stands for Encrypted State. The latest iteration, called AMD SEV-SNP, further enforces integrity by ensuring that the hypervisor cannot tamper with the page mappings, hence the suffix SNP for Secure Nested Paging [3].

Both SEV-ES and SEV-SNP encrypt the guest, i.e., CVM memory state, including registers and physical memory page content. This is also the case for past, contemporary, and future TEEs, namely Intel SGX [24], Intel TDX [26], and Arm CCA [9], respectively. AMD departs in one notable way—it allows the hypervisor to read the encrypted state of the CVM memory. This attacker capability, combined with the underlying cryptographic scheme (AES-XEX), has been shown to be vulnerable to ciphertext side-channel leakage [31, 34]. In particular, the attacker can observe changes in the ciphertext, or lack thereof, to infer the CVM's execution state (e.g., control variables). In other words, repeating plaintext that leads to ciphertext collisions leaks information. Recent works have applied these insights to neural networks that exhibit execution patterns that are prone to both model weight and input leakage [59, 60]. Clearly, allowing the hypervisor to read the ciphertext of a victim's memory during execution leaks information. However, the adversary has limited control; the ciphertext changes only when the memory content changes naturally during execution.

In this paper, we present a new attack called Heracles. Our observation is rooted in: (a) the hypervisor's capability to move the victim CVM pages in DRAM, such that it knows the source physical address and controls the destination physical address; (b) this moving of pages results in a re-encryption; (c) the AES-XEX [43] scheme uses physical addresses for generating tweak values; (d) the victim CVM takes in hypervisor-controlled data through several APIs. With our ability to move pages, coupled with AMD's tweak scheme, we construct a chosen plaintext oracle: the attacker can provide several known plaintext values and observe the resulting ciphertext. To exploit this, we construct a dictionary of plaintext-ciphertext pairs in the preparation step. During the attack, we observe the victim's ciphertext and use the dictionary to extract the corresponding plaintext. This departs from prior ciphertext-side channel attacks that focused on passively observing leakage. Instead, we actively move pages in the memory to trigger re-encryption strategically, observe the ciphertext, and use the oracle to correctly guess the plaintext.

Practically, the encryption granularity is 16 bytes, which leads to a dictionary of size 2^{128}. If we assume a priori knowledge of the victim data (e.g., ASCII values for passwords), it can reduce the search space. However, for data that does not adhere to these

This is the author's version of the work.
CCS'25, October 2025, Taipei, Taiwan
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

distributions (e.g., cryptographic keys), the worst-case complexity is high. To this end, our second contribution makes HERACLES practical by avoiding brute-force. We build four primitives that are based on the victim's execution that results in: changing data at a byte granularity, changes to data in-place, either due to unique or repeated values, and attacker-controlled padding boundaries. Thus, we start by inspiration from prior works on ciphertext-based side-channels that target in-place changes to data that has repeated values, and build three new primitives that are specific to HERACLES. We show that these strategies lead to a significant reduction in the search space and make HERACLES practical.

We analyze the AMD SEV-SNP hypervisor interface and find three ways the hypervisor can move pages in memory. Next, we build a chosen plaintext oracle by bringing attacker-controlled plaintext into victim memory, swapping pages, and comparing ciphertexts. Lastly, we showcase five victim applications where we apply the four primitives. Notably, we exploit the `rep movsb` construct often used in memory copies and force a byte-by-byte copy such that we can use HERACLES. We recover passwords used in sudo and bash, cryptographic keys in mbedtls, and cookies from a web server.

Mitigating HERACLES requires disabling at least one of the two hypervisor capabilities: (1) moving pages in DRAM; (2) reading CVM ciphertext. Although AMD can remove hypervisor access to interfaces that move pages, it also hinders the hypervisor from doing efficient resource management (e.g., dynamic VM scaling). However, this only requires a firmware update and can be rolled-out for existing hardware in deployment, thus addressing (1). For addressing (2), unfortunately, prior defenses are not effective gainst HERACLES [28, 41, 54, 55]. They do not stop the hypervisor from reading the ciphertext; they only address specific ciphertext side-channel leakage. On the other hand, revoking the hypervisor's access to ciphertext to directly address (2) can not only stop HERACLES but also all prior ciphertext side-channels on AMD SEV-SNP. In fact, AMD has announced a feature for 5th Generation AMD EPYC Processors codenamed Turin with Zen 5, where the hardware can limit visibility of CVM ciphertext to the hypervisor [7]. We cannot estimate the performance impact of this feature because we were not able to enable it on our Zen 5 CPUs, due to a lack of hardware, firmware, and/or BIOS support. It remains to be seen if the trade-off between removing hypervisors ability to efficiently manage guest memory is worth it, or if the performance overheads of the hardware support are modest.

To summarize our key contributions in HERACLES:

- Chosen Plaintext Oracle. We use the hypervisor's ability to move CVM memory in DRAM and read ciphertexts to build a chosen plaintext oracle.
- Primitives. To showcase the practicality of the chosen plaintext oracle beyond brute force, we devise four primitives that leverage the victim's data patterns to reduce the search space.
- Impact on real applications. We show that HERACLES is feasible on real-world applications, such as leaking data from memory copies, authentication passwords in sudo and bash, cryptographic keys in mbedtls, and session cookies for a web server.

Responsible Disclosure. We reported our findings to AMD in January 2025. AMD acknowledged the vulnerability and asked for a coordinated disclosure with an embargo date. Additionally, AMD informed us that they will release a SEV-SNP specification and firmware update that restricts the ability to move pages. They clarified that this is a feature update, and not a mitigation in response to our disclosure, that limits HERACLES. This feature was made public in May 2025 (SEV-SNP ABI version 1.58 [7]).

Artifacts. To support open science, our code is public at https://heracles-attack.github.io/

2 Overview

Isolating virtual machines from the hypervisor has gained traction in recent years, sparking the term Confidential VMs (CVMs). CVMs safeguard user data and code against an untrusted hypervisor.

AMD Secure Encrypted Virtualization with Secure Nested Paging (SEV-SNP). AMD SEV-SNP lets users run entire VMs within a trusted execution environment. AMD hardware allows the hypervisor to read the encrypted memory and register state of a CVM. Furthermore, the hypervisor remains responsible for CVM scheduling and memory management. For the latter, the hypervisor can move encrypted CVM memory in DRAM—a highly privileged operation. Thus, AMD does not allow the hypervisor to perform this operation itself but asks a privileged co-processor to perform the memory movement.

Memory Encryption. SEV encrypts guest memory with a unique per-CVM encryption key to guarantee confidentiality. It employs AES as the encryption algorithm, offering a key size of either 128 or 256 bits. AES organizes the plaintext into 16-byte blocks and encrypts each one separately. To strengthen SEV against known attacks, AMD adopts an encryption mode called XOR-Encrypt-XOR (XEX). XEX uses tweak values derived from the DRAM address used in the memory access. Tweak values are XORed ($\oplus$) with the plaintext (before encryption) and the ciphertext (after encryption). For decryption, the same applies but in reverse. Tweak values ensure identical plaintext pairs at different memory locations produce distinct ciphertexts [43] as shown in Figure 1.

2.1 Ciphertext Attacks

On AMD SEV-SNP, attackers can exploit the availability of ciphertexts to deduce information about plaintexts. XEX mitigates these attacks to some extent by encrypting identical plaintexts at different memory locations into distinct ciphertexts. The XEX encryption mode prevents the attacker from comparing ciphertexts across different memory locations. As each physical address results in a unique tweak value, it produces different ciphertexts even for the same plaintext data. However, when XEX mode encrypts the same plaintext value at the same physical address, it generates identical ciphertext values as shown in Figure 2. Researchers have widely exploited this lack of encryption freshness. Specifically, prior works primarily leverage the availability of ciphertext as a side channel to uncover confidential information [31, 34, 59–61].

Cipherleaks [34], the first ciphertext-based side-channel against SEV-SNP, targeted the CVMs register state that was constantly written to the same memory location. The attacker could compile a dictionary of register values and their matching ciphertexts to

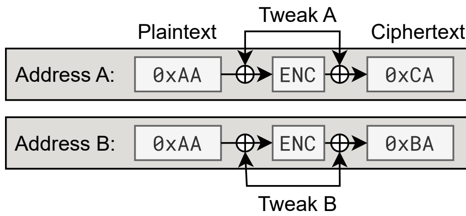

Figure 1: SEV-SNPs memory encryption with address-based tweak values. The same plaintext at different memory locations encrypts to different ciphertexts. The tweak value limits the impact of ciphertext-based attacks.

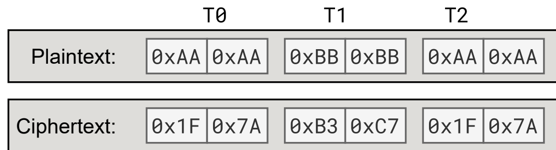

Figure 2: The same plaintext at the same memory location but different timestamps encrypts to the same ciphertext. An attacker monitoring ciphertexts infers that the plaintexts at T0 and T2 are equal.

