Relocate-Vote: Using Sparsity Information to Exploit Ciphertext Side-Channels

Yuqin Yan[†], Wei Huang[†‡], Ilya Grishchenko[†], Gururaj Saileshwar[†], Aastha Mehta[*], and David Lie[†]

[†]University of Toronto, [‡]Seneca Polytechnic, [*]University of British Columbia
yuqin.yan@mail.utoronto.ca, wei.huang1@senecapolytechnic.ca, gururaj@cs.toronto.edu,
aasthakm@cs.ubc.ca, {ilya.grishchenko,david.lie}@utoronto.ca

Abstract

Confidential computing implementations encrypt guest Virtual Machine (VM) memory to protect workloads from a malicious hypervisor. However, its use of system physical addresses as tweak values causes deterministic encryption for each physical memory address, creating a ciphertext side-channel. To exploit this weakness, we propose Relocate-Vote, a novel primitive that exposes frequency distributions across various memory locations by abusing management commands supported by confidential computing architectures such as `SNP_PAGE_MOVE` in AMD SEV-SNP. Unlike previous attacks that rely on secret information temporally written into specific locations, Relocate-Vote takes advantage of biases in the distribution of values that applications naturally exhibit, which are preserved under memory encryption with the same tweak values, and uses the spatial distribution of those values to leak sensitive information from confidential VMs. In this work, we demonstrate the generality of this attack primitive by using it to de-randomize Address Space Layout Randomization, extract 3D object data from OpenVDB, and leak token information during sparse LLM inference.

1 Introduction

Confidential computing is gaining more focus due to its potential for deploying sensitive workloads to the cloud even when the hypervisor is not trusted [40]. Compared with the traditional hardware and software stack, confidential computing prevents the hypervisor deployed by the cloud computing service provider from directly accessing and tampering with the states of guest virtual machines, creating confidential virtual machines (CVMs). Major cloud service providers (CSPs), including Microsoft Azure [28], Amazon Web Services (AWS) [5], and Google Cloud Platform (GCP) [10], offer CVM instances based on AMD SEV [2] and Intel TDX [9].

To protect the confidentiality of data in use, particularly in the memory of a CVM, implementations of confidential computing such as AMD SEV encrypt memory with a VM-specific key. Using AES-based encryption in XEX or XTS mode, 16-byte memory blocks are encrypted into 16-byte ciphertexts. To prevent the same plaintext from generating the same ciphertexts at different locations, tweak values derived from physical addresses are incorporated as an additional input that parameterizes a block ciphertext without altering the key. However, recent studies have exploited the vulnerability of address-based tweaks that causes identical plaintexts in a CVM at the same system physical address (sPA) to be encrypted to the same ciphertext. This deterministic output reveals internal states and enables a malicious hypervisor to exploit the collision of ciphertexts at the same location, known as ciphertext side-channel [25]. Prior ciphertext side-channel attacks have targeted cryptographic libraries [23, 25], DNN weights [45], and inputs [44]. Existing attacks rely on collision at specific memory locations over time, collecting ciphertexts and analyzing collision patterns for each location separately, as tweak values prevent analysis across different memory locations. For instance, CipherSteal [44] records binary sequences that flag whether ciphertexts at specific memory locations change for each write operation during execution. However, pinpointing the memory locations for collecting the ciphertext sequences remains challenging. Current approaches either rely on a combination of the performance monitoring unit (PMU) and precise implementation details of cryptographic libraries [23] or lack end-to-end attack implementations and instead assume some capability, such as using Intel PIN, to filter the write operations generated by the victim applications from system-generated memory traffic [44, 45].

In this work, we present a new end-to-end attack that does not rely on ciphertext collisions at specific locations. Instead, our primitive, Relocate-Vote, leverages frequency distribution biases of plaintext values. Because tweak values are physical-address dependent, these biases are preserved in ciphertexts for each memory location. This enables inferring the location of highly frequent prevalent values in an application's memory and extracting the spatial distribution of prevalent and non-prevalent values. Common examples of prevalent values are zeros or NULLs, and often used to represent a default or unallocated state in applications. Many applications exhibit

sparsity, having a large proportion of prevalent values in their memory, and the spatial distribution of the non-prevalent values in such applications can leak sensitive information about the application.

