Shadows in Cipher Spaces: Exploiting Tweak Repetition in Hardware Memory Encryption

Wei Peng[*] Yinshuai Li[*] Yinqian Zhang[†]
Research Institute of Trustworthy Autonomous Systems
Department of Computer Science and Engineering
Southern University of Science and Technology

Abstract

Hardware memory encryption serves as the foundation for TEE security, where processors transparently encrypt data bound for DRAM while maintaining plaintext within CPU boundaries—a critical defense against physical attacks like memory bus snooping and cold-boot attacks. Although ubiquitous in major TEE implementations (Intel SGX/TDX, AMD SEV), design flaws have introduced severe vulnerabilities including ciphertext replacement attacks, ciphertext replay attacks, and ciphertext side-channel attacks.

Our work makes three key contributions: First, we present the first comprehensive analysis of Hygon CSV's memory encryption engine, a prominent TEE in China's confidential computing market. Second, we identify a novel vulnerability class stemming from tweak value repetition within 64-byte blocks, causing identical 16-byte plaintexts to generate identical ciphertexts. Third, we demonstrate how this enables CipherShadow Attacks through: (1) an automated binary scanner detecting vulnerable code patterns, (2) end-to-end attacks demonstrating both OpenSSH authentication bypass and machine learning training data reconstruction.

1 Introduction

Trusted Execution Environments (TEEs) are hardware-assisted secure runtime environments that protect application code and data integrity/confidentiality, even against privileged or physical adversaries (e.g., malicious OS/kernel or system administrators). Their security foundation in part relies on hardware-enforced memory encryption through a dedicated memory encryption engine, which transparently encrypts/decrypts data traversing the memory bus. This maintains persistent encryption in DRAM while enabling plaintext processing within the CPU boundary, effectively mitigating physical attack vectors including memory bus snooping [20] and cold-boot attacks [26].

[*] These authors contributed equally to this work.
Corresponding Author: yinqianz@acm.org

The security guarantees of hardware memory encryption fundamentally depend on implementation choices by hardware vendors, which typically balance security against performance and scalability. For example, Intel SGX employs robust encryption with integrity verification and temporal freshness, but these strong protections imposed strict enclave size limits [22]. While Scalable SGX removed the size constraints, it sacrificed freshness guarantees in the process [31]. On the other hand, VM-based TEEs like SEV and TDX adopt simpler encryption schemes to accommodate larger protected memory regions, but their weaker cryptographic properties have led to demonstrated vulnerabilities [35, 38, 52, 56].

For instance, early versions of AMD SEV were vulnerable to replacement attacks due to weak tweak functions [37, 56]. Later versions, including AMD SEV and SEV-ES since the Zen 2 architecture, were susceptible to replay attacks because of insufficient integrity protection [39]. Even with added write isolation protection and a stronger tweak function, the initial version of AMD SEV-SNP still faces novel ciphertext side-channel attacks [35]. These vulnerabilities stem from two architectural characteristics: (1) the attacker's ability to continuously observe encrypted memory, and (2) the lack of temporal freshness in memory encryption [35, 38].

Motivated by these discovered vulnerabilities, we aim to investigate whether such security flaws represent an endemic problem across TEE architectures. Particularly, this paper delivers the first in-depth security analysis of Hygon China Secure Virtualization (CSV), a commercially significant yet understudied TEE that dominates China's confidential computing market, with widespread deployment in major cloud platforms including Tencent Cloud [6] and Alibaba Cloud [5]. CSV has become the TEE of choice for numerous research initiatives and practical applications [2, 12, 13].

Through our security analysis of CSV's architectural design, we identify a novel class of memory encryption vulnerability, which stems from CSV's reuse of tweak values across multiple 16-byte blocks within a cache line. Specifically, CSV's memory encryption scheme utilizes a single derived key per 64-byte cache line, rather than implementing

distinct tweak values for individual 16-byte sub-blocks. This architectural decision produces deterministic ciphertext patterns wherein any identical 16-byte plaintext sequences within a 64-byte block will generate matching ciphertext outputs.

To validate the implications of this newly discovered vulnerability, we present a new class of attacks that we term as CIPHERSHADOW Attacks. This attack vector capitalizes on the repetitive nature of encryption keys within the same 64-byte aligned memory block, paving the way for the exploitation of inherent weaknesses in memory encryption. The ramifications of CIPHERSHADOW extend beyond mere data leakage, encompassing the potential for adversaries to orchestrate complete takeovers of confidential virtual machines. Such breaches pose a grave threat to the overall security posture of the system, especially when memory integrity measures are supposedly in place.

We make the following contributions in this paper:

- We demystify the design of hardware memory encryption of Hygon CSV and empirically verify its susceptibility to known attacks against AMD SEV.
- We identify a new class of vulnerabilities in Hygon CSV, which is caused by the repeated use of the same tweak values within the same cacheline-sized memory blocks.
- We taxonomize the security designs of hardware memory encryption and establish a connection between various design choices and known attacks against existing memory encryption schemes.
- We introduce CIPHERSHADOW attacks, a novel class of attacks that exploits the vulnerability of repetitive tweak values; design and implement a binary scanner to automate the discovery of exploitable gadgets in program binaries.
- We showcase the power of CIPHERSHADOW attacks, by performing end-to-end attacks demonstrating both OpenSSH authentication bypass and training data reconstruction in machine learning programs.

2 Hygon CSV Demystified

Hygon is a manufacturer of x86 CPUs [1], with its processors widely deployed by major cloud service providers, including Tencent Cloud [6] and Alibaba Cloud [5]. Among its security offerings, Hygon CSV is a prominent VM-Based TEE solution. While CSV has seen significant adoption in both industry and academia [2, 10, 12, 13], its security mechanisms remain understudied. In this section, we conduct a comprehensive analysis of CSV, focusing first on critical aspects of its architecture design, memory encryption scheme, and potential vulnerabilities.

2.1 Comparing Architectures of CSV and SEV

Hygon's CPU architecture is based on the AMD Zen architecture; however, the security features of its CSV technology were developed independently of AMD's SEV. Hygon CSV has evolved through multiple generations—CSV v1, v2, and v3—which align closely with AMD's SEV [33], SEV-ES [32], and SEV-SNP [44] in both architectural design and security objectives. To systematically evaluate these technologies, we provide a detailed comparative analysis, highlighting critical similarities and distinctions between the CSV and SEV architectures.

2.1.1 Similarities between CSV and SEV

CSV v1. As illustrated in Figure 1, CSV v1 employs a nested page table (NPT) for Guest Physical Address (GPA) to Host Physical Address (HPA) translation, similar to SEV. The Address Space ID (ASID) field serves dual purposes: it distinguishes between different VMs' page tables and is used by the Platform Security Processor (PSP) to retrieve each VM's unique encryption key (VEK). This key enables the hardware memory encryption engine to transparently encrypt and decrypt VM memory.

CSV v2. Similar to SEV-ES, CSV v2 provides encrypted protection for VM register states during world switches by storing them in the VM Save Area (VMSA) using SM4 encryption. Also similar to SEV-ES, it categorizes VMEXIT events into Non-Automatic VM Exits (NAE) and Automatic VM Exits (AE), with the former maintaining register confidentiality, and the latter for managing special instructions (e.g., `VMMCALL`, `CPUID`) that require register exposure. The NAE workflow follows SEV-ES's established process - a VMM Communication Exception (VC) [14] triggers the Guest OS's VC handler to write relevant registers to the Guest-Hypervisor Communication Block (GHCB) based on the exit reason, followed by hypervisor processing via `VMGEXIT` and subsequent VM resumption through VMRUN.

CSV v3. CSV v3 and SEV-SNP share the fundamental security objective of enforcing robust memory isolation between VMs and untrusted software components. In SEV-SNP, this is achieved through a Reverse Map Table (RMP) that maintains strict guest ownership records for physical pages, preventing unauthorized hypervisor remapping or accesse [15]. Similarly, CSV v3 implements a Secure Memory Isolation Management Unit (SIMU), a hardware module that provides hardware-enforced memory isolation with critical configuration parameters managed exclusively by the PSP. It features a Secure Page Ownership Table (SPOT) similar to RMP to maintain the memory ownership of VM. thereby preventing external access to protected VM physical memory regions [7].

Software Compatibility. CSV maintains Application Binary Interface (ABI) compatibility with SEV, aiming at seamless software interoperability between the two platforms. This compatibility enables OSes and applications designed for SEV environments to run unmodified on CSV implementations. A notable example of this cross-compatibility can be

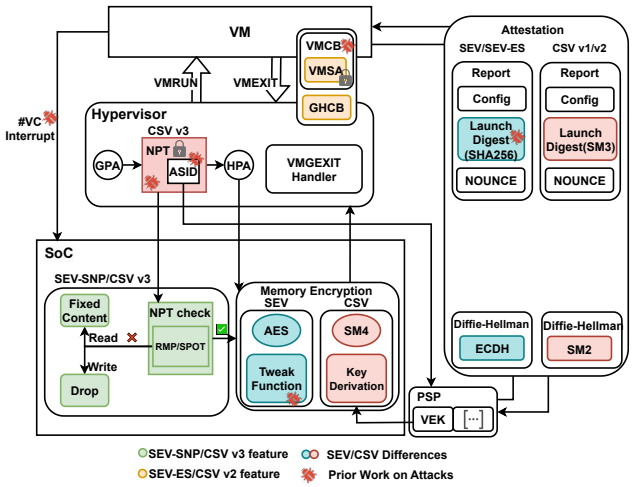


Figure 1: Comparison of the memory encryption design of CSV and SEV. Yellow blocks are features added in CSV v2 and SEV-ES; green blocks are features added in CSV v3 and SEV-SNP; blue and red blocks denote the different designs adopted by SEV and CSV, respectively.

observed in major Linux distributions: both openEuler [9] and Anolis OS [8] operate flawlessly on CSV v2 hardware with their SEV-ES-optimized kernel versions. This binary-level compatibility significantly simplifies the adoption for existing SEV-enabled software stacks in CSV environments.