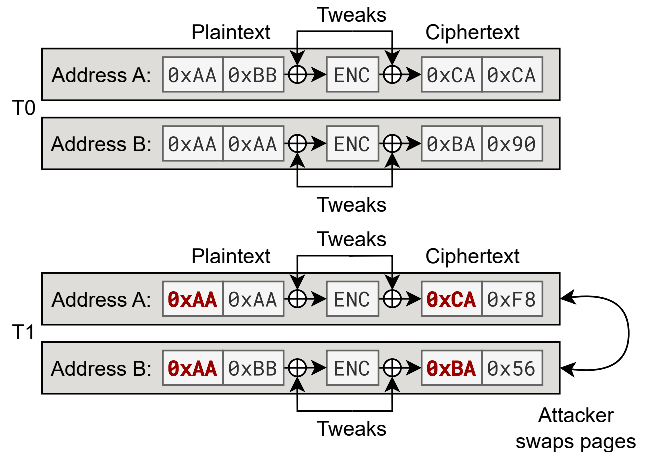

Figure 3: T0: Attacker-controlled data resides on Page A and secret data on Page B. The tweak value differs for Page A and Page B, resulting in two fully different ciphertexts. T1: Attacker swaps Page A and Page B. The re-encrypted plaintexts leak partial information about Page A's content. The Attacker infers the first byte of the secret is `0xAA` because the ciphertexts match. This simplified example assumes a block size of 1 byte, compared to 16 bytes for AES.

exploit the deterministic encryption of the register state. Figure 2 shows how the same value written to the same address at different times (T0 and T2) encrypts to the same ciphertext. Shortly before CVM suspension, the microcode writes the register state always to the same address. If the attacker interrupts a CVM at two different timestamps and the register state ciphertexts match, the attacker infers that the registers contain the same plaintext. From there, an attacker may extract sensitive data from registers and compromise various cryptographic implementations. Cipherleaks only targets the memory containing the register state. AMD mitigated the vulnerability by adding freshness to the encryption of the register state. During every register state write, a random microcode-derived value is XORed with all registers before the microcode writes the registers to memory. The added freshness results in distinct ciphertexts despite encrypting the same plaintext.

Other works exploit the same flaw but focus on general guest memory content (e.g., variables changing during program execution) rather than register state [31, 59, 60]. Thus, all prior attacks exploit repeating plaintext that leads to ciphertext collisions, where the success depends on the application. Due to its root-cause coupled with the attacker's limited control, prior SEV-SNP ciphertext-based attack can only extract repeating values from a CVM.

2.2 Heracles Attack

Heracles leaks non-repeating values, based on three observations.

(1) The hypervisor can move guest pages in memory.
(2) Moving memory causes the memory controller to re-encrypt the moved memory with XEX $\oplus$ tweak values corresponding to the new DRAM address.
(3) XEX tweak values are constant through the CVM lifecycle.

Crucially, our first observation allows the attacker to expand its control beyond prior works. Combining all 3 observations, the hypervisor can build a chosen plaintext oracle to infer secret non-repeating CVM data.

Figure 3 shows how the attacker swaps Page A and Page B of the CVM and learns partial information about the plaintext stored on these pages. Specifically, the swapping causes the page content to be re-encrypted with the same key but different tweak values (T1 in Figure 3). By comparing the ciphertexts, the hypervisor infers that the first plaintext bytes of the swapped pages are identical since the first byte of the ciphertext is identical after the swap. However, the fact that Page A and Page B share the same first byte does not leak any information about the byte content itself to the hypervisor. Thus, only relying on plaintext values already present in the CVM leaks information in a limited fashion. Next, we explain how Heracles uses this observation to amplify the leakage.

Building Chosen Plaintext Oracle. A Chosen Plaintext Oracle lets an adversary create ciphertexts from chosen plaintexts. In the Figure 3 example, we can build such an oracle if the attacker controls Page B's content. Since we have hypervisor privileges, Heracles can use various CVM I/O interfaces to inject plaintext data into the CVM. Now at T1 in Figure 3, when the attacker swaps the victim page A with the attacker-controlled page B, by observing the ciphertext, it infers whether the guessed plaintext was identical to the plaintext on the victim page. If the ciphertexts do not match, the attacker repeats until the guess is correct. Put together, Heracles can leak CVM memory content, even if it contains values that are not already present or not repeated in the CVM.

Challenge in using Chosen Plaintext Oracle (CPO) via Brute-force. The CPO, producing ciphertext for AES, takes a 16-byte value

as input and outputs a 16-byte ciphertext. Thus, guessing arbitrary data with HERACLES requires bruteforcing all 16 bytes. Without any prior knowledge, HERACLES would need to exhaustively test on average 2^{127} distinct plaintexts through the CPO. While a priori knowledge about the plaintext under attack reduces the search space significantly, keys and other high-entropy data are random and do not offer a reduction in the search space. If we know the location of the key, we have to brute-force multiple 16-byte blocks depending on the key length. Although this brute-force approach remains theoretically feasible, its practicality is impossible due to the extensive computational resources required. Consequently, we introduce this solely as a theoretical primitive rather than a primitive usable to attack real-world systems. More importantly, this motivates the need to build better attack strategies to demonstrate that HERACLES is practical.

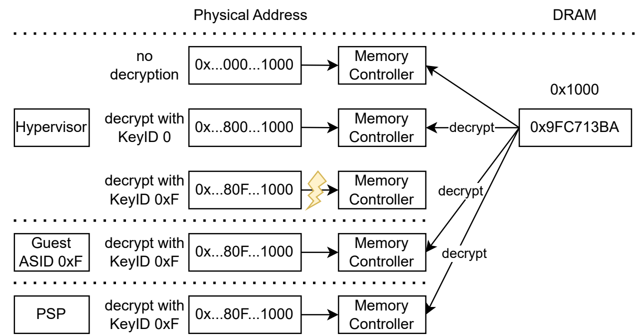

Figure 4: Influence of the physical address on the decryption/encryption operation of the memory controller

3 AMD Platform

This section introduces the relevant AMD platform components for HERACLES. We explain the details of AMD's memory encryption and how the memory controller selects the correct keys.

3.1 SEV-SNP Memory Encryption

SEV-SNP encrypts all guest memory in DRAM. AMD processors with SEV-SNP enabled may choose between AES-128 and AES-256 for memory encryption. AES encrypts blocks of 16 bytes individually. AMD uses XOR-Encrypt-XOR (XEX) as the mode of operation for the encryption algorithm. XEX does not chain ciphertext blocks, meaning each 16-byte block may be encrypted and decrypted individually. Being able to encrypt and decrypt 16-byte blocks individually has performance advantages but comes at the cost of losing some security properties e.g., compared to GCM encryption mode. XEX uses tweak values to mask the plaintext before and the ciphertext after encryption. The memory controller masks the plaintext by XORing the tweak value to the plaintext, analogously for the ciphertext. Each DRAM address has a different tweak value. Tweak values ensure the same plaintext encrypts to different ciphertexts depending on the memory location. SEV-SNP does not refresh the tweak values, which remain static for the lifecycle of SEV-SNP guests. This results in the same plaintext being written as the same ciphertext at the same memory location. The memory controller automatically decrypts and encrypts data on memory reads and writes.

3.2 Memory Controller

The memory controllers interface the CPU cores and the physical DRAM. A trusted co-processor (PSP, detailed in Section 4.1) programs the encryption keys into the memory controller hardware using a proprietary management protocol [4]. The PSP assigns a unique encryption key to each SEV-SNP VM, freshly generated on CVM creation. SEV-SNP uses the upper physical address bits to propagate key information through the bus to the memory controller. Figure 4 shows different combinations of the propagation. Upon receiving a memory request, the memory controller checks whether the destination address has the C-bit set (usually bit 51). The C-Bit indicates that the data attached to the memory address should be encrypted for memory write requests. For read requests, it indicates that the memory controller must decrypt the fetched data before sending it to the cores. If the C-bit is set, the memory controller uses the higher bits of the physical address, that represent the Address Space Identifier (ASID), as an index into the data structure that stores the encryption keys [4].

Next to the encryption key, the memory controller generates two tweak values based on the memory address. The first tweak value is XOR'ed to the plaintext before encryption, and the second is XOR'ed to the ciphertext after encryption. For decryption, the same operation applies but in reverse. Wilke et al. reverse-engineered the tweak values and discovered that it only provide 4 bytes of entropy [56]. Furthermore, they showed that both the tweak values used in one encryption are identical. Notably, the key index that the memory controller uses to select the corresponding key equals ASID of the SEV-SNP CVM. Thus, ASIDs must be unique to prevent key collusion. ASID 0 is exclusively reserved for hypervisor usage. Depending on the system configuration, the memory controllers may also encrypt ASID 0 or treat it as a special case. The security of SEV-SNP relies on the fact that the hypervisor cannot demand the memory controller to decrypt memory with a guest key.

4 Page Moving APIs

Moving encrypted pages in DRAM allows the hypervisor to select the tweak values used to encrypt the page. We systematically analyze all publicly available SEV-SNP documents and code to identify all the interfaces that a hypervisor can use to move encrypted guest pages in memory. We found two different privileged entities capable of moving pages, exposing in total three APIs to the untrusted hypervisor.