Similar to previous works [23, 25, 44, 45], this paper assumes the hypervisor is malicious and can access encrypted states of the CVM. To sample the frequency distribution of ciphertexts efficiently, our attack abuses management commands such as `SNP_PAGE_MOVE` in AMD SEV-SNP that are normally used by hypervisors to relocate confidential memory pages. By misusing these commands to cause different physical pages to be encrypted with the same sPA, the attacker can sample the ciphertext value distribution at arbitrary memory locations and derive the ciphertext corresponding to prevalent values. While previous attacks [25, 45] try to monitor every write operation to specific locations since the secrets are temporally written and missing write operations recordings downgrades the attacks' performance, our attack can reveal secret sparsity information already populated as a snapshot laid out in memory even without write operations, where the spatially populated secrets persist until being overwritten.

To demonstrate the security implications of this primitive, we present proof-of-concept attacks on three applications that rely on sparsity. First, we show how Relocate-Vote enables the de-randomization of ASLR for `glibc` addresses in applications running on CVMs, which is often the first step in applying ROP [8]. In this scenario, the sparsity information in the page table is already populated before the attack begins. Second, we demonstrate how the 3D object structure can be leaked during OpenVDB operations including construction and read-only traversal, where the library is used in medical image processing pipelines [20] and in industrial prototype design workflows, such as rocket engine development [1]. The sparsity information encodes the spatial arrangement of active voxels, enabling an attacker to reconstruct the object's geometric structure. Third, we show how Relocate-Vote can be applied to extract the geographic information of input tokens in sparse large language model (LLM) inference, where the ReLU activation function [31] is used for computational efficiency [38]. The attacker exploits the sparsity introduced by ReLU, which maps negative values to zero, to capture activation patterns associated with secret tokens.

We summarize our contributions as follows.

- We describe a new attack primitive on confidential computing systems, Relocate-Vote, that leverages management commands supported by the confidential computing architecture to learn ciphertexts of prevalent values, enabling the attacker to test for prevalent value matches on private pages at a 16-byte granularity.
- We demonstrate the implications of revealing the ciphertexts of prevalent values by demonstrating attacks that expose sensitive information by exploiting memory sparsity.
- We study the influence of Relocate-Vote on major cloud service providers (CSPs) and propose potential countermeasures to mitigate these attacks.

Responsible Disclosure. We have responsibly disclosed our findings to AMD on January 22, 2025. AMD has confirmed the vulnerability and assigned a pending security brief, AMD-SB-3021. AMD plans to release a feature update described along with the security brief.

2 Background: Confidential Computing

Confidential computing employs hardware extensions to protect the state of sensitive virtual machines (VMs) from being directly accessed by the hypervisor on which the VMs run. Specifically, confidential computing implementations typically achieve this through a combination of encryption and access control over read operations. For encryption, implementations such as AMD SEV-SNP and Intel TDX use AES-based encryption to encrypt CVM's private memory, encrypting 16-byte plaintext into 16-byte ciphertexts. However, the encryption is deterministic, generating the same ciphertext for the same plaintext when the physical address is the same, leading to a ciphertext side-channel [23, 25].

In terms of access control, AMD SEV-SNP originally did not implement any access control to prevent the hypervisor from accessing CVM memory. It was not until ABI specification version 1.55 (September 2023) that AMD introduced a ciphertext-hiding guest policy. A guest policy is a firmware-enforced configuration specified by the guest owner during launch that restricts the hypervisor's capabilities. The ciphertext-hiding variant prevents hypervisor's access to encrypted CVM memory via read access control. However, enabling this feature requires at least 5th-generation EPYC with DDR-BF [11], an optional DDR5 feature, making it incompatible with the previous generations of EPYC processors. Moreover, the ciphertext hiding policy must be specified by the guest to make the enforcement take effect. As a result, deploying this feature is non-trivial, as it requires specific hardware components, configuration by the cloud provider and enforcement by the CVM user. At the time of writing, hypervisors can access memory ciphertexts on AMD SEV-SNP confidential instances on major CSPs (AWS, Azure and GCP). In contrast, Intel TDX enforces access control over memory reads as a mandatory security measure, preventing the hypervisor from accessing ciphertexts in CVM's memory.