2.1.2 Differences between CSV and SEV

A key distinction between CSV and SEV lies in their memory encryption engine. While SEV processors rely on international encryption standard, AES [27], CSV adopts the SM4 symmetric encryption algorithm [11], which is compliant with China's national commercial cryptographic standards. Furthermore, CSV employs a specialized key derivation mode for encrypting 128-bit plaintext blocks. We defer our analysis of the memory encryption engine of CSV in Section 2.2.

SEV-SNP uses RMP to protect the integrity of page tables within VMs, while the attributes of NPT are still under the control of the hypervisor. CSV v3 encrypts the NPT and manages it through SIMU, blocking hypervisor access and manipulation [7], serving as a mitigation for page-fault-based side-channel attacks and numerous end-to-end attacks that require page fault tracking assistance.

2.2 Demystifying CSV Memory Encryption

In this section, we empirically explore how CSV's memory encryption is designed.

2.2.1 CSV's Encryption Mode

A disclosed CSV patent [3] details a dynamic key derivation mechanism that regenerates encryption keys per cacheline-sized (64-byte) memory block - a fundamental architectural divergence from static-key approaches employed in SEV and TDX. The cryptographic operations are formally defined as:

$$Enc : M \xrightarrow{Enc,P} E_{K \oplus T(P)}(M) = C$$
$$Dec : C \xrightarrow{Dec,P} D_{K \oplus T(P)}(C) = M$$

In this model, each memory block is encrypted using a key derived from the root key K and its physical address P. Specifically, the encryption key is computed as $K \oplus T(P)$, where the tweak function $T(P)$ is a transformation of P (e.g., hashing or bit-masking) to a random bit string of the same length as the encryption key. This ensures that the same plaintext stored at different physical addresses results in different ciphertexts, enhancing security against memory remapping or replay attacks. Note that the actual transformation $T(P)$ and key derivation mechanisms are not specified in the patent, making the description conceptual rather than implementation-specific.

2.2.2 Reverse-Engineering CSV's Encryption Scheme

We conducted a series of experiments to reverse-engineer CSV's encryption mode, with respect to encryption freshness, granularity of encryption, and protection of integrity.

Experimental Setup. We collected plaintext-ciphertext pairs by: (1) writing data (plaintext) to C-bit-set memory, triggering encryption to DRAM; and (2) reading from the same address with C-bit unset, retrieving the raw ciphertext. This approach exploits CSV's memory encryption behavior where C-bit controls encryption/decryption.

Observations. We systematically tested the encryption scheme by: (1) varying plaintext values and lengths, and (2) collecting ciphertexts from both identical and distinct memory locations. For comparative analysis, we replicated all experiments on SEV under identical conditions. Our investigation revealed several key findings:

- **128-bit Block Encryption.** Our experiments showed that flipping a single plaintext bit alters only the corresponding 128-bit ciphertext block in CSV, and vice versa, confirming its 128-bit block size encryption.
- **No Freshness.** We observe that when we repeatedly encrypt the same plaintext at the same physical address, the ciphertext remains unchanged. This indicates a lack of freshness in CSV's memory encryption.
- **No Integrity Protection.** Our observations reveal that an adversary with host privileges can modify guest VM ciphertext without triggering system crashes. This demonstrates

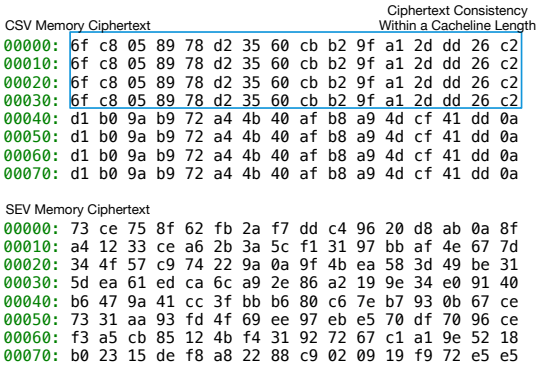


Figure 2: Ciphertext of CSV and SEV from the same 128-byte plaintext blocks.

that CSV v1/v2, like SEV and SEV-ES, provides neither cryptographic integrity protection nor access controls to encrypted guest memory.

- **Tweak Values Repetition: New Problem.** Our experiments observe that CSV reuses the tweak values for encryption key derivation within 64-byte cacheline boundary. As shown in Figure 2, when encrypting all-zero plaintexts, we observe a pattern of repeated 16-byte ciphertext chunks within the same 64-byte block, but completely different ciphertexts between two 64-byte blocks. This suggests that CSV repeatedly uses the same tweak value for the key derivation procedure [3] of each of the 16-byte chunks within the same 64-byte block. We posit that this design decision principally originates from performance optimization considerations for cache-line-aligned memory operations, where maintaining consistent encryption parameters across a full cache line (typically 64 bytes) reduces cryptographic overhead during bulk memory accesses. However, this mismatch - 16-byte encryption granularity and 64-byte tweak granularity - leads to tweak value repetition, a new vulnerability that we will explore in-depth in this paper.

2.2.3 Summary of Findings

To sum up, CSV's memory encryption ressemble that of SEV in that (1) they both employ 128-bit block-mode deterministic encryption schemes; (2) they both use physical address-based plaintext tweak or key derivation; (3) they provide unique encryption key for each confidential VM, (4) they both lack of integrity protection and freshness.

However, these two schemes are different from the following aspects: (1) CSV uses SM4 for encryption and SEV uses AES; (2) In CSV, within a 64-byte aligned memory block, the four 16-byte chunks have identical ciphertext if the plaintexts are the same. In SEV, each of the four 16-byte chunks will have different ciphertexts even if the plaintexts are identical.

2.3 Validating Known Attacks on CSV

Given the architectural similarities between CSV and SEV, we hypothesize that known SEV vulnerabilities may also affect CSV. However, differences in their memory encryption implementations could introduce distinct security properties. To systematically evaluate these possibilities, we conduct empirical testing of CSV against documented VM-TEE attacks (including those targeting AMD SEV and Intel TDX). Our vulnerability assessment results are presented in Table 1. We defer a systematic assessment of defects caused by memory encryption design to Section 3 and focus only on other vulnerabilities in this section.

2.3.1 Interrupt Injection Attack

Interrupt injection attacks from the hypervisor represent a critical security challenge for TEEs, with demonstrated vulnerabilities affecting SGX [47], SEV-SNP [42,43], and TDX [43]. While SEV-SNP introduces two optional hardware modes to restrict hypervisor interrupt injection [44], compatible software implementations had not been widely applied before WeSee [42] and Heckler [43]. CCA [17] employs the Realm Management Monitor (RMM) to protect realm VM states during hardware interrupts, yet recent research shows it still permits hypervisor-initiated interrupt injection [18].

WeSee [42]. SEV-SNP can be compromised through hypervisor-injected #VC exceptions, resulting in unauthorized disclosure and modification of the `rax` registers. Our experiments show that while CSV v1 resists hypervisor attacks by ignoring injected `VC` exceptions (#29) and NAE VMEXIT instructions (e.g., `VMMCALL`). CSV v2/v3 shares SEV-SNP's vulnerability: When injecting VC exceptions from the hypervisor and setting the `exit_reason` corresponding to `VMMCALL`, we successfully handled the `VMMCALL` in the `VMGEXIT` handler, enabling `rax` leakage. Moreover, by manipulating the `rax` values through the `VMGEXIT` handler, we demonstrate that these modifications are observable within the guest VM. This confirms the susceptibility of CSV v2/v3 to WeSee attacks. However, in CSV v3, since page-fault tracking can no longer be performed, successful exploitation using WeSee will become more complex.

Heckler [43]. SEV-SNP is susceptible to malicious injection of INT 0x80 and INT 0 (`SIGFPE`) interrupts by the hypervisor. To verify this vulnerability in CSV, we ran an empty loop in the guest VM while injecting INT 0x80 and INT 0 interrupts from the hypervisor. Our tests showed that both interrupts can be injected successfully: INT 0x80 leads to a segmentation fault in the victim program and INT 0 triggers a divide-by-zero signal that terminates the VM with an `asm_exec_divide_error`. These findings demonstrate that CSV v1-v3 fail to prevent interrupt injection by the hypervisor. In CSV v3, the exploitation of Heckler will be limited by the lack of page fault tracking assistance.

Table 1: Summary of existing TEE attacks on different platforms. Each cell indicates whether a vulnerability has been validated on the corresponding platform. ✔jindicates the platform was initially vulnerable but later patched through hardware. ⚠ indicates a potential threat of attack that is yet to be validated.