4.1 Platform Security Processor

One of the privileged entities is the Platform Security Processor (PSP). The PSP, also known as the AMD Security Processor (ASP) or just Security Processor (SP), enforces the security semantics of SEV-SNP. As the highest privilege component on the platform, the PSP generates and programs the encryption keys into the memory controller. It plays a vital role in SEV-SNP initialization and guest bootstrapping (i.e., ensuring CVM ASIDs are unique). The PSP captures the initial attestation value of a SEV-SNP CVM during boot. Besides the guest boot, the PSP also plays an important role

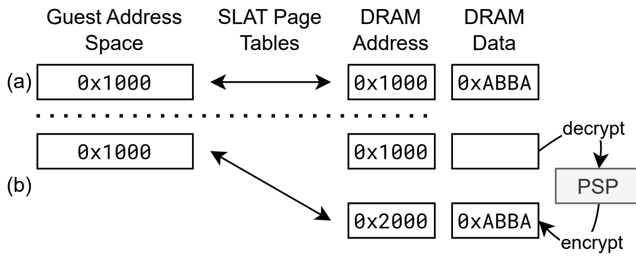

Figure 5: (a) Initial Memory layout. (b) Memory layout after PSP has moved encrypted guest data from DRAM address 0x1000 to 0x2000. The DRAM data shows the plaintext that the CVM sees when reading the memory.

in managing the lifecycle of a SEV-SNP CVM. It exposes interfaces to the hypervisor and for SEV-SNP guests. The hypervisor uses a set of MMIO registers to request services from the PSP [7]. Two of these services move encrypted guest pages in DRAM. The hypervisor is not capable of directly moving pages as it would violate security guarantees offered by SEV-SNP. To grant the hypervisor the flexibility to reorganize the memory layout, AMD offers it an API, wherein the PSP—a higher privileged entity—actually executes the move operation as shown in Figure 5 (b).

4.1.1 Move API. The function `SNP_PAGE_MOVE` moves encrypted guest pages in DRAM. It takes as input a source and a destination address. The source and destination are subject to a variety of security checks. The hypervisor must put these pages in a locked state called Pre-Guest. The locking ensures that neither the hypervisor nor the guest can write to the pages. Locking is necessary to prevent concurrent access to the DRAM since the PSP cannot move an entire page atomically. Figure 3 shows how the hypervisor may swap the content of two pages using the PSP move API.

Specifically, `SNP_PAGE_MOVE` takes a guest memory source page and a hypervisor destination page. Upon request and after passing all security checks, the PSP copies the content of the source page to its internal buffer and writes it to the destination page. The PSP sets the guest ASID to the CVM ID to decrypt the page content. The same applies when the PSP writes the content to the destination page. The memory controller automatically selects the correct key and tweak values for the source and destination pages. If successful, the PSP marks the destination page as guest memory and the source page as hypervisor-owned.

The API allows Heracles to encrypt a given page in guest memory with the tweak value of a hypervisor-chosen destination page. Moving pages is fully transparent to the guest, meaning the guest has no architectural way of detecting that the DRAM address of its data has been changed. This makes it especially interesting for attacks since the guest cannot detect if pages have been moved or if the hypervisor moves the pages at the correct time, e.g., when the guest is not scheduled.

4.1.2 Swap API. The functions `SNP_PAGE_SWAP_OUT` and `SNP_PAGE_SWAP_IN` allow moving encrypted guest pages to disk and back to memory. `SNP_PAGE_SWAP_OUT` takes a guest memory source page and a hypervisor destination page. The guest memory is encrypted with a special swap key and written to the hypervisor page. Simultaneously, the PSP creates a metadata entry on another page containing the hash value of the memory as well as other security-relevant information. Importantly, the metadata entry does not contain the DRAM location of the page. When swapping the page in with `SNP_PAGE_SWAP_IN`, the hypervisor may choose whichever DRAM location it likes as long as the hypervisor is the owner of that memory region. By combining swap-out with swap-in, we can move an encrypted guest page within the DRAM and effectively control the tweak values. It achieves the same effect as `SNP_PAGE_MOVE` but with one more API call.

Table 1: Attack capability overview of different ciphertext-based attacks on SEV-SNP. ● – Attack can leak data pattern; ○ – Attack cannot leak data pattern; ◐ – Attack can leak data theoretically but not practically.

Primitive	Heracles	Cipherleaks [34] Li et al. [31]	Ciphersteal [60]	Hypertheft [59]
P0: Bruteforce	◐	○	○	○
P1: Byte-By-Byte	●	○	○	○
P2: Chosen Boundary	●	○	○	○
P3: In-Place (repeat)	●	●	●	●
P4: In-Place (unique)	●	○	○	○

4.2 DMA Engine

Next to the PSP, AMD EPYC processors include another trusted internal component capable of moving encrypted guest pages. Public documentation reveals a powerful internal Direct Memory Access (DMA) engine named MPDMA [5]. AMD introduced MPDMA to move large amounts of (encrypted guest) memory between two different memory backends (i.e., DRAM / CXL memory). Its API is likely much faster compared to the PSP since one command may move up to 128 pages at once. Furthermore, unlike the PSP, it takes a list of up to 128 pages to be moved. MPDMA supports three different commands, one to move normal unencrypted memory and another one to move encrypted memory. The third command is a nop command and exists solely for testing. MPDMA is a passive component with respect to guest memory. It may change the physical location where guest memory resides, but it does not actively change its value or state. Thus, all MPDMA operations are transparent to the guest, just like the previously introduced PSP operations. While AMD documents the MPDMA engine, we cannot use it as AMD disables it by default, and there is no public documentation on how to enable it. Checking the status registers from the documentation shows the registers are either not set or the MPDMA is disabled [5].

5 Heracles Primitive

We expand on the details of Heracles attack summarized in Section 2.2. Section 5.1 describes how Heracles builds the Chosen Plaintext Oracle (CPO). In Section 5.2–5.4, we explain our three attack strategies to overcome the need for bruteforce. Table 1 compares Heracles attack capabilities against previous ciphertext vulnerabilities.

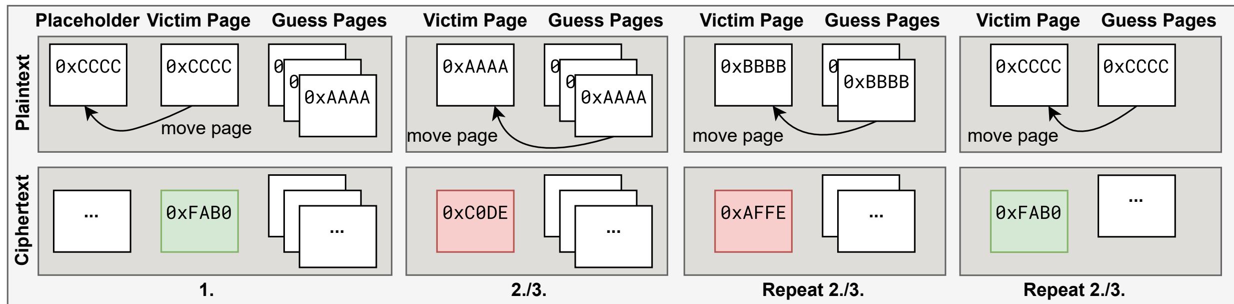

Figure 6: The Victim Page is the physical page where the CVM data under attack resides. The Guess Pages hold the data the attacker injects into the CVM. 1. The attacker moves the victim page to a temporary page. 2./3. The attacker swaps the Guess Pages to the previous DRAM address of the Victim Page and compares the resulting ciphertext with the one from the previously snapshot and repeats 2./3. until the ciphertexts match.

5.1 Steps in Building CPO

Heracles builds a CPO to break SEV-SNPs' confidentiality. This subsection explains the three steps involved in leaking 1 byte of unknown guest memory. An attacker may repeat the steps to leak an arbitrary amount of data. It can use the CPO to guess data of up to the used block length of 16 bytes. Trivially bruteforcing 128 bits is the first primitive **P0** that Heracles introduces. However, using the CPO to brute force 16 bytes is computationally infeasible. If an adversary uses a CPO to guess up to 8 bits (1 byte), the number of CPO queries does not exceed $2^8 = 256$ in the worst case. Compared to 2^{64} queries for a 64-bit value or 2^{128} queries for a 16 byte block. Thus, we consciously limit Heracles's CPO to guess only up to 1 byte at a time. As we show in the next section, even with this restriction, we can leak almost all memory moved in a CVM. Figure 6 gives a high-level overview of the attack.

Finding sensitive data in memory. As a necessity, an adversary must know where the sensitive data resides in the guest memory. Nearly all previous SEV attacks have proposed techniques to find pages of interest [17, 31, 39, 42, 45, 46, 56]. All prior attacks use second-level address translation (SLAT) page faults to trace the execution state of a CVM. The hypervisor controls the SLAT page tables and can set bits such as the Non-Executable and Present bit. The corresponding guest access causes a page fault, revealing that the guest is trying to execute code on a page. To fine-tune the results, techniques may use performance counters [17, 31] or resort to single-stepping [57]. Heracles uses previous page-fault and single-stepping techniques to precisely find pages of interest.