3 Attack Overview

Threat model. We assume the attacker controls a malicious hypervisor on which the victim CVM executes. Although the trusted hardware implementing confidential computing prevents the attacker from directly accessing or modifying the

internal state of the CVM, the attacker can still read the ciphertexts of the encrypted pages of the victim CVM and retains the abilities provided by the hardware to manage resources being used by the CVM. Specifically, the hypervisor can issue page relocation commands, normally used to manage memory resources. These commands—such as `SNP_PAGE_MOVE` and `SNP_PAGE_SWAP_OUT+SNP_PAGE_SWAP_IN` in AMD SEV-SNP—are executed by the trusted hardware, which ensures permission checks and provides re-encryption during relocation. In addition, the attacker can manipulate the permission bits in the host page tables that map guest physical addresses to system's physical pages to induce page faults in the guest CVM. This technique [23, 25], known as controlled-channel attacks [43], can be used to monitor the CVM's page usage by clearing permission bits on victim's pages, forcing the victim CVM to trigger permission violations when accessing these pages. For example, the attacker can remove the W (write) bit to trigger page faults due to write operations or remove P (present) bit to monitor guest CVM's page accesses. A malicious hypervisor with AMD SEV-SNP with ciphertext-hiding disabled is an instance of an attacker in our threat model.

3.1 Relocate-Vote

In this paper, we introduce a novel Relocate-Vote primitive that exploits ciphertext frequency distributions to learn the ciphertexts of prevalent values in a CVM page frame. This primitive uses the previously mentioned page relocation commands (Section 3), to relocate different CVM pages to the same physical page, causing different pages to be encrypted with the same key and tweak values (Figure 1). Thus, the frequency of the ciphertexts on different pages in each encryption block on the target page frame reflects the underlying plaintext frequency. The attacker uses this to (a) learn the ciphertexts corresponding to the prevalent values for a single physical page frame and then (b) extend this to recover the ciphertexts of the prevalent values for all page frames.

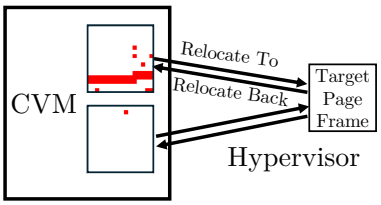


Figure 1: An attacker can arbitrarily relocate CVM pages onto a designated target page frame. This frame is allocated by the hypervisor and temporarily assigned to the CVM during each relocation operation.

Throughout this paper, the *prevalent value* usually refers to the most commonly occurring value in an application. However, in certain cases (which we will specify), it may also refer to the second most common value or the top few most common values. To learn such values, the attacker relocates a collection of CVM pages to a target page frame to gather ciphertext frequency distributions for each 16-byte encryption block. This reveals the ciphertexts of prevalent values, which can be of two types: (1) Global prevalent values are plaintexts with the highest frequency throughout the system (e.g., usually zeros); (2) Local prevalent values occur frequently only within specific regions of CVM memory (and are therefore less frequent globally than global prevalent values), often reflecting application-specific semantics of NULL values, such as the default background values in the OpenVDB example discussed in Section 5.2. To infer ciphertexts of global prevalent values, the attacker samples a subset of CVM pages and computes the ciphertext frequency distribution for each memory location. The most frequently occurring value will correspond to the prevalent plaintext value. To learn ciphertexts of local prevalent values, the attacker relocates faulting pages identified via the controlled channel, thus isolating application pages from other pages in the CVM.

The attacker can then extend their knowledge of the learned prevalent values on the target page frames to the rest of memory by relocating other CVM pages to the target frame. The location of the prevalent values on these other page frames will thus be known, and the attacker can then relocate these pages to arbitrary physical page frames to learn the corresponding ciphertexts on those page frames as well.