Attack Vectors	Vulnerability	Example Attacks	SEV	SEV-ES	SEV-SNP	CSV v1	CSV v2	CSV v3	SGX	TDX	CCA
Memory Encryption Abuse	Vulnerable Tweak Function	SEV-Unsecure [56]	✔	✗	✗	✗	✗	✗	✗	✗	✗
	Insufficient Entropy	SEVurity [52]	✔	✔	✗	✗	✗	✗	✗	✗	✗
	No Freshness	Software-based Ciphertext Side Channels [35, 38]	✓	✓	✔	✓	✓	✗	✗	✗	✗
	Unequal Granularity	CIPHERSHADOW	✗	✗	✗	✓	✓	✗*	✗	✗	✗
Interrupt Injection	Inject Int 80	Heckler [43]	✓	✓	✔	✓	✓	✓†	✗	✓	✗
	Inject #VC	WeSee [42]	✓	✓	✔	✓	✓	✓†	✗	✗	✗
	Inject SIGFPE	Sigy [47], Heckler [43]	✓	✓	✔	✓	✓	✓†	✓	✗	⚠
	Inject APIC Timer	Single-Steping [49, 51, 53]	✓	✓	✓	✓	✓	✓	✔	✓	⚠
Register Leakage	Exposing Registers during World Switch	SEVerESt [50]	✓	✗	✗	✓	✗	✗	✗	✗	✗
	Exposing Performance Information	SEVerESt [50]	✓	✓	✓	✓	✓	✓†	✗	✗	✗
Memory Remapping	Page Table Tampering	SEVered [39], SEVerity [40]	✓	✓	✗	✓	✓	✗	✗	✗	✗
Other Attacks	Calculation Errors in Attestation Measurement	undeSErVed [54]	✓	✓	✗	✓	✓	✗	✗	✗	✗
	Lack of protection for ASID	CrossLine [36]	✓	✓	✗	✓	✓	✗	✗	✗	✗
	Abuse of invd	CacheWarp [55]	✓	✓	✔	✓	✓	✓†	✗	✗	✗

†: Hygon mitigates end-to-end attacks by preventing Page Fault Tracking.
*: One-shot ciphertext leakage can still be exploited.

Single-Stepping [49, 51, 53]. Such attacks leverage the hypervisor's capability to inject timer interrupts, enabling control flow tracing within TEEs. SGX [49] is vulnerable to such attacks via high-resolution timers (e.g., LAPIC). Countermeasures like AEXNotify [21] introduce ISA extensions allowing enclaves to register interrupt handlers. The SEV family [53] and CCA lack interrupt frequency detection mechanisms, leaving them exposed. TDX attempts to mitigate these attacks through its trusted TDX module, which detects suspicious interrupt patterns and introduces random delays. However, recent research in TDXdown [51] reveals that these protections can be circumvented—either by manipulating CPU frequency to bypass timing checks or via the StumbleStepping side channel, which leaks instruction counts within a TD. To implement interrupt-based single-stepping on CSV, we configure the APIC timer interval from the hypervisor to precisely control guest VM execution timing. By leveraging hardware performance counters, we distinguish between three execution states: single-step, zero-step, and multi-step transitions. By adjusting the APIC intervals, we establish a reliable method for achieving consistent single-step execution.

2.3.2 Register Leakage and Manipulation

Unprotected register states during world switches pose a critical security risk, enabling both register value leakage and manipulation. TEEs employ different approaches to register protection during world switches: SEV-ES/SEV-SNP and CSV v2/v3 encrypt registers on VMEXIT to prevent direct leakage; SGX uses an encrypted State Save Area (SSA) [29] to save the register information of the enclave during context switches; TDX strengthens isolation by executing its module in a hardware-secured environment, guaranteeing encrypted register state during exits [19]; while CCA adopts a stricter model where the RMM mediates all context switches, enforcing complete register isolation from untrusted software [16].

SEVerESt [50]. Our experiments reveal that this vulnerability persists in CSV v1 due to its exposure of general-purpose registers in plaintext during VM context switches. Like SEV-ES, CSV v2 resolves this security gap through register state encryption, effectively preventing such instruction inference attacks. Moreover, like SEV-ES where applications fingerprints can be extracted through Instruction-Based Sampling (IBS) [50], CSV v2/v3 also provides performance monitoring mechanism with the capability of monitoring TLB and cache misses, retired instruction counters, context switches, bad speculation, and branch miss events, among others, making CSV v2/v3 susceptible to such attacks. The collection of application fingerprint information requires details such as page addresses. In CSV v3, the lack of page fault assistance prevents attackers from obtaining such information, thereby limiting these types of attacks.

2.3.3 Memory Remapping Attacks

In SEV and SEV-ES, the hypervisor retains full control over the NPT, enabling it to monitor or manipulate VM memory mappings undetected [39, 40]. SEV-SNP addresses this limitation by introducing hardware-enforced page table validation via the RMP, which prevents unauthorized remapping

and enforces strict page access control [44]. Similarly, SGX relies on the hardware-protected Enclave Page Cache Map (EPCM) to validate enclave pages, ensuring software cannot access or modify it [22]. In contrast, TDX and CCA protect page tables by virtualizing or relocating them within the TEE context, preventing the untrusted host from directly inspecting or modifying guest address mappings [16, 30].

SEVered and SEVerity [39, 40]. Our experiments show that on both CSV v1 and v2, the hypervisor can modify the NPT to successfully perform memory remapping attacks. Specifically, we implemented a program in the guest VM that reads data in a busy loop from a buffer located on a specific memory page. In the hypervisor, we modified the page table entry for the buffer's GPA in the NPT to point to a different physical page. We observed that the output of the test program changed immediately after the remapping. In CSV v3, the NPT is encrypted and no longer under the control of the hypervisor; hence, such attacks can no longer be successfully carried out.

2.3.4 Other Known Attacks

CacheWarp [55]. The SEV processor series allows hypervisors to utilize the privileged `invd` instruction, which invalidates dirty VM cache lines without writing them back, causing VMs to use stale data. SEV-SNP fixes this vulnerability by updating hardware patches to disable the "enable invd behavior" MSR. To test this vulnerability on CSV v1, v2, and v3, we pinned the victim VM and the attacker program to different physical cores so that they share the Last Level Cache (LLC). First, to avoid system crash, the attacker program executes `wbinvd` to write back all cache lines in the LLC, as the LLC might contain dirty data from the kernel. Then in the victim VM, a program runs in a busy loop; in each iteration, it reads the value from a fixed address, increments it by 1, writes it back, and then evicts its copy in the private cache to LLC via cache `Priming`. Right after the victim's `Priming` and before its subsequent read, the attacker program executes `invd` to invalidate the LLC, without writing back dirty data. In our experiments, we observe that the victim reads a stale data in the next iteration. This demonstrates that CSV v1-v3 are potentially susceptible to CacheWarp attacks. However, in CSV v3, the absence of page fault assistance will make a successful end-to-end attack much harder.

undeSErVed [54]. In SEV and SEV-ES, rearranging 16-byte-aligned data in OVMF during VM launch preserves the launch digest, enabling secret leakage [54]. Our investigation of CSV v1/v2 reveals that while CSV replaces SEV's SHA-256 with SM3 for launch digest calculation—hashing OVMF appended with an optional hash table (kernel, initrd, launch commands)—it inherits the same vulnerability: modifying 16-byte-aligned OVMF data leaves the digest unchanged, demonstrating that CSV merely substitutes SM3 without fixing SEV's flawed measurement scheme. In CSV v3, altering the 16-byte-aligned OVMF data will prevent passing the `LAUNCH_FINISH` phase, indicating that this vulnerability has been fixed.

CrossLine [36]. SEV and SEV-ES are vulnerable to ASID (Address Space Identifier) spoofing attacks due to the lack of hardware-enforced ASID protection [36]. During VMEXIT, an attacker can maliciously modify the ASID and associated memory mapping information, enabling a compromised CVM to impersonate a victim CVM and access its memory using the victim's encryption key. While SEV-SNP addresses this issue through hardware-based memory ownership verification and mapping integrity protection, our experimental results demonstrate that CSV v1 and v2 remain susceptible - a malicious hypervisor can still arbitrarily manipulate ASIDs of protected CVMs. CSV v3 ensures the trust and integrity of ASIDs and their corresponding memory through SPOT, thereby preventing this attack.

3 Memory Encryption Taxonomy in Commodity TEEs

To comprehensively assess the security implications of these vulnerabilities and their relationship to other weaknesses in the memory encryption architecture, in this section, we systematically analyze the memory encryption mechanisms in different TEEs, starting from three design considerations: encryption freshness, encryption granularity, and integrity protection. We categorize designs with different encryption characteristics into four security levels and landscape the attack threats faced by each security level, as shown in Figure 3. This allows us to assess the security of memory encryption in mainstream TEE products, thereby evaluating the security of CSV memory encryption.

3.1 Security Design Choices

Hardware-assisted memory encryption engines encrypt and decrypt data traversing the memory bus on the fly. This ensures that only plaintext is available inside the CPU for computation, while ciphertext is stored in DRAM to protect data confidentiality. We categorize the key design choices in hardware-assisted memory encryption that influence its security properties into three main aspects: **encryption freshness**, **encryption granularity**, and **integrity protection**.

Encryption Freshness. For performance reasons, hardware memory encryption engines primarily support symmetric encryption algorithms such as AES or SM4. The former is the acronym for the Advanced Encryption Standard [27], which was officially adopted by the National Institute of Standards and Technology (NIST) of the United States in 2001. Most TEE designs use AES as their memory encryption algorithm, including Intel SGX, Intel TDX, ARM CCA, and AMD SEV.