Step 1: Getting attacker-controlled plaintext into victim memory. SEV and SEV-ES attacks inject attacker-controlled data in the CVM [33, 42, 56]. As with finding pages in memory, prior work has extensively studied CVM I/O interfaces and the possibility of injecting attacker-controlled plaintext. Heracles uses ICMP packets to inject data into the CVM. Compared to prior work [39], this has the advantage that we control a full page (0x1000 bytes) rather than just a small portion of a buffer. Only controlling a subsection of a page may work for specialized cases where the page alignment of the secret data and the subsection align. For the CPO to succeed, the attack-controlled data must have the exact page offset as the target data. By controlling a 0x1000 page, an attacker can fill the entire page with the guessed 16-byte blocks and eliminate the need to precisely inject data at the correct offset.

Step 2: Swapping Pages. To swap pages, we can use one of the three APIs introduced in Section 4. We resort to only using the `SNP_PAGE_MOVE` API. To swap the pages efficiently, we need a third page for temporary storage. Figure 6 visualizes the swapping process in memory. We snapshot the ciphertext on the page containing the secrets. Subsequently, we use the SNP API and move the secret page to another DRAM location. The DRAM location with the snapshotted ciphertext tweak values is now unoccupied. In the next step, we copy the guest memory page containing the plaintext we injected into the free DRAM slot.

Step 3: Comparing Ciphertexts. As a last step, we compare the ciphertext of the newly moved page with the snapshot. If they match, we know our guess is correct. Otherwise, we go back to Step 1 and repeat until we are successful. For each round, we need to invoke `SNP_PAGE_MOVE` twice. First to move the old page from the DRAM location associated with the tweak values of interest, and second to move the page containing the new data to that DRAM location.

5.2 P1: Byte-By-Byte

Heracles leaks data that changes byte-by-byte, as its most impactful attack primitive. As the name indicates, the data must be moved or changed in a byte-by-byte manner, and an attacker needs a priori knowledge about the initial memory content.

Figure 7 shows how a victim copies the data from a source address byte-by-byte to the destination memory address. An adversary interrupts the CVM between two subsequent byte copies and executes the Heracles attack as described in Section 5.1. The adversary must ensure to leak the data in consecutive order since the bytes of previous leaks are needed to construct the oracle plaintext for the following bytes. Missing individual bytes increases the computational complexity of guessing the next byte. If the attacker misses one byte, a subsequent guess must test $2^8 \cdot 2^8 = 2^{16}$ plaintext; for two missed bytes, the complexity increases to 2^{24}, and so on. Leaking the byte at position 4 in Figure 7 efficiently (i.e., only needing to test 2^8 plaintexts) requires all previous bytes to be known.

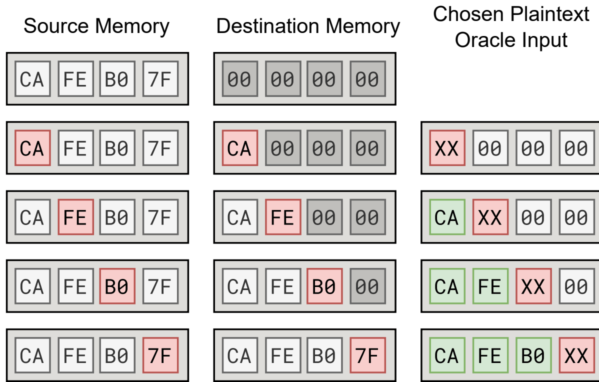

Figure 7: Byte-By-Byte memory copy from source to destination. Between individual byte copies, an adversary executes HERACLES to leak the byte marked in red. The green bytes are known to the attacker.

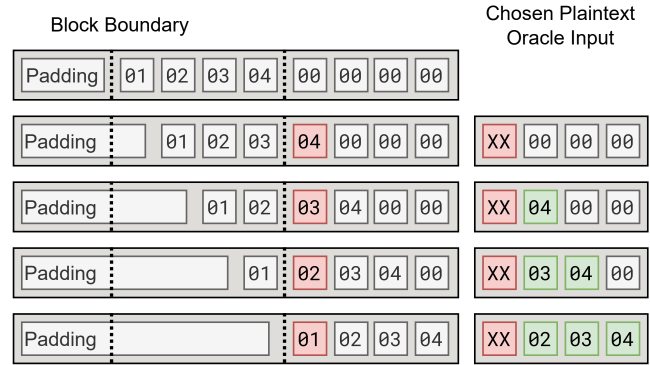

Figure 8: An adversary controls the length of the padding. By controlling the length, he can offset the secret value to cross block boundaries. Offsetting the secret value byte-by-byte allows HERACLES to guess the secret byte-by-byte rather than guessing 4 bytes at one.

We observe byte-by-byte changing data in a variety of applications. Linux `memcpy` resorts to moving data in a byte-by-byte pattern. Furthermore, an occurrence of byte-by-byte changing data includes all kinds of user inputs. A user only inputs one character at a time, and programs process the user input character-by-character (e.g., Bash and Sudo). Most notably, user inputs also include passwords as they are entered by the user. Contrary to P0, we can significantly reduce the search space to guess keys copied by `memcpy`. Instead of brute forcing the entire 16-byte block, we brute force each 1-byte value individually. Brute forcing the key byte-by-byte reduces the search space from 2^{128} to $2^8 \cdot 16$. As we show in the following sections, guessing $2^8 \cdot 16$ values is computationally feasible in real-world systems.

5.3 P2: Chosen Boundary

Chosen boundary data is data whose block position is affected by (malicious) user input or other external state. Depending on the input, the blockwise offset of the chosen boundary data changes in memory. An example is a format string taking a string as an argument. Based on the length of the string argument, all data following the input will be offset in memory.

```c
printf("User %s entered %d\n", user_var, input);
```

Listing 1: Simple Format String in C

Listing 1 shows an example. The content of *user_var* offsets every character following its position in the format string, depending on the length. An empty string in *user_var* does not offset the following characters, whereas a string of length 5 offsets the following characters by 5 in memory.

HERACLES leaks chosen boundary data if an attacker can induce changes to the offsets on a byte-by-byte basis. Figure 8 shows an example of an attacker using a chosen boundary privilege. The attacker controls the padding in front of a secret value. Subsequently, the attacker increases the length of the padding (e.g., by sending network requests with more data) and pushes the secret byte-by-byte in a new block. After each increment of the padding by one byte, the attacker uses HERACLES to leak the secret 1-byte value newly pushed into the new block. The attacker continues until the entire secret has been successfully leaked

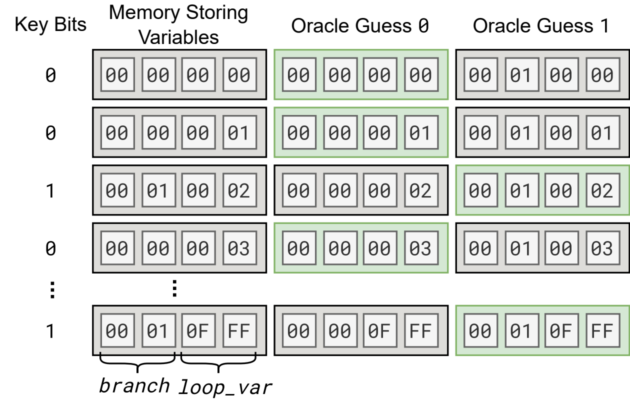

Figure 9: Positioning of Variable *branch* and *loop_var* from Listing 2 in memory. An adversary guesses 0 or 1 for the key using HERACLES's CPO. The green boxes show a ciphertext match with the loop variables, which leaks the key.

```c
for (int loop_var = 0; loop_var <= 255; loop_var++){
    int branch = key_bits[loop_var];
    do_square();
    if(branch)
        do_mul();
}
```

Listing 2: Simplified Square and Multiply Algorithm

5.4 P3-P4: In-Place

Some values change in-place (e.g., counter, branch variable). Rather than being copied from one memory location to another, like byte-by-byte changing data, the values remain static at one location. We distinguish between two types of in-place changing data.

P3: Repeating Values. Repeating values will take the same value over and over again throughout program execution. This can be a variable indicating whether a conditional branch should be taken or not (e.g., *branch* variable in Listing 2). An adversary may trace a change in the branch variable by observing the ciphertext if the variable is the only changing data in an encryption block (16 bytes for AES). Even without Heracles, an attacker may easily leak patterns by observing changes in the ciphertext and subsequently building a dictionary of plaintext-ciphertext pairs [31, 34].

P4: Unique Values. Unique values also change in-place but do not repeat during program execution (e.g., non-overflowing counters). Real-world applications may have repeating values grouped with unique values. A monotonously incrementing counter in a plaintext block will never have the same value throughout the lifecycle of a CVM, resulting in distinct ciphertext values. Listing 2 shows a code example where unique and repeating values are naturally grouped in the same encryption block. The compiler aligns *loop_var* and *branch* such that they are in the same 16-byte encryption block (Figure 9 shows a simplified version where the block size is reduced to 4 bytes). *loop_var* is incrementing throughout the loop, and thus, the 16-byte block will never have the same value despite *branch* taking the same two values over and over. Despite the ciphertext never repeating, Heracles is still capable of leaking the secret key. Figure 9 visualizes how an attacker uses Heracles to bypass the interleaving of two variables. The attacker must only guess two variables, of which one is monotonously incrementing and the other one can either be zero or one. The program increments *loop_var* in each loop iteration. Both variables are predictable, and thus, the plaintext memory block of those two variables is also predictable. In each loop iteration, only two variations are possible, as depicted in Figure 9. An attack can use Heracles to leak the branch variable through loop iterations, effectively leaking the key.

6 Case-Studies

We present 5 novel case studies showing the impact of Heracles. While the first 4 case studies are exclusive to Heracles, the last case study showcases how Heracles re-enables previous attacks.

6.1 Memcpy

Heracles leaks data moved by `memcpy`. We discuss how `memcpy` works on modern processors and its use of hardware acceleration.

Legacy `memcpy` implementations use a combination of 64, 32, and 8-bit stores to move data. This kind of implementation has two downsides: it is slow due to non-optimized loads and stores, and implementing it might be cumbersome—software needs to mimic the memory copy in a loop, and finding the optimal size may vary from CPU generation to CPU generation. The `rep` prefix allows to repeat an instruction as many times as defined in a register The `rep` prefix may only be used with a pre-defined set of instructions (e.g., `movs` instructions). Initially, the `rep` prefix was used to prefix the movement of 8-byte memory blocks in `memcpy` (`rep movsq`). Not all `memcpy`'s are 8-byte aligned; thus, it was necessary to add additional code to copy the unaligned data.

Intel and AMD introduced a microarchitectural optimization called `FSRM` to move complexity from the software layer to the microcode. `FSRM` allows taking the `movsb` instruction, prefixing it

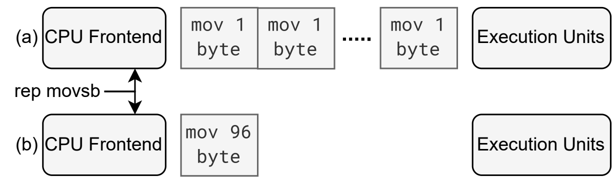

Figure 10: (a) expected `rep movsb` μops breakdown according to public documentation (b) observed `rep movsb` breakdown. Memory is moved in chunks of 96 bytes if size permits.

with `rep` (`rep movsb`), and the microcode decides the optimal block size to move data rather than just copying data in 1-byte chunks. According to Intel benchmarks, it outperforms SIMD instructions such as AVX in most memory copy scenarios [25].

`rep movsb`. Linux uses `rep movsb` as its `memcpy` backend if the CPUID flag `FSRM` is present. All modern AMD EPYC CPUs we tested support `FSRM`, namely Zen 3/4/5. Code can use one `rep movsb` to move gigabytes of memory. Despite being fast, it can still block the core for a relatively long time. The CPU may receive interrupts to interrupt individually executing instructions or μops to circumvent the long blockage. From the documented μops [6], it may seem we can trivially interrupt `rep movsb` between individual `movsb` instructions and use Heracles to leak the data. However, our experiments single-stepping `rep movsb` reveal microcode optimizations. We either get zero or single steps, but the single step moves 96 bytes at once. This shows the CPU pipeline recognizes `rep movsb` and performs an uninterruptible optimization to move multiple bytes atomically. Figure 10 (b) shows the simplified flow in the CPU.

Breaking the Optimization. Neither AMD nor Intel document the microcode optimizations, and public sources such as the LLVM MCA [35] contain insufficient information regarding `rep movsb`. Thus, we do not know what optimizations AMD uses and if they are reversible. Initially, we tried exhausting the store ports of the CPU with a co-located hyperthread but were unsuccessful. Ultimately, we mapped the source and destination memory as uncachable through the hypervisor-controlled SLAT page tables. The data pages being uncachable reverts the internal microcode optimization and results in `rep movsb` being effectively unrolled into a sequence of `movsb` instructions. Figure 10 (a) shows the corresponding result in the CPU pipeline. We use SEV-Step to single-step single `movsb` operations and Heracles to leak the copied data. Note that simply single-stepping `rep movsb` is not sufficient to leak any data; we still need to use the primitive in combination with Heracles.

Impact. Since the Linux kernel uses `rep movsb` for its `memcpy` implementation, this allows Heracles to leak data passed to memcpy. As a second requirement, we need a priori knowledge about the destination memory address to successfully execute Heracles. In most cases, the requirement is fulfilled as Linux zeros out many buffers before use. We manually analyzed the kernel and found that the pre-condition holds for the kTLS and DM-crypt key buffer. Table 2 shows the memory copies of different sizes and whether the destination buffer is zeroed for all `memcpy` invocations in Linux. Zeroed destination buffers can always be leaked by Heracles, whereas non-zero buffers require a case-by-case analysis of the underlying function. We can see that most of the larger memory copies have

Table 2: Linux kernel memcpy destination buffer contents prior to the memcpy listed by size.

	0 to 8 bytes	8 to 16 bytes	16 to 64 bytes	64 to 128 bytes	128 to 1024 bytes	> 1024 bytes
Zero	145917	66244	55306	13585	1689	24625
Non-Zero	187168	13828	27250	2373	257	5457
Zero-Rate	43.81%	82.73%	66.99%	85.13%	86.79%	81.86%

the destination buffer zeroed and are thus valid targets. Our experiment shows that larger buffers are more likely to have a zeroed memory compared to small (< 8-byte) buffers.

6.2 Bash

Bash is the default shell on nearly all Linux distributions. Independent of the terminal emulator, all characters entered by a user eventually end up in shell memory for further processing. Our experiments show bash processes all characters the user enters unless another application (e.g., sudo, vim) runs in the session. SSH sessions on the remote server also create a Bash shell in the CVM. We leak all characters a user types into a CVM Bash shell.

```
if (rl_line_buffer == 0)
  rl_line_buffer = (char *)xmalloc (256);
```

Listing 3: Bash input buffer init

Bash allocates a 256-byte buffer at startup using xmalloc (Listing 3). The buffer holds all character inputs typed by a user. If the user presses Enter, the buffer is reset, and the length is set to 0, but there is no new allocation of memory. This implies the buffer's memory location remains static throughout the lifetime of the shell. Since no memory has been freed before initializing the buffer, all the memory returned by malloc has an initial static value known to an attacker. As the user types, the entered characters are appended to the character buffer allocated at the beginning. No matter how fast the user types, Bash uses a `read` syscall with length 1 and only reads one character at a time. We use HERACLES byte-by-byte changing data primitive to leak all characters a user enters in Bash.

6.3 Sudo

Sudo is used as the default utility to run commands as root on a majority of Linux distributions. Executing Sudo followed by a command requires the user to authenticate themselves for the command to execute as root. There are multiple ways of authentication, but the most common one is by entering the password. Since Sudo is running, the user password is not buffered by Bash memory but is directly written into Sudo's memory. Like Bash, Sudo processes the password string character by character in a memory buffer. Sudo does not use malloc functions to allocate a buffer but resorts to memory in the BSS section. Sudo zeroes the memory before being used as a buffer for the password.

```
while (true) {
    nr = read(fd, &c, 1);
    if(c == '\n')
        break;
    [...]
    *cp++ = c;
}
```

Listing 4: Sudo Password Reading

Listing 4 shows how Sudo copies the password to the memory location. Between each character, we use HERACLES to leak the password byte-by-byte. We leak the password effectively since we know the initial memory values of the buffer.

6.4 Mongoose

Mongoose is a lightweight `C/C++` webserver, with the reference usage requiring just a few lines of code to function. HERACLES leaks the session cookie to effectively hijack the user session. We verify that during runtime of said reference, the text buffer containing the `HTTP` request is located at a static memory address. Modern web applications often identify sessions using cookies or tokens, mostly consisting of character encoding data. As a consequence, an attacker, who can cause web requests to be issued from a context where the victim is authenticated and therefore has a cookie set, can cause repeated requests containing this value with a different length prefix. In combination with the requests of different lengths, we use HERACLES to leak data using the chosen boundary primitive.

A typical setting for this could include the user being signed in to the victim web application while also accessing a web page under the control of the attacker. The attacker can then, for example, use `window.location.replace` or a similar API to cause repeated requests, which cause the cookie to be placed in carefully controlled, shifting locations on the server. Then, with the HERACLES primitives, the attacker can leak and overtake the victims' session. This attacker model may seem strong. However, for example, a malicious cloud provider can be a realistic and sufficient instance of such an attacker in the real world.