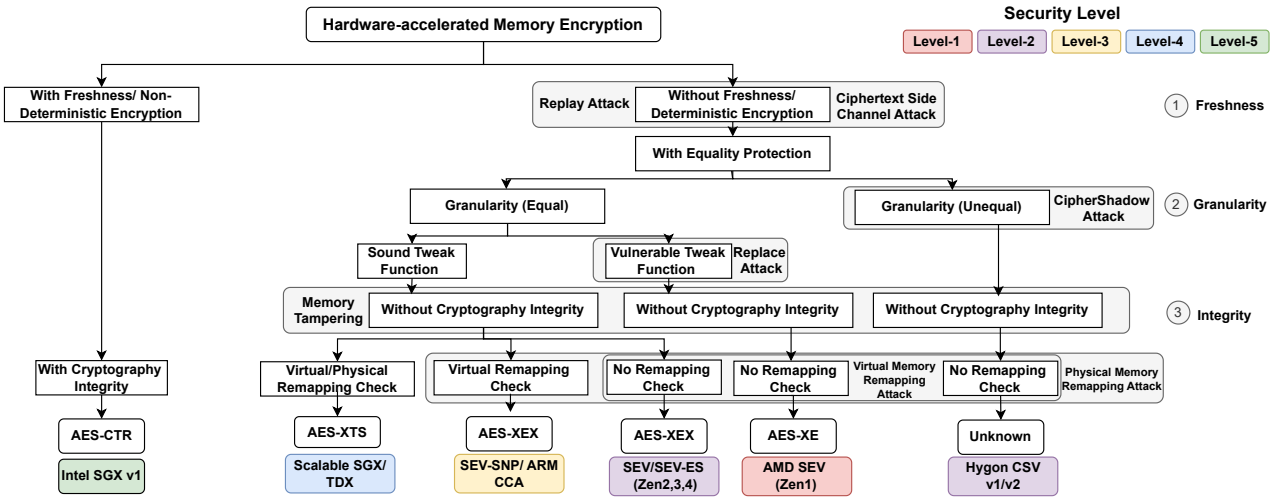


Figure 3: A Taxonomy of Hardware Memory Encryption.

SM4 [11] was designed by China's State Cryptography Administration and is widely used in a variety of encrypted communication and e-Commerce contexts within China. The SM4 algorithm has a key length and block length of 128 bits and operates in a structure of 16 rounds of iteration. Hygon CSV chooses SM4 as its encryption algorithm [4].

With a specific encryption key, symmetric encryption algorithms (AES and SM4) always deterministically transform a plaintext block into the corresponding ciphertext block. However, with hardware memory encryption, this leads to deterministic encryption, resulting in a lack of the freshness of encryption. To address this, equality protection is essential to ensure that the same plaintext in different memory locations does not produce identical ciphertext.

Encryption Granularity. Granularity refers to the size of the encryption block and the size of the tweak window. As shown in Figure 3, when these sizes are equal, it is referred to as *equal granularity*. On the other hand, if the sizes are different, it is classified as *unequal granularity*.

- *Encryption Block Size:* In the context of memory encryption, a 16-byte block is typically used to align with the requirements of block cipher systems such as AES/SM4, where 16-byte (128-bit) encryption units are employed.
- *Tweak Size:* The tweak function mitigates deterministic encryption by incorporating physical addresses to produce unique ciphertexts for identical plaintexts. The tweak size — i.e., the number of entropy bits used—directly impacts the level of diversification. While a full 128-bit tweak is unnecessary, vendors can select smaller sizes based on performance-security trade-offs.

Integrity Protection. To ensure the integrity of encrypted memory, different TEEs adopt distinct design choices:

- *Cryptography Integrity:* Cryptographic integrity protection ensures the integrity of data by incorporating Message Authentication Code (MAC) values. Attackers lacking the integrity key for computing the MAC will cause a mismatch in the MAC values, thereby enabling the detection of malicious tampering with the ciphertext.
- *Remapping Checks:*
  Remapping checks are used to verify whether confidential memory is mapped to crafted memory addresses through either virtual memory-remapping or physical memory-remapping attacks, thereby hindering the implementation of potential memory tampering and replay attacks. Virtual remapping checks are used to inspect virtual memory-remapping attacks, where attackers manipulate the virtual address mappings of a TEE instance, remapping the virtual memory to meticulously crafted memory locations, thereby altering the data within the confidential instance. Physical remapping checks are used to inspect physical memory-remapping attacks, where the physical addresses of the TEE instance are remapped to 'fabricated' physical addresses (i.e., aliasing), allowing attackers to observe and modify the TEE instance's memory by reading and writing to these physical addresses.

3.2 Security Levels of Memory Encryption

Based on memory freshness, memory integrity, and known attacks, we summarize the following five security levels of memory encryption.

- **Level-1:** This level implements a basic memory encryption mechanism, but there are flaws in the design of the encryption scheme.
- **Level-2:** This level provides encryption without cryptographic integrity protection and lacks remapping checks, allowing both virtual and physical memory-remapping attacks, enabling memory tampering and replay attacks.

- **Level 3:** This level introduces a virtual remapping check that prevents tampering with memory of TEE instances through virtual memory-remapping attacks, but it is still vulnerable to tampering through physical memory remapping.
- **Level-4:** This level incorporates both virtual and physical remapping checks to prevent virtual and physical memory-remapping attacks.
- **Level-5:** This level offers a memory encryption algorithm that includes both freshness and integrity, capable of defending against replay and memory tampering attacks from both software and hardware attackers at the cryptographic level.

As shown in Figure 3, different levels of memory encryption are vulnerable to varying types of attacks. For example, memory encryption without freshness faces attacks from ciphertext side-channel [35, 38] and replay attacks [28, 39, 40]. Unequal encryption granularity leads to the CIPHERSHADOW attack that we explore in-depth in this work. The absence of remapping checks exposes the confidential memory to memory-remapping attacks [23, 39, 40]. Recent studies have shown that physical attacks on the unencrypted address bus of SGX can lead to off-chip side-channel attacks [34]. However, the lack of encryption on the address bus falls outside the scope of discussion for TEE memory encryption mechanisms.

3.3 Design Spectrum of Memory Encryption

AMD SEV (Zen 1) (Level-1, XE With Flaws in the Tweak Function). To address vulnerabilities caused by deterministic encryption, AMD initially used XOR-and-Encrypt (XE) mode in the first generation of SEV. However, this mode was later proven unsecure with a vulnerable tweak function [37, 56], resulting in the leakage of the tweak value through the collection of plaintext-ciphertext pairs.

AMD SEV/SEV-ES (Zen 2,3,4) (Level-2, XEX Without Remapping Checks). XOR-Encrypt-XOR (XEX) provides a more secure tweak function but has vulnerabilities on some SEV platforms due to its use of limited 32-bit entropy, allowing brute-force attacks on the tweak constants and performing ciphertext replacement attacks [52]. The XEX tweak function uses physical addresses as input and lacks freshness, which causes identical plaintexts at the same physical address to produce the same ciphertext. This makes it vulnerable to ciphertext side-channel attacks [35, 38]. Furthermore, hypervisor manipulation of NPT without remapping checks can lead to virtual memory-remapping attacks, causing replay attacks to be implemented. [39, 40].

AMD SEV-SNP and ARM CCA (Level-3, XEX With Virtual Remapping Checks). SEV-SNP uses the same XEX mode for memory encryption as SEV and SEV-ES, which lacks the property of freshness. Therefore, it is also susceptible to ciphertext side-channel attacks. The latest version of SEV-SNP eliminates such attacks through Ciphertext Hiding [15], which restricts the hypervisor's access to VM ciphertext by matching the ownership of accessed memory. Additionally, SEV-SNP introduces a virtual remapping check mechanism that uses RMP to verify the integrity of GPA to HPA mappings during page table walks. Recent studies have shown that although RMP can defend against virtual memory remapping, its unencrypted data storage leaves it susceptible to modification by physical memory-remapping attacks. Once altered, the platform becomes vulnerable to virtual memory-remapping and ciphertext side-channel attacks again [23].

Similarly, Arm CCA [16] employs AES-XEX or QARMA-based memory encryption without MACs, thereby lacking both integrity and freshness guarantees. The Granule Protection Table [24], which is entirely located in external memory, is similar to SEV-SNP's RMP and can only defend against virtual memory-remapping attacks. As a result, CCA remains vulnerable to physical memory-remapping attacks.

Intel TDX/Scalable SGX (Level-4, XTS With Virtual and Physical Remapping Checks). Intel TDX [19, 41] and Scalable SGX [31] both employ AES-XTS [25] mode for memory encryption. While both XEX and XTS involve two XOR operations, XTS includes a tweak value derivation function and without freshness, thus, is vulnerable to ciphertext side-channel attacks. But, Scalable SGX and TDX both utilize memory access control, which can prevent other programs from accessing confidential memory, thereby mitigating ciphertext side channels. SGX leverages DRAM's ECC as ownership to isolate access from outside programs, and uses ACTM to isolate access from other confidential instances. When attackers attempt to access confidential memory through remapped physical addresses, they will be detected based on differences in permissions, thereby achieving physical remapping checks. TDX uses a remapping check similar to Scalable SGX and provides optional cryptographic integrity protection mechanisms. It uses ciphertext, tweak value, MAC key, and TD-owner extracted from ECC to generate a 28-bit MAC value to prevent memory tampering attacks, thereby ensuring memory integrity.

Intel SGX v1 (Level-5, AES-CTR With Merkle Tree). The AES-CTR encryption mode used by Intel SGX v1 introduces a notion of freshness via counters for each block, which inherently provides resistance to ciphertext side-channel and replay attacks. By constructing a binary tree structure and comparing root hashes, the integrity of the entire confidential memory can be verified and prevent memory tampering attacks. However, the use of Merkle trees presents certain limitations. They necessitate extra memory to store the computed hashes, which can be problematic in environments with limited memory capacity. Moreover, as the hash tree expands, the verification process becomes more time-consuming due to the increased depth of the tree. Thus, the implementation of the Merkle tree imposes a significant maintenance overhead.