6.5 mbedtls

`mbedtls` is a `C` cryptographic library providing a TLS implementation, as well as access to its building primitives. We observed that the function responsible for computing the modular exponentiation, when built *out of the box*, places two important, exponent-dependent variables on well-aligned offsets on the stack. Namely, `window_bits` and `window` in Listing 5 are the only variables placed next to each other in a 16-byte block. A trace of these variables provides enough information to fully reconstruct the exponent. We use single-stepping to obtain such a trace and HERACLES to decrypt it. The variable space is limited by the maximal window size. In the code, this is defined as 6; however, the default value is statically set to 3. Therefore, we only have to test $3 * 2^3 = 24$ values. For many usages of this operation, the exponent is either the key itself, or at least key-dependent, thus resulting in a powerful attack against this implementation.

```
do {
    /* [...] */
    /* Insert next exponent bit into window */
    ++window_bits;
    window <<= 1;
    window |= (E[E_limb_index] >> E_bit_index) & 1;

    /* [...] */
} while (!(E_bit_index == 0 && E_limb_index == 0));
```

Listing 5: mbedtls loop with exponent dependent variables

7 Implementation & Evaluation

We perform our experiments on an EPYC 7313 processor with SEV firmware version `1.55:21` because SEV-Step [57] was developed on the same CPU generation. The CPU runs microcode version `0x0a0011d5`. We verified the availability and functionality of the `SNP_PAGE_MOVE` API on an EPYC 9135 and EPYC 9334 processor with SEV firmware version `1.55:44` and `1.55:32`, respectively. SEV firmware version `1.55:44` marks the latest version available to our motherboard at the time of writing. Table 3 contains the software version we use.

Table 3: Application Version and Commits.

App	Branch	Commit/Version
Sudo	master	`627ae4b09c744a7c`
Bash	master	`6794b5478f660256`
mbedtls	dev	`c811fb79ad7cd6ad`
mongoose	Release	v7.17
Linux	master	`adc218676eef2557` (v6.12)

Getting Data into the CVM. Heracles uses network packets to inject attacker-controlled data for the CPO in the kernel. Specifically, we develop a method to inject page-continuous data using ping packets. Our method has the advantage that we can fill an entire page with attacker content. As the hypervisor, we set the MTU to 9000 and send ping packets with the respective payload of up to 8900 bytes. The kernel places the payload in up to three additional continuous pages. A payload of 8900 bytes exceeds the `struct sk_buff` size, and thus, Linux resorts to so-called fragments to store the additional data associated with the ping packet. The `struct sk_buff` holds references to the fragments. Importantly, the fragments are contiguous in virtual kernel memory, which results in also being contiguous in physical memory for 4KiB chunks.

Leaking 1 byte of data. Leaking 1 byte of random data takes, in the worst case (256 tries), around 2.5 seconds. One-fifth of the time is attributed to swapping the pages, which takes 1 millisecond per swap, accumulating to roughly 500 milliseconds. The remaining time is attributed to QEMU picking up the ping packet, QEMU sending the network packet to the virtio queue, QEMU sending the interrupt to the CVM, the CVM picking up the interrupt, and ultimately the CVM copying the data from the untrusted hypervisor buffer into its address space. By finding optimized I/O interfaces, earlier hookpoints, and integrating QEMUs functionality directly into the exploit code, it is possible to significantly reduce the time needed to leak one byte. However, 500 milliseconds poses the lower limit since swapping pages remains a necessity. The success probability is 100%. We leave the optimization of the attack for future work.

Single Stepping `rep movsb`. We use SEV-Step [57] to single-step `rep movsb`. We perform our experiments on the same CPU generation, namely Zen 3, as SEV-Step. On newer EPYC generations, we experienced problems with setting up SEV-Step, and porting SEV-Step to newer generations is beyond the scope of this paper.

SEV-Step reports the highest accuracy with a timer value of `0x33`. We use a higher timer value since we map some of the SEV guest pages as uncachable in the SLAT page tables. Mapping pages as uncachable likely causes a higher latency to fetch data, and thus, needs a higher timer value to reliably single-step. Table 4 shows the zero, single, and double steps of `rep movsb` we achieve using a test `memcpy` within the guest. We allocate two `0x256000` byte buffers in the guest Linux kernel, disable interrupts, and subsequently call `memcpy` to copy data from the source to the destination.

Table 4: Stepping precision of `rep movsb` for different timer values.

	0-Steps Absolute	1-Steps Absolute	2-Steps Absolute	0-Steps Percentage	1-Steps Percentage	2-Steps Percentage
$0x53$	283751	2449002	203	10.38%	89.61 %	0.007%
$0x54$	331872	2449384	12	11.93%	88.07%	0.0004%
$0x55$	197928	2444132	2638	7.48%	92.41%	0.09%
$0x56$	211342	2432732	8338	7.96%	91.71%	0.314%

We do not encounter any triple steps in our experiments, and only rare occasions of double steps. An attacker needs to brute force 2^{16} bytes in case of double-steps. While brute forcing two bytes roughly takes 256 times longer than brute forcing 1 byte, it is still computationally feasible. It is easy for an adversary to distinguish between zero and single steps by observing a change in the ciphertext in the destination memory. The adversary knows with 100% probability that there has been a single or a double-step if there has been a change in the ciphertext. By setting the timer value to `0x54`, we decrease the likelihood of double-steps to 0.0004%. Our experiments do not show much difference between a timer value `0x53` and `0x54` in terms of probability for different step sizes. We assume that the timer value stabilizes and double steps are caused by a specific sequence of microarchitectural events.

Interrupting `memcpy`. Heracles uses techniques from WeSee [45] and SEV-Step [57] to interrupt `memcpy`. We achieve single-stepping the execution of `memcpy` through SEV-Step. After each character, the attacker injects a virtio interrupt into the CVM, causing it to copy attacker-controlled data into its memory. Linux copies and processes (only for certain applications, e.g., ICMP packets) the data in an IRQ context. We repeat the process until Heracles succeeds. Our experiments show that for random data, we need on average 1.3 seconds per byte to leak. Whereas with reduced entropy data, such as ASCII characters, we can halve the time to 0.7 seconds.

Bash. Bash reads the input character by character. Bash processes the character on the fly, i.e., for auto-completion, and does not wait for the full input. This makes it slightly more challenging to profile since Bash will not only process the password but also call other functions to parse the input. We use two pages and no single-stepping to monitor changes in the ciphertext. We fault on the `read` syscall page and the userspace page of bash to read the user input character-by-character. If we encounter a page fault on the `read` page or Bash page, we mark the other page as non-present, resulting in another page fault when it is executed/accessed. On every page fault of the bash page, we check if there has been a change in the ciphertext and execute Heracles if that is the case.

Sudo. Unlike `rep movsb`, Sudo does not require precise single stepping. It is sufficient to rely on page faults. Sudo reads the password character-by-character using the `read` system call. We have to execute Heracles right after Sudo reads one character through `read`. We profile it by page faulting on the page hosting the while-loop in

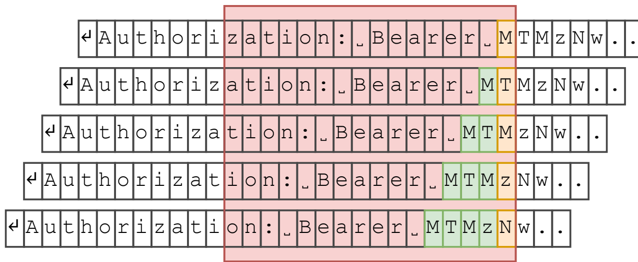

Figure 11: Illustration of how we use Heracles to leak Mongoose cookies. By *moving* the unknown secret backwards in the 16–byte block block under attack, we ensure that only 1 byte of said block is unknown.

Listing 4 and the `read` system call function. The buffer containing the password is zeroed and `0x100` byte aligned by the compiler for security reasons. Sudo reads the password character-by-character in the `getln` function. Sudo allocates the static buffer for the password within `tgetpass` function, where it is zeroed on every call to the function. The compiler injects the code to zero the buffer indirectly. This is the default behavior for Sudo since it was built with many mitigations on by default. The buffer size is `0x400` bytes. Our attack succeeds with 100% probability.

Mongoose. A simple web service, written using the Mongoose framework [37], only has a single event loop, where received HTTP requests are parsed and then handed off to a callback specified by the user. We found that the parsing step always leaves the textual representation of the HTTP request at the same address. Furthermore, much of the content of this buffer is inherently known due to the predictable fields in the header. Consequently, by causing the victim's web browser to repeatedly make requests containing the secret shifted by some bytes, we can leak the secret value and, depending on the implementation, hijack the session. Figure 11 illustrates the attack. On the server side, we use techniques from prior work to page fault once the webserver has filled the buffer. The attack proceeds in much the same way as the other case studies, by using prior knowledge of the content, and at most 256 tries of brute forcing, to leak the next byte of the secret.

mbedtls. We leak the exponent in `mbedtls` `mbedtls_mpi_core_exp_mod` function. The function uses the sliding window algorithm for fast exponentiation. By leaking the window bits, we effectively reconstruct the exponent. The program parses the exponent bit-by-bit; therefore, the window variable changes bit-by-bit. Between each change, the algorithm executes between 34546–50512 instructions, depending on whether the window is being flushed. By tracing the accesses to the variable `window` and `window_bits`, which make up one block of ciphertext, we can trace the execution of the program. By observing the ciphertexts and using Heracles to infer the plaintext values, we successfully recover the used exponent with 100% probability. As the default configuration only ever produces 24 different plaintext combinations and always uses the same tweak, we can build a dictionary of the corresponding ciphertexts.

8 Discussion

We discuss the trade-offs of Heracles, the root causes, and why current software mitigations are insufficient. We outline hardware mitigations to eliminate each of the root causes.

8.1 Speed vs. Generality

Cipherleaks [34] and follow-up work [31] leak keys in 0.5–8.53 seconds. They achieve such speeds because they only need to compare the ciphertext to one value in their dictionary. Heracles achieves a similar speed for the case where we use P3 and P4 to leak data. However, with increased speed comes limitations in the variety of leakage. While P3 and P4 have a small search space, P1 and P2 must test 128 values on average in each iteration. We trade off performance in favor of leaking more generic data patterns.

When applied naively, Heracles takes up to 2.5 seconds to leak one random byte in the worst case. While 80–90% of the time is attributed to our non-optimized implementation, the remaining 10–20% is attributed to the PSP swapping the pages and cannot be optimized. Even with 250–500 milliseconds to leak one byte, the lower bound is up to 256 seconds to leak a 1024-byte buffer. During this time, the CVM thread stalls and may be unresponsive. This may seem limiting at first. However, we can split the attack into two phases: an active ciphertext collection phase, which is as fast as Cipherleaks [34], and a second active phase where we recover the plaintext. An attacker can recover the plaintext during the second active phase while the CVM has no active user connected. This may be the case if the user disconnects from the CVM or the cloud provider drops network packets to fake a network problem. The guest cannot differentiate between a network problem and an attack, which can make Heracles invisible to the guest.

Further, depending on the leaking primitive, we can significantly reduce the attack time. First, ASCII only comprises 128 different characters, and some characters are more likely than others. Thus, if we leak an ASCII password, we only need 0.4 seconds on average for each character. To leak a 16-character password in sudo Heracles only needs 6.5 seconds, which is comparable to prior works needing 10–20 seconds [45, 61].

8.2 Root Cause Analysis

Three different design flaws in SEV-SNP enable Heracles: (1) the ability of the hypervisor to read ciphertext; (2) the lack of freshness of the memory encryption; and (3) the hypervisor's ability to move pages. While one of them alone does not impact SNP's security severely, grouped, they are a considerable threat. Changing one of the flaws fully stops our proposed attacks and greatly improves SEV-SNPs' security. Selected software fixes may apply to stopgap our case studies introduced in Section 6, but addressing the leakage patterns in Section 5 requires hardware changes.

8.3 Software-Based Defenses

Current ciphertext side-channel defenses exclusively focus on in-place changing data (Section 5.4) [28, 41, 54] or userspace [55]. The insight is to blind the plaintext value such that, despite being the same plaintext, the ciphertext differs. Simply speaking, they add a software-defined tweak value to the plaintext. Depending on the

application, the mitigations cause overhead, and the binary must be specifically compiled with the mitigations in place.

In principle, adding software tweak values to memory stops HERACLES from leaking memory. Practically, all proposed defenses exclusively protect userspace applications, but HERACLES leaks not only userspace but also kernel memory. Thus, even if all userspace applications are hardened, HERACLES still leaks userspace data processed in the kernel (e.g., keys and file I/O). Applying the proposed defenses to kernelspace is not straightforward for multiple reasons. All data structures interfering with hardware cannot use proposed interleaving defenses as it would break the expected layout (e.g., ring buffer queue, DMA engine) or masking, since the value is read at some point by hardware [28, 41]. Kernelspace must initialize the defense and manage its memory, but at the same time, the kernel must be protected by the defense, leading to a bootstrapping problem. Multiple virtual addresses map to one physical address, making masking-based defenses error-prone [28]. Lastly, SSE instructions (needed for 16-byte atomic writes in Zebrafix [41]) are generally not available at all places in the kernel (e.g., entry/exit paths) and require invasive changes to those paths. In summary, at the time of writing, software-based defenses do not mitigate all occurrences of HERACLES attacks in practice, due to their focus on userspace. Future defenses should be designed to protect the kernelspace.

8.4 Hardware Mitigations

We discuss 3 potential hardware mitigations and their overhead.

Fresh Tweak Values. HERACLES relies on XEX tweak values being static throughout the lifetime of a CVM. Generating a fresh value on every memory write for every 16-byte block stops HERACLES. However, generating fresh values on every memory write requires invasive hardware changes. The memory controllers would need to allocate DRAM to store the random tweak values and must become capable of generating random values on the fly as memory requests arrive. Furthermore, writing the tweak values to DRAM would decrease the memory controller performance by a factor of at least two, which may not be a tolerable performance-security tradeoff. Using fresh tweak values is the correct cryptographic mitigation to stopgap HERACLES. It requires invasive changes and results in performance degradation.

Limiting Move APIs. Without the capability of moving pages, HERACLES cannot circumvent the tweak values and thus cannot build a CPO. Disabling the hypervisor's ability to dynamically move pages through the PSP stops our attack at no direct performance cost. However, there might be indirect costs since cloud providers rely on overprovisioning their machines and thus require a way of swapping out guest memory pages to disk. Furthermore, cloud providers move pages to defragment their memory and create more efficient memory regions, which reduces TLB pressure by combining smaller pages into larger ones. Limiting the move APIs can be easily deployed without changes to the hardware. AMD announced this feature with their SEV-SNP ABI specification 1.58 in May 2025 [7]. However, the specification only limits the PSP API to move pages; it remains to be seen if the MPDMA engine is also affected.

Restricting Ciphertext Visibility. Previous ciphertext attacks were application-specific and fixable by software. In contrast, HERACLES introduces generic leakage patterns, both in kernel and userspace, observable to the hypervisor. We recommend restricting ciphertext access as a mitigation to stop all ciphertext-based attacks. Starting with Zen 5, the hypervisor may enable an additional security feature to limit the hypervisor's visibility of guest ciphertexts. Enabling it results in all memory read accesses of the hypervisor being checked by hardware. Previously, only write accesses were subject to those checks. This mitigation fully stops HERACLES as well as software-based ciphertext attacks in general. We believe this is the mitigation all cloud providers should deploy. Most of the deployed SEV-SNP installations do not have hardware support at the time of writing [7]. The feature is not enabled by default and may induce high performance overhead [7]. We were unable to quantify the performance overhead of the ciphertext hiding feature on Zen 5—either silicon or SEV firmware support or both are missing for this feature. We use the newest SEV firmware version `1.55:54` to perform our experiments. When setting the ciphertext hiding bit according to the SEV-SNP documentation [7], the initialization fails with an `INVALID_PLATFORM` error code. As per SEV-SNP specification [7], our firmware version should support ciphertext hiding. The CPUID flag, on our Zen 5 CPU, indicates support for ciphertext hiding, yet the SNP initialization fails if we enable it. Since some features may require the latest socket, we contacted our motherboard supplier about the lack of feature support. Our analysis concludes that despite the announced support for the ciphertext-hiding feature, software/hardware support is still missing.

For platforms without ciphertext-hiding support, we recommend limiting the availability of the hypervisor to move pages as a software solution, as we summarized above. Disabling the move APIs limits the flexibility of the hypervisor but mitigates HERACLES.

8.5 Applicability to other TEEs

Other TEE platforms, such as Intel TDX [26], Intel SGX [24], and Arm CCA [9] restrict the availability of ciphertext. TDX, for instance, returns `0x00` if the hypervisor tries to read TD-owned memory. However, TDX allows physical pages to be relocated with the `TDH.MEM.PAGE.RELOCATE` API . According to the specification, TDX encrypts the same plaintext at different timestamps to the same ciphertext, since it uses the XTS encryption mode and does not cycle its IVs [26]. Thus, HERACLES may also apply to Intel TDX if an attacker has the capability of snooping the DRAM bus. The same applies to SEV-SNP when ciphertext-hiding is enabled.

9 Related Work

We discussed the closely related works in Section 2.1, and Table 1 summarizes the differences in attacker capabilities.

Ciphertext Based Attacks. Cipherleaks [34] exploits the lack of freshness in the VMSA register page to build a register dictionary to leak cryptographic keys. Subsequently, Li et al. [31] look at other memory locations where register data may be temporarily stored and exploit them similarly to Cipherleaks. Ciphersteal [60] and Hypertheft [59] use a ciphertext side-channel on SEV-SNP to leak DNN

weights and neural network input data, respectively. Beast [16] introduces a plaintext recovery attack with chosen boundary privilege against TLS 1.0/ SSL 3.0. Similarly, Bosman et al. [11] use the chosen boundary privilege in combination with memory deduplication to leak secrets. Li et al. [32, 33] build an encryption and decryption oracle on SEV. However, to construct the oracle, the authors use weaknesses specific to SEV, which are mitigated on SEV-SNP. Heracles builds a Chosen Plaintext Oracle by combining the weakness in XEX-based encryption scheme on AMD SEV-SNP with the hypervisor's ability to bring in new data into the CVM, move pages in DRAM, and read ciphertext.