Consequently, SGX v1 is restricted to support small memory sizes [22, 48].

Hygon CSV v1/v2 (Level-2, SM4 With Tweak-derived Keys): CSV employs the SM4 encryption algorithm with an encryption key derivation mode that lacks freshness, making it susceptible to ciphertext side-channel and replay attacks. Additionally, the absence of remapping checks and cryptography integrity protection makes it equally vulnerable to both virtual and physical memory-remapping attacks. The key derivation strategy used by CSV implements unequal granularities, leading to new security issues for CSV. Specifically, this characteristic of unequal granularities results in a repeated tweak value within a 64-byte block, leading to more severe replace attacks, as the contents of the adjacent four 16-byte chunks can be interchanged. This also results in additional information leakage, revealing whether the four adjacent chunks contain the same plaintext. We refer to these enhanced attack vectors resulting from the repeated tweaks as the CIPHERSHADOW attack, and further discuss the security impact of the CIPHERSHADOW attack later in this paper. Notably, the root cause of these new vulnerabilities in CSV is its misaligned encryption granularity and tweak granularity, not the tweak function itself.

4 CIPHERSHADOW Vulnerability Analysis

In this section, we present CIPHERSHADOW attacks, which exploit the newly discovered vulnerability in CSV's memory encryption scheme.

4.1 Threat Model

Our threat model considers attacks from both privileged software and adversaries with physical accesses to the TEE platform. Specifically, we assume the software adversary can execute privileged instructions, manage the scheduling of confidential VMs, and access the ciphertext of the encrypted guest memory with either read-write permissions or read-only permissions. The adversary can further leverage controlled-channel attacks (e.g., page fault or interrupt-based) and cache side-channel attacks to extract sensitive information from the protected VM, including identifying memory pages containing critical functions or data structures. For physical attacks, we assume the adversary has access to the server hardware and can carry out low-level hardware attacks, such as intercepting memory bus traffic or extracting DRAM contents via a one-time cold boot attack. We do not consider Denial-of-Service (DoS) attacks, as they are outside the TEE's confidentiality guarantees and are generally detectable by the TEE owner.

4.2 CIPHERSHADOW Attack Overview

Next, we provide an overview of how CIPHERSHADOW can hijack the control flow of confidential VMs and thus breach their confidentiality and integrity.

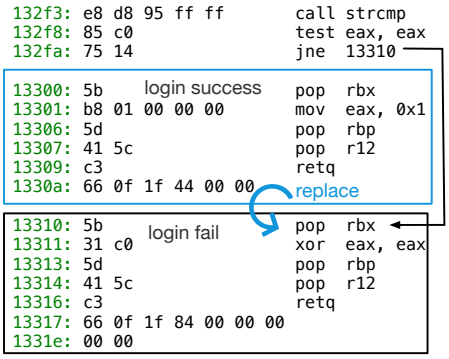


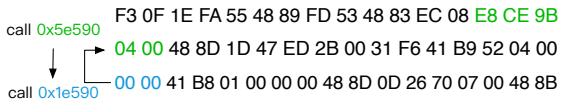
Figure 4: Password authentication assembly code of OpenSSH. The execution of "login success" or "login fail" will be determined by the result of the `strcmp` function, which indicates whether the input password matches the correct one.

CIPHERSHADOW Example. As shown in Figure 4, The machine code for "login failed" and "login successful" corresponds to different 16-byte chunks within the same 64-byte block. Due to the encryption mode employed by CSV (see Section 2.2), this arrangement is also reflected in their encrypted ciphertext. A malicious hypervisor can manipulate the login verification process by replacing the ciphertext chunk that contains the code for "login failed" with the chunk that corresponds to the code for "login successful". Consequently, even if an incorrect password is provided by an illegal user, the server will grant permissions, enabling unauthorized and malicious access.

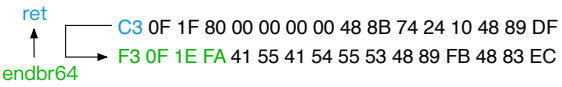
4.2.1 CIPHERSHADOW Gadgets

CIPHERSHADOW is exploitable with certain code or data patterns. A **gadget** refers to a 16-byte ciphertext memory chunk that an adversary can use to replace an adjacent ciphertext chunk. This substitution should allow the attacker to manipulate the program's execution successfully without causing the program to crash. Such a ciphertext replacement can be used by an adversary to compromise either the control flow or the data flow of a program. Control flow gadgets alter the program's execution flow, while data flow gadgets can modify data handling by changing operand values or data structures.

Control Flow Gadgets. Control flow gadgets allow the adversary to add new instructions or alter the operands of existing instruction, by replacing memory chunks in code pages. Specifically, modifications and additions to control flow instructions such as `jmp`, `call`, and `ret` can affect the correct execution of instruction sequences, potentially altering the result of the correct function call, Figure 5 illustrates examples of control flow gadgets.



(a) `Call` Gadget: Replacing the second 16-byte chunk in a 64-byte block with the third results in a function call to a different address.



(b) `Ret` Gadget: Replacing the second chunk at the beginning of the next function with one that includes a `ret` instruction causes an immediate return.

Figure 5: Illustration of gadget manipulations by modifying 16-byte chunks in code blocks.

Data Flow Gadgets. Data flow gadgets pose a serious threat to the program's data flow, as they have the potential to hijack memory or register states, leading to the processing of incorrect data. They can be generated by modifying the ciphertext of the four adjacent 16-byte executable text or data segments. The altered instruction sequence may change the propagation of data in registers or memory, thereby disrupting the program's data flow. Similarly, replacing within data memory can cause altered data to flow, resulting in malicious modifications to function parameters, return registers, and the memory regions of local and global variables.

4.2.2 One-time Ciphertext Leakage

The CSV's reuse of the tweak function within a 64-byte block not only facilitates attacks due to ciphertext block replacement, but also enables new vectors for information leakage from one-time dump of the ciphertext memory. This one-time ciphertext leakage allows attackers to extract sensitive information of the VM, as discussed in Section 6.3.

5 CIPHERSHADOW Gadgets Scanner

This section presents a scanner for detecting CIPHERSHADOW gadgets in production binaries and evaluating their exploit potential in sophisticated attack scenarios.

5.1 Overview

The workflow of the scanner is illustrated in Figure 6. The scanner analyzes executables at the function level, where the scanner first isolates the functions of the executable and then analyzes each of *moving windows* within a function. A moving window is a 64-byte aligned memory block. Each window comprises four *movable* chunks, which are 16-byte aligned blocks. Shifting or duplicating chunks within a window can still yield valid and meaningful plaintext, although certain plaintext, when decoded as instructions, may cause application to crash. To thoroughly explore potential vulnerabilities, we apply a ciphertext replacement strategy to generate possible modified functions.

The analysis module comprises static control flow analysis and static data flow tracing, which are used to extract the control flow graph (CFG) and the data flow graph (DFG) from the original and modified functions. Subsequently, during different tests, the CFG and DFG of the original functions are compared with those of the modified functions. If differences are detected–a screening procedure performed by a *unusable gadget filter*–that excludes modifications likely to cause crashes are performed. The remaining gadgets are flagged as potentially exploitable.

5.2 Methodology

Reducing Search Space. There are 4^4 possible replacements for movable chunks within a 64-byte moving window. However, complex replacement operations are likely to cause crashes, resulting in a substantial amount of unfruitful exploration. To reduce the search space, we only consider replacing only a single chunk. For complex transformations that replace multiple chunks will significantly alter the control flows of the program, ultimately causing crashes. As our principle is to locate gadgets that can potentially threaten program security without crashing its execution, we only replace a single chunk with each of the other chunks within a moving window, resulting in only 12 (i.e., 3×4) candidate gadgets per window. Our analysis reveals minimal downstream impact, with newly generated instructions differing from the original by only 4.27 instructions on average.

Static Data Flow Tracing. Static data flow trace checks if a chunk replacement can lead to changes in data flow-related instructions (such as `mov`, `add`, `xchg`, etc.). We first perform a static data flow analysis from the entry point of the original function, using symbolic values for initial register and memory values. We monitor data flow-related instructions and update the register and memory sets based on the semantics of each instruction until the end of the function. This allows us to obtain the DFG, including program memory and register states at every address within the function.

After replacing the movable chunk and obtaining the modified function, we locate the new instructions generated by the replacement. If these new instructions alter the original data flow, we use the register and memory values obtained from the original function at this address as the initial state set. Then we perform a static data flow analysis from the new instruction using the same process as described above to obtain the modified DFG. Afterward, these two data flow sets are compared, with particular focus on whether the data flow

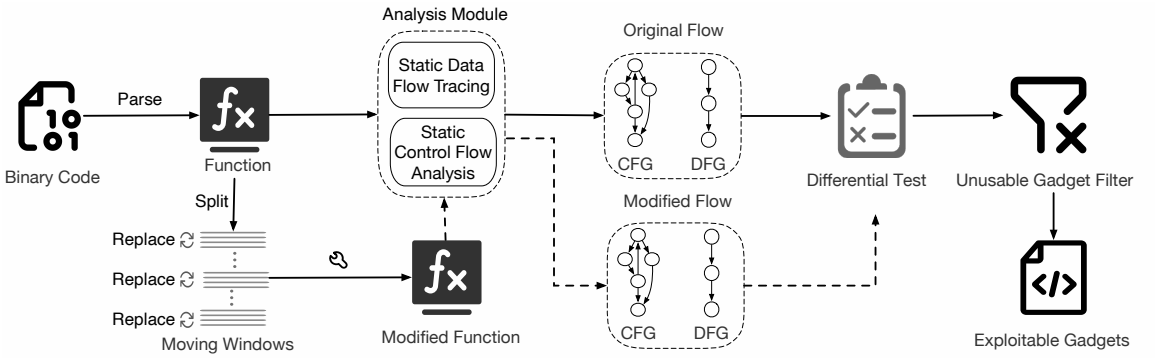


Figure 6: Key components and workflows of CIPHERSHADOW gadget scanner.

of the relevant registers changes during the call, conditional jump, and return instruction. If so, the new instructions are recorded as a potential data-flow gadget.