Other Attacks on AMD SEV. Buhren et al. [12] glitch the AMD Platform Security Processor to leak confidential keys embedded into the processor. Hetzelt et al. [23] analyze Linux device drivers for CVMs and reports multiple weaknesses. Cachewarp [61] uses the `INVD` instruction to compromise SEV-SNP CVMs. Heckler [46] and WeSee [45] use the interrupt injection interface of a CVM to compromise its execution integrity. SEVerity [39] injects arbitrary code into SEV-ES CVMs. SEVurity [56] introduces new methods to inject data into an SEV-ES CVM and reverse engineers the XEX value protection. Heracles also leverages the hypervisor's ability to inject new data into the CVM in SEV-SNP and benefits from the XEX analysis. SEVered [38] extracts data from SEV CVMs by manipulating second-level address translation tables. CounterSEVeillance [17] exploits the availability of performance counters on SEV-SNP CVMs to leak branch information and division results. Chiang et al. [13] exploit cache and memory contention side channels to infer confidential information from the guest. BadRam [15] uses unauthenticated DRAM memory module sizes to create memory aliasing on AMD EPYC platforms, breaking SEV-SNP. Google found multiple vulnerabilities in SEV-SNP as part of a pre-release security analysis [19]. Some of these works, similar to Heracles, use the attacker's ability to observe page faults and/or single-step the CVM to refine the attack window.

Attacks on Intel-based TEEs. AEPICLeak [10] is a non-side channel architectural bug to compromise SGX. There are numerous side-channel [21, 22, 22, 40, 48, 50–52, 58] and interface [14, 30, 44, 47, 49, 53] attacks on SGX. Heckler [46] uses `int0x80` interrupts to compromise a TDX CVMs. TDXDown [27] single steps TDX CVMs to amplify a cache side-channel. Google reported multiple weaknesses in TDX during their pre-production analysis [20]. Neither SGX nor TDX allow the untrusted privileged software to observe the victim enclave/CVM ciphertext, rendering Heracles or even prior ciphertext attacks impossible.

10 Conclusion

We present Heracles, an attack on AMD SEV-SNP that exploits the hypervisor's ability to not only observe ciphertext but also move XEX-based encrypted pages in DRAM. Based on these capabilities, we show that the hypervisor can launch bruteforce-based attacks to leak CVM data. Then we build novel leakage patterns to amplify the leakage and demonstrate it on 5 case-studies. Heracles serves as a strong motivation to revoke the hypervisor's ability to observe ciphertext on AMD SEV-SNP.

11 Acknowledgment

We thank Supraja Sridhara, Kenny Paterson, and the reviewers for their constructive comments and discussions, which helped improve the paper. Thanks to the authors of SEV-Step for open-sourcing their framework, which helped us in accelerating our prototype design.

References

[1] Alibaba. 2024. Build a TDX confidential computing environment .
[2] Amazon. [n. d.]. AWS Nitro Enclaves - Create additional isolation to further protect highly sensitive data within EC2 instances. https://aws.amazon.com/ec2/nitro/nitro-enclaves/.
[3] AMD. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity protection and more.
[4] AMD. 2023. AMD-ASPFW.
[5] AMD. 2023. 58151: Tiered Memory Page Migration Operations Guide.
[6] AMD. 2023. Repeat String Operation Prefix.
[7] AMD. 2025. SEV Secure Nested Paging Firmware ABI Specification.
[8] AMD. accessed 2025-08-02. AMD SEV. https://www.amd.com/en/processors/amd-secure-encrypted-virtualization.
[9] ARM. accessed 2025-8-2. Arm Confidential Compute Architecture (ARM-CCA).
[10] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*.
[11] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*. Paper=https://download.vusec.net/papers/dedup-est-machina_sp16.pdfWeb=https://www.vusec.net/projects/dedup-est-machinaPress=https://goo.gl/ogBXTm
[12] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *ACM CCS*.
[13] Li-Chung Chiang and Shih-Wei Li. 2025. Reload+Reload: Exploiting Cache and Memory Contention Side Channel on AMD SEV. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1014–1027. https://doi.org/10.1145/3676641.3716017
[14] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. 2021. SmashEx: Smashing SGX Enclaves Using Exceptions. In *ACM CCS*.
[15] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. 2025. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In *46th IEEE Symposium on Security and Privacy (S&P)*.
[16] Duong and Juliano Rizzo. 2011. Here Come The XOR Ninjas Thai. https://api.semanticscholar.org/CorpusID:50845005
[17] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. 2025. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In *Network and Distributed System Security Symposium 2025: NDSS 2025*.
[18] Google. [n. d.]. Confidential Computing | Google Cloud. https://cloud.google.com/confidential-computing.
[19] Google. 2022. AMD Secure Processor for Confidential Computing.
[20] Google. 2023. Intel Trust Domain Extensions (TDX) Security Review.
[21] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.
[22] Wenjian He, Wei Zhang, Sanjeev Das, and Yang Liu. 2018. Sgxlinger: A new side-channel attack vector based on interrupt latency against enclave execution. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 108–114.
[23] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. 2021. VIA: Analyzing Device Interfaces of Protected Virtual Machines. In *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. ACM, 273–284. https://doi.org/10.1145/3485832.3488011
[24] Intel. [n. d.]. Intel Software Guard Extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.
[25] Intel. 2024. Intel Optimization Reference Manual Volume-1 v50.
[26] Intel. accessed 2025-8-2. Intel Trust Domain Extensions (Intel TDX).
[27] Intel. accessed 2025-8-2. Intel Trust Domain Extension Research and Assurance.
[28] Ke Jiang, Sen Deng, Yinshuai Li, Shuai Wang, Tianwei Zhang, and Yinqian Zhang. 2025. CipherGuard: Compiler-aided Mitigation against Ciphertext Side-channel Attacks. arXiv:2502.13401 [cs.CR] https://arxiv.org/abs/2502.13401
[29] David Kaplan. 2017. SEV-ES.
[30] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *ASPLOS*.

[31] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *IEEE S&P*.
[32] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2021. CrossLine: Breaking "Security-by-Crash" Based Memory Isolation in AMD SEV. In *ACM CCS*.
[33] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *USENIX Security*.
[34] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security*.
[35] LLVM. 2025. LLVM MCA.
[36] Microsoft. [n. d.]. Azure confidential Cloud - Protect Data In Use | Microsoft Azure. https://azure.microsoft.com/en-us/solutions/confidential-compute/.
[37] Mongoose. 2025. Mongoose.
[38] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *EuroSec*.
[39] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. 2021. SEVerity: Code Injection Attacks against Encrypted Virtual Machines. In *IEEE S&PW*.
[40] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. 2021. Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 663–680. https://www.usenix.org/conference/usenixsecurity21/presentation/puddu
[41] Anna Pätschke, Jan Wichelmann, and Thomas Eisenbarth. 2025. Zebrafix: Mitigating Memory-Centric Side-Channel Leakage via Interleaving. arXiv:2502.09139 [cs.CR] https://arxiv.org/abs/2502.09139
[42] Martin Radev and Mathias Morbitzer. 2021. Exploiting Interfaces of Secure Encrypted Virtual Machines. In *Reversing and Offensive-Oriented Trends Symposium* (Vienna, Austria) *(ROOTS'20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3433667.3433668
[43] Phillip Rogaway. 2004. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In *Advances in Cryptology - ASIACRYPT 2004*, Pil Joong Lee (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–31.
[44] Jose Rodrigo Sanchez Vicarte, Benjamin Schreiber, Riccardo Paccagnella, and Christopher W. Fletcher. 2020. Game of Threads: Enabling Asynchronous Poisoning Attacks *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3373376.3378462
[45] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. 2024. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *IEEE S&P*.
[46] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. Heckler: Breaking Confidential VMs with Malicious Interrupts. In *USENIX Security*.
[47] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, and Shweta Shinde. 2024. Starmie: Exploiting Signal-handling in Intel SGX Enclaves. In *ACM CCS*.
[48] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 991–1008.
[49] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. 2019. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *ACM CCS*.
[50] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX*.
[51] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *ACM CCS*.
[52] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*.
[53] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *Computer Security – ESORICS 2016*, Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows (Eds.). Springer International Publishing, Cham, 440–457.
[54] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. 2023. Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6789–6806. https://www.usenix.org/conference/usenixsecurity23/presentation/wichelmann
[55] Jan Wichelmann, Anja Rabich, Anna Pätschke, and Thomas Eisenbarth. 2024. Obelix: Mitigating Side-Channels Through Dynamic Obfuscation. In *2024 IEEE Symposium on Security and Privacy (SP)*. 4182–4199. https://doi.org/10.1109/SP54263.2024.00261
[56] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *IEEE S&P*.
[57] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. 2023. SEV-Step A Single-Stepping Framework for AMD-SEV. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 1 (Dec. 2023), 180–206. https://doi.org/10.46586/tches.v2024.i1.180-206
[58] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*. 640–656. https://doi.org/10.1109/SP.2015.45
[59] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. 2024. 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* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4346–4360. https://doi.org/10.1145/3658644.3690317
[60] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. 2025. CipherSteal: Stealing Input Data from TEE-Shielded Neural Networks with Ciphertext Side Channels . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 79–79. https://doi.org/10.1109/SP61157.2025.00079
[61] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security*.