Static Control Flow Gadget Detection. Static analysis focuses on specific control flow instructions, including `jne`, `je`, `jmp`, `call`, and `ret`. The process begins with the generation of a control flow graph of the original function. After each replacement of the movable chunk, the original instructions are overwritten by the newly generated instructions, which may introduce new control flow instructions or modify existing ones. A new control flow graph of the modified function is generated and compared with the original one. If there are differences in the control flow graph, the altered instructions are marked as potentially exploitable control flow gadgets.

Unusable Gadget Filter. Replacing a movable chunk in a moving window may still generate instructions that ultimately cause program to crash, even when generating gadgets triggering invalid control or data flows. To eliminate useless gadgets, we define the following rules for filtering of usable gadgets.

- **Jump to a non-executable memory address.** The generated jump instructions have great randomness in their targets, including illegal virtual addresses, unmapped address segments, program address space, shared library address space, and kernel address space. Address space layout randomization (ASLR) [46] and page table permissions restrict our ability to identify valid jump targets beyond the current program's memory mapping. Consequently, we filter gadgets to retain only those targeting executable pages within the program's address space.
- **Generate Illegal Instruction.** We filter gadgets generating illegal instructions that violate the format of the x86 ISA.
- **Access to an illegal memory address.** We implement strict gadget filtering that excludes any instructions causing memory access violations, whether through permission errors (R/W/X), privilege level transgressions, or invalid address accesses, ensuring only viable exploitation candidates remain for analysis.

Results. The scanner is implemented on top of angr [45]. Table 2 shows that there are numerous potential exploitable gadgets in security-related functions. We can observe that replacing just a single 16-byte chunk is highly likely to change the program's control flow or data flow; moreover, most of these will produce illegal instructions and illegal memory accesses, leading to crashes. Manual efforts are needed afterward for exploiting the gadgets for actual attacks. The main performance overhead of the scanner is influenced by the function size and the number of data flow gadgets obtained. When newly added or deleted instructions affect data propagation, complete static data flow tracing will be conducted from the modified instructions to the function return, which accounts for most of the overhead of the scanner. Additionally, since the analysis is performed on binary files, angr uses the IR to simplify the handling of complex assembly instructions, and the translation process will also affect the cost of program analysis. On average, the processing time for one moving window is 4.2s.

6 CIPHERSHADOW Attack

In this section, we show case the power of CIPHERSHADOW attacks. First, we show CIPHERSHADOW can be leveraged to collect application's *page fingerprints* (Section 6.1), thereby revealing the physical addresses of the targeted applications to aid end-to-end attacks. Second, we show CIPHERSHADOW can compromise confidential VMs by enabling unauthorized logins (Section 6.2.1). Third, we show even without software read/write permissions, adversaries with one-shot ciphertext leakage can still leverage CIPHERSHADOW to disclose sensitive information such as training datasets of AI applications (Section 6.3).

All experiments were carried out on two Hygon CPUs, including `Hygon C86 5285` with 16 CPU cores that support CSV v1, and `Hygon C86 7390` with 32 CPU cores that support both CSV v1 and v2.

Application	Function Name	#64-byte Block	Time	Potential Control Flow	Potential Data Flow	U1	U2	U3	Control Flow	Data Flow
Sshd	do_authenticated	21	57.20s	228	158	54	79	105	95	53
Mysql-Server	FSE_optimalTableLog_internal	5	22.37s	11	12	0	8	7	3	5
Nginx-Server	resolve_srv_names	10	23.85s	68	48	16	24	31	28	17
Redis-Server	genericZrangebyscoreCommand	19	171.37s	212	187	94	54	142	66	43

Table 2: Numbers of gadgets in security-sensitive functions. U1, U2, and U3 represent the numbers of filtered gadgets that will cause a crash due to (1) jump to a non-executable memory address, (2) generate illegal instruction, and (3) access to an illegal memory address, respectively. *Potential* reflects the number of gadgets before filtering. The size of each function is represented as the number of 64-byte blocks in the function code.

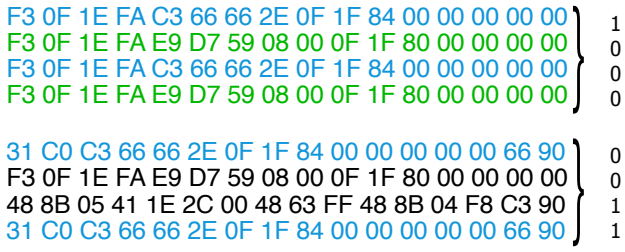


Figure 7: An example of the identical 16-byte chunk within a 64-byte memory block. The appearance of the identical 16-byte chunk may be caused by function prologues or epilogues, loop unrolling, and compiler optimization.

6.1 Application Fingerprinting

Our analysis demonstrates that repeated 16-byte ciphertext patterns within 64-byte memory blocks can function as distinctive fingerprints, enabling adversaries to precisely locate physical pages. This achieves equivalent physical address inference capabilities to existing page-fault-based [37] and interrupt-based [49, 53] side-channel techniques.

Fingerprints Calculation As shown in Figure 7, the sequence of identical 16-byte chunks within 64-byte memory pages can be used as a feature to generate a fingerprint, which is the same generated by both encrypted and plaintext pages.

Within each 64-byte block, the four 16-byte chunks can exhibit 15 distinct patterns: (1) all chunks unique, (2) six variations with two identical chunks and two distinct ones, (3) three variations with two identical pairs, (4) four variations with three identical chunks, and (5) all chunks identical. We encode each pattern using a 4-bit identifier. Therefore, 4 bits can be used to encode the similarity patterns for each 64-byte block. By concatenating 64 consecutive 4-bit identifiers from a 4KB memory page, we generate a compact 32-byte fingerprint.

Evaluation of Fingerprint Uniqueness. To evaluate whether identical 16-byte ciphertext patterns can uniquely identify memory pages (i.e., exhibit minimal fingerprint collisions), we conducted extensive analysis across multiple common applications. Our evaluation assessed fingerprint uniqueness both intra-program (within individual programs) and inter-program (across different programs). As shown in Table 3, our results reveal that: (1) only 26.9% of all pages demonstrate program-unique fingerprints on average, with higher collision rates occurring within programs than between them; (2) this stems from 45.8% of pages (on average) containing no identical 16-byte blocks, generating non-distinctive fingerprints; and (3) pages containing repeated 16-byte blocks show significantly lower collision probabilities, confirming their stronger discriminative power as fingerprints.

Program	Pages	Unique (%)	Unique vs. ssh	Unique vs. nginx	Unique vs. mysql	Unique vs. qemu
ssh	217	15.28	—	13.89	12.50	10.65
nginx	293	21.84	20.82	—	20.48	16.38
mysql	1880	40.11	39.89	40.00	—	39.31
qemu	3904	30.46	30.33	30.20	30.20	—

Table 3: Evaluation of intra-program and -inter-program uniqueness of page fingerprints. *Unique* represents the percentage of total pages that possess a unique fingerprint compared to other pages.

6.2 Malicious Login against OpenSSH

We selected exploitable gadgets from the scanner results in Section 5 to achieve complete exploitation. We showcase two different attack patterns, which allow bypassing functions and hijacking library functions by modifying the Procedure Linkage Table (PLT) or Global Offset Table (GOT), leading to malicious login against OpenSSH running in the CSV.

6.2.1 Critical Function Bypass

We present a novel attack methodology that hijacks function returns using a single gadget, enabling malicious control-flow manipulation to force specific return values. This poses critical security risks to sensitive operations including password validation and privilege checks. As a concrete demonstration, Figure 8 illustrates our successful bypass of OpenSSH's `do_authenticate` function - the core authentication handler responsible for establishing secure communication channels post-authentication.

Figure 8a showcases the assembly code at the entry of the `do_authenticate` function, and the threat code snippet located at `0x27580`, which can write to `rax` register and call

another address. We replace the first movable chunk in the entry 64-byte block with the fourth chunk, leading to hijacking of the control flow and data flow. The altered program flow is shown in Figure 8b, where the `do_authenticate` function is initially called at address `0xea63`.The address of the next instruction is then pushed onto the stack. When the execution reaches `0x27588`, the memory chunk replacement leads to the addition of a call instruction. The next address is pushed onto the stack and jumps to the target address `0x90830`. Subsequent `pop` instructions move the pushed address from top of the stack to `r15` which is a useless register before return, restoring the stack state to its initial state when the function was invoked. As a result, the function will directly return, bypassing the critical functionality of the original function.

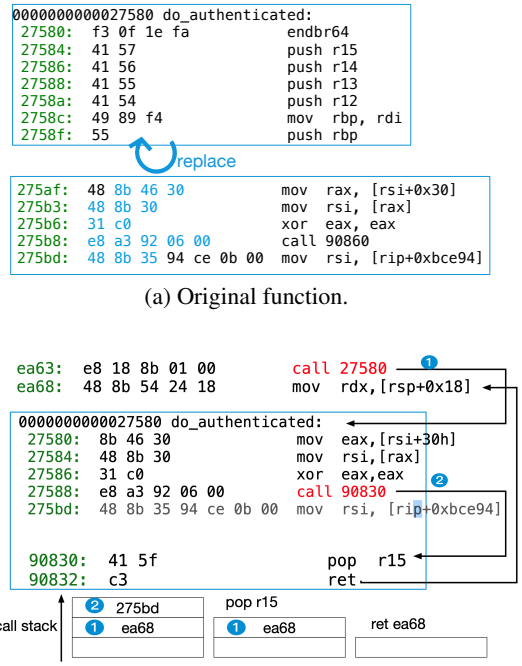
This attack highlights two key characteristics of CIPHERSHADOW: (1) the non-deterministic nature of control flow manipulation, and (2) the varied impacts on program behavior. The attack operates by injecting new instructions at the function entry point that simultaneously alter both data flow (through sensitive register modification) and control flow (via redirection to other function endpoints). Remarkably, subsequent instructions naturally mitigate the injected instructions' side effects while still achieving the attacker's goal of function logic bypass and corruption. However, this attack method exhibits inherent limitations in return value manipulation, as its effectiveness depends on: (1) the data flow alterations induced by injected instructions, and (2) the runtime state of registers and memory. In our OpenSSH case study, for example, the injected `xor` instruction clears `eax`, forcing a return value of 0. Furthermore, attackers can combine multiple gadgets to create more sophisticated exploitation chains, potentially achieving broader malicious capabilities through carefully sequenced operations.

6.2.2 Library Function Hijack

This attack combines multiple gadgets to hijack the invocation of library functions in a dynamically linked program by replacing its 16-byte aligned PLT entry with one of three adjacent entries, causing the manipulation of the return values of those library functions.

The `sys_auth_passwd` function in OpenSSH is used to verify the login user's permissions. It uses the library function `strcmp` to compare the input password with the correct one. When the passwords match, `strcmp` returns 0. In a dynamically linked program, the call to the `strcmp` function first executes the trampoline code in the PLT, and then executes the actual function in the dynamically linked library.

We have found two gadgets, the first gadget replaces `strcmp` entry with `DSA_free` entry in PLT and another in `cfsetispeed` alters the ciphertext chunk at `0xeb210` to the chunk at `0xeb230`. Combining these two gadgets can hijack `strcmp` to `cfsetispeed` and return 0. The specific execution path after exploitation is depicted in Figure 9. ❶ Retrieves



(a) Original function.

(b) The control flow after replacement.

Figure 8: Bypassing `do_authenticated()` in OpenSSH.

the exactly `strcmp` address from the PLT. ❷ Jumps to the Libc library to call the `cfsetispeed` function. ❸ Clear the `eax` register and immediately return. After the execution of `strcmp`, the test instruction will set the Zero Flag, allowing jump to the successful login.

The successful implementation of CIPHERSHADOW's end-to-end attack relies on accurately pinpointing the replacement target of a 64-byte block in physical memory. To achieve this, we combined page-fault tracking and application fingerprinting as described in Section 6.1 to locate the physical page containing the target block.

First, we perform offline fingerprint calculation on the plaintext SSH binary file used in the VM to obtain the fingerprint information of the target page. Subsequently, we use system-wide code page fault tracking by clearing the execution bit of all pages. When the adversary attempts to use SSH for logging in, the attacker will generate the corresponding fingerprint for each encrypted page that triggers a page fault and compare it with the fingerprint obtained from the offline calculation. If they match, it confirms successful location, thereby enabling the calculation of the page-internal offset of the target 64-byte block to carry out the aforementioned attack process.

It is important to note that the target page's fingerprint in the attack may not be unique within the application, meaning that other pages may share the same fingerprint, leading to difficulties in accurately locating the target page during the attack. To address this, we have incorporated more granular register change information used in CacheWarp [55] to en-

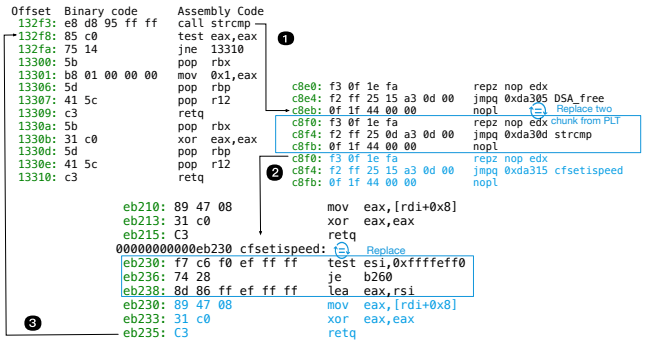


Figure 9: *sys_auth_passwd* function hijacking.

hance the uniqueness of our page fingerprints. Specifically, we analyze the register patterns changed by the instruction sequences of colliding pages to distinguish between the two pages. When tracking a physical page with the same fingerprint as the target page, we use single-step execution to capture the register change pattern. If it is similar to that analyzed offline, it confirms a match to the target physical page.

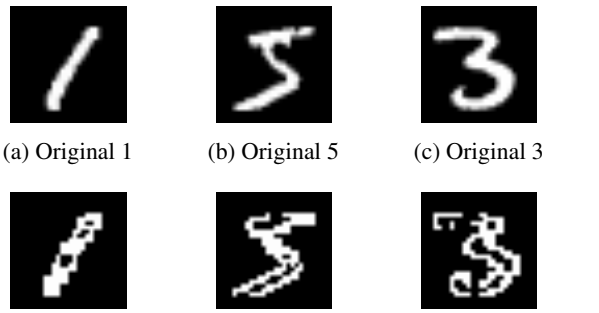
In this attack, the first target PLT page does not have a unique fingerprint, while the second gadget's page does. Therefore, we use page fingerprinting and single-step execution (discussed in Section 2.3.1), to precisely locate the `call strcmp` instruction. Afterwards, we utilize page faults to track the physical page corresponding to the PLT, modify the PLT entry corresponding to `strcmp`, and then employ page fingerprinting to identify the memory page containing the second gadget, thereby achieving the entire attack. Across 20 repeated attack trials, we accurately identified the target memory page and successfully logged in each time.

6.3 Inference Attacks Due to One-Shot Ciphertext Leakage

In addition to enabling active corruption of confidential VMs by replacing neighboring ciphertext chunks, CSV's encryption mode introduces a distinct form of information leakage attack. Specifically, an adversary can perform a one-shot memory bus snooping or cold-boot attack to monitor the ciphertext memory and infer whether two 16-byte plaintext chunks within the same 64-byte memory region are identical.

This vulnerability originates from CSV's deterministic encryption scheme operating within fixed-size memory windows. While alternative encryption modes—including AES-CTR (Intel SGX v1), AES-XTS (Intel TDX, Scalable SGX), and AES-XEX (ARM CCA/AMD SEV)—provide inherent resistance to such attacks, CSV v3's memory access restrictions still fail to fully mitigate the risk against physical adversaries.

Attack Setup. To demonstrate real-world impact, we show how a single ciphertext observation can enable reconstruction



(a) Original 1 (b) Original 5 (c) Original 3

(d) Recovered 1 (✓) (e) Recovered 5 (✓) (f) Recovered 3 (✗)

Figure 10: Recovered 28×28 image from the MNIST training dataset. ✓ indicates that the model's predictions for the original image and the recovered image are the same. ✗ indicates that the predictions are different.

of machine learning training data. When a protected VM trains a computer vision model, CIPHERSHADOW exploits ciphertext repetition patterns to partially recover input images, despite memory encryption.

As a concrete example, we consider a realistic training scenario involving a Multi-layer Perceptron Classifier (MLP-Classifier) implemented in Python and trained on the MNIST dataset—a benchmark of grayscale images of handwritten digits. Each image is 28×28 pixels, with sparse content: background pixels are mostly zero, while digits appear as clusters of non-zero grayscale values. During training, the input images are loaded into contiguous memory as default `numpy.ndarray` structures. As a result, the memory layout of the training data consists of a densely packed buffer of pixel values, where each image occupies $28 \times 28 \times 4$ bytes (4 bytes per float32 pixel) and is stored sequentially in memory. This setup mimics a typical in-memory training pipeline and allows analysis of its memory footprint.

Attack Results. A one-shot ciphertext leakage reveals contiguous 64-byte memory regions containing repeated 16-byte ciphertext patterns. We exploit this to identify training data segments by: (1) mapping identical ciphertexts to black background pixels (i.e., 0s), and (2) assigning distinct ciphertexts to non-zero pixel values. The reconstructed image set is then evaluated by comparing predictions from the trained MLP model against original image outputs, with matching predictions confirming successful reconstruction. We tested on 5,000 reconstructed images and achieved an average recovery success rate of 77.14%. Our experiments further reveal that even for unsuccessfully reconstructed images, the numerical content remains visually discernible to human observers, as illustrated in Figure 10.

7 Countermeasures

Our discussions with Hygon indicate that CSV v3 implements read and write access control mechanisms [7], which

are sufficient to defend against CIPHERSHADOW in certain scenarios: It prevents application fingerprinting (Section 6.1) and ciphertext replacing (Section 6.2.2) from software adversaries, but they remain vulnerable to one-shot ciphertext leakage from adversaries with hardware access (Section 6.3). CSV v3 uses SPOT and encrypted NPT to protect the integrity of memory mappings, thereby enabling a virtual remapping check. However, there is no documentation describing whether it includes a physical remapping check, and we have not yet experimentally verified whether it is vulnerable to physical memory-remapping attacks. Therefore, we conjecture that the security level of the memory encryption of CSV v3 is level-3, the same as SEV-SNP.

A more effective hardware solution, however, would be to address the flawed design choice of unequal granularity in the memory encryption mechanism. By aligning the tweak function granularity with the encryption blocks, this vulnerability can be inherently eliminated, even though it may potentially increase the latency of key derivation.

8 Conclusion

This paper presents the first comprehensive security analysis of Hygon CSV. Through systematic investigation of VM-based TEE vulnerabilities on the CSV platform, we uncover a new hardware memory encryption flaw: the misalignment between tweak function granularity and encryption blocks causes ciphertext repetition within 64-byte blocks when processing identical plaintext. This architectural vulnerability enables severe security consequences including sensitive data leakage and control-flow hijacking. We formalize these threats as CIPHERSHADOW attacks, develop an automated gadget scanner for vulnerability detection, and demonstrate practical exploits in real-world deployment scenarios.

Open science

All necessary research artifacts including datasets, scripts, binaries, source code, and PoCs for validating known SEV attacks on CSV, will be made publicly available at https://doi.org/10.5281/zenodo.15614377.

Ethics Considerations

We identified the vulnerability of tweak repetition in CSV's memory encryption back in early 2023 and promptly reported it to both Hygon and Alibaba Cloud, the latter offers commercial CSV-based VM rental services. Following the initial disclosure, we engaged in multiple follow-up discussions with Hygon via video conferences. Hygon acknowledged the vulnerability in CSV v1/v2 and asked us to postpone the disclosure to prevent public concern before mitigation could be deployed. The vendor indicated that the upcoming CSV v3, integrated into its 4th-generation CPUs, would address the issue through architectural changes.

In coordination with Hygon, we aligned our paper submission with the commercial release timeline of CSV v3. Although CSV v3 was originally scheduled for release in October 2023, it was delayed due to manufacturing issues and was not launched at scale until March 2024. Accordingly, we submitted the first draft of our paper in 2024.

During the Usenix Security revision process, we expanded the investigation to validate known SEV attacks against Hygon CSV. The results, presented in Table 1, were disclosed to Hygon on May 19, 2025. As these attack vectors are already publicly documented, Hygon responded that they had existing awareness of the attacks and their potential applicability to CSV v1 and v2.

In addition to ethical disclosure, all experiments were conducted exclusively on isolated laboratory servers under controlled conditions, with no access to users' personal data or sensitive information. As a result, no harm was caused by the techniques or methods applied. We adhered to standard ethical guidelines and principles, including respecting privacy, avoiding harm, and maintaining transparency.

Acknowledgments

We appreciate our shepherd and the anonymous reviewers for their constructive suggestions, and Ruiyi Zhang for his help with the verification of CacheWarp. The work was in part supported by National Natural Science Foundation of China under grant No. 62361166633, National Key R&D Program of China under grant No. 2023YFB4503902.

References

[1] AMD–Chinese joint venture. [Online]. Available: https://en.wikipedia.org/wiki/AMD-Chinese_joint_venture. Accessed: Jan 2024.

[2] Attestation Service (AS or CoCo-AS). [Online]. Available: https://github.com/confidential-containers/trustee/tree/main/attestation-service#evidence-format. Accessed: Aug 2024.

[3] External secure memory device and system-on-chip SOC. [Online]. Available: https://patents.google.com/patent/CN107609405B/en. Accessed: Jan, 2024.

[4] Hygon China Security Virtualization (CSV). [Online]. Available: https://www.alibabacloud.com/blog/openanolis-officially-launches-its-first-csv-confidential-container-solution-with-hygon_599143. Accessed: Jan, 2024.

[5] Instance of Hygon Server on Alibaba Cloud. [Online]. Available: https://help.aliyun.com/zh/ecs/user-guide/general-purpose-instance-families#g7h. Accessed: April 2024.

[6] Instance of Hygon Server on Tencent Cloud. [Online]. Available: https://cloud.tencent.com/document/product/1397/72793. Accessed: April 2024.

[7] Memory controller, data writing and reading method, and computer system. [Online]. Available: https://patents.google.com/patent/CN119166290A/en. Accessed: May 2025.

[8] openAnolis. [Online]. Available: https://gitee.com/anolis/cloud-kernel. Accessed: May, 2025.

[9] openEuler. [Online]. Available: https://gitee.com/openeuler/kernel. Accessed: May, 2025.

[10] Privacy Computing Interconnection Framework. [Online]. Available: https://github.com/secretflow/InterOp/tree/main. Accessed: Aug 2024.

[11] SM4 (ShangMi 4.0). [Online]. Available: https://en.wikipedia.org/wiki/SM4_(cipher). Accessed: Jan 2024.

[12] TEEP Use Case for Confidential Computing in Network. [Online]. Available: https://www.ietf.org/archive/id/draft-ietf-teep-usecase-for-cc-in-network-07.html#name-use-cases. Accessed: Aug 2024.

[13] TrustFlow: zero-trust computing system. [Online]. Available: https://github.com/asterinas/trustflow/tree/main/trustflow/attestation. Accessed: Aug 2024.

[14] AMD. SEV-ES Guest-Hypervisor Communication Block Standardization. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56421.pdf, 2025. Accessed: May 2025.

[15] AMD. SEV Secure Nested Paging Firmware ABI Specification. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf, 2025. Accessed: May 2025.

[16] ARM. Realm Management Monitor specification. [Online]. Available: https://developer.arm.com/documentation/den0137/latest/. Acce May, 2025.

[17] ARM. Arm CCA Security Model 1.0. [Online]. Available: https://developer.arm.com/documentation/DEN0096/latest/, 2021. Accessed: Jan, 2024.

[18] Andrin Bertschi, Supraja Sridhara, Friederike Groschupp, Mark Kuhne, Benedict Schlüter, Clément Thorens, Nicolas Dutly, Srdjan Capkun, and Shweta Shinde. Devlore: Extending arm cca to integrated devices a journey beyond memory to interrupt isolation. *arXiv preprint arXiv:2408.05835*, 2024.

[19] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach. *ACM Computing Surveys*, 56(9):1–33, 2024.

[20] Dwaine Clarke, G Edward Suh, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Checking the integrity of memory in a snooping-based symmetric multiprocessor (smp) system. *MIT CSAIL CSG-TR-470*, 42(1-3):335–346, 2004.

[21] 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 23)*, pages 4051–4068, 2023.

[22] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[23] 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 *46th IEEE Symposium on Security and Privacy*. IEEE, 2024.

[24] Arm Developer. Learn the architecture: introducing arm confidential compute architecture.

[25] Morris J Dworkin. Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices. 2010.

[26] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[27] Simon Heron. Advanced encryption standard (aes). *Network Security*, 2009(12):8–12, 2009.

[28] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. *ACM SIGPLAN Notices*, 52(7):129–142, 2017.

[29] Intel. Exception Handling in Intel® Software Guard Extensions (Intel® SGX) Applications. [Online]. Available: https://cdrdv2-public.intel.com/671544/exception-handling-in-intel-sgx.pdf. Accessed: May, 2025.

[30] Intel. Intel® Trust Domain Extensions (Intel® TDX) ModuleBase Architecture Specification. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/733575. Accessed: Jan 2024.

[31] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. Supporting intel sgx on multi-socket platforms. *Intel Corp*, 2021.

[32] David Kaplan. Protecting vm register state with sev-es. *White paper*, 46:158, 2017.

[33] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 13:12, 2016.

[34] Dayeol Lee, Dooyoung Jung, IanT. Fang, Chia-Che Tsai, and RalucaAda Popa. An off-chip attack on hardware enclaves via the memory bus. *USENIX Security Symposium,USENIX Security Symposium*, Dec 2019.

[35] 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 *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022.

[36] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Crossline: Breaking" security-by-crash" based memory isolation in amd sev. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2937–2950, 2021.

[37] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected I/O operations in AMD's secure encrypted virtualization. *USENIX Security Symposium,USENIX Security Symposium*, Jan 2019.

[38] 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 21)*, pages 717–732, 2021.

[39] 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*, pages 1–6, 2018.

[40] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. Severity: Code injection attacks against encrypted virtual machines. In *2021 IEEE Security and Privacy Workshops (SPW)*, May 2021.

[41] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. Demystifying attestation in intel trust domain extensions via formal verification. *IEEE access*, 9:83067–83079, 2021.

[42] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. Wesee: using malicious# vc interrupts to break amd sev-snp. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4220–4238. IEEE, 2024.

[43] 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 24)*, pages 3459–3476, 2024.

[44] AMD Sev-Snp. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 53(2020):1450–1465, 2020.

[45] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[46] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE symposium on security and privacy*, pages 574–588. IEEE, 2013.

[47] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, and Shweta Shinde. Sigy: Breaking intel sgx enclaves with malicious exceptions & signals. *arXiv preprint arXiv:2404.13998*, 2024.

[48] 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*, pages 665–678, 2018.

[49] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, Oct 2017.

[50] 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*, Asia CCS '19, page 73–85, New York, NY, USA, 2019. Association for Computing Machinery.

[51] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. Tdx-down: Single-stepping and instruction counting attacks against intel tdx. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 79–93, 2024.

[52] 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)*, May 2020.

[53] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. Sev-step: A single-stepping framework for amd-sev. *arXiv preprint arXiv:2307.14757*, 2023.

[54] Luca Wilke, Jan Wichelmann, Florian Sieck, and Thomas Eisenbarth. undeserved trust: Exploiting permutation-agnostic remote attestation. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 456–466, 2021.

[55] 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 24)*, 2024.

[56] Du Zhaohui, Ying Zhiwei, Ma Zhenke, Mai Yufei, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. *Cornell University - arXiv,Cornell University - arXiv*, Dec 2017.