The page relocation commands have two important properties for the purposes of our attack: (1) *silent relocation*, where the hypervisor can arbitrarily relocate guest pages in a way that is invisible to the guest (detailed in Appendix A); (2) *non-interrupting*, where the CVM can continue executing while the relocation occurs and thus have minimal performance impact on the guest. However, since relocation temporarily invalidates the mapping from the CVM's page to its original page frame, the CVM may block for a short period of time if it happens to access the page that is being relocated. If the CVM does not access any page while it is relocated, then relocation does not affect the CVM's execution.

3.2 Exploiting Sparsity

Distinguishing prevalent values enables attackers to recover sparsity information from the victim CVM. This sparsity arises from secret-dependent memory behavior in the victim, where sensitive information influences how non-prevalent values are arranged in memory. This can occur in two ways: (1) secret-dependent offsets (Figure 2a), where different secrets cause the same data pattern to appear at different positions within memory buffers, and (2) secret-dependent layouts (Figure 2b), where the secret determines the data population, resulting in different spatial patterns of non-prevalent values across the buffer.

Our attacks consist of two phases: an *offline phase* conducted before the victim CVM's execution and an *online*

(a) Offset shifts (b) Layouts

Figure 2: Two forms of secret-dependent sparsity information observed in memory: offset shifts and layouts. Prevalent values are in white. Non-prevalent values are in red.

phase that takes place during the victim CVM's execution. In the *offline phase*, we assume the attacker knows the victim applications running inside the CVM and analyzes their behavior to understand how sparsity manifests in the applications. Through this, the attacker learns how to decode the pattern of prevalent and non-prevalent values to extract sensitive information from the application. In the *online phase,* the attacker first learns the ciphertexts of prevalent values in the CVM using Relocate-Vote, then monitors CVM's execution through controlled-channel attacks to locate pages containing secret information and finally recovers the encoded secrets by decoding the distribution of prevalent and non-prevalent values. To highlight the core idea of the attack and simplify implementation, our current attack prototype uses only a single target page frame during the online phase. Upon receiving a page fault via the controlled channel, the attacker immediately relocates the faulting page to this target frame, thereby revealing the collision status of each encrypted block on the faulting page. We discuss the performance implications on the victim's workloads and potential optimizations of this design in Section 8.3.

Case studies. We demonstrate three proof-of-concept attacks to show the breadth of scenarios to which Relocate-Vote can be applied. First, we target Address Space Layout Randomization (ASLR), a widely adopted software hardening technique that randomizes symbol addresses. By leveraging sparsity information in the CVM's page tables, we are able to derandomize the base address of `glibc`. The offline phase involves training classifiers to identify and extract secrets from page tables of target services running in the victim CVM. Second, we extract 3D object data processing with the OpenVDB library, which is widely used in scientific computing, simulations, and sensitive domains such as industrial design and medical imaging [20]. The offline phase involves analyzing how the victim populates secret data during object construction or accesses pages during data traversal. Successfully recovering such data reveals a serious vulnerability in handling confidential assets such as medical imaging [20] and industrial product prototypes [1]. Third, we exploit sparsity leakage in LLM inference workloads that exhibit sparse activations. Although confidential computing aims to protect the model, inputs, and intermediate states, token-level information can be leaked through the activation patterns of ReLU layers when LLM decodes prompt tokens. The offline phase includes analyzing the behavior of the serving framework—particularly ReLU activation signals—and training models to infer token information from activation patterns.

4 De-randomizing ASLR on CVMs

In this section, we focus on de-randomizing ASLR protections on applications running in CVMs. Unlike previous attacks initiated by privileged software [43], which depend on victims directly exposing their faulting virtual addresses, the hypervisor in our attack only accesses guest physical addresses and encrypted guest page tables. However, as the CVMs virtual address layout affects the population of prevalent and non-prevalent values in the guest page table pages, this still enables the attacker to infer which memory regions are populated.

The data structure of page tables is sparse because the virtual address space is significantly larger than the physical memory, and a process typically utilizes only a subset of the entire virtual address space. This sparsity results in virtual memory regions being either unallocated in the virtual address space or untranslated from a virtual address to a physical address in the page table. We refer to such regions as unmapped regions. Conversely, a region is considered a mapped region if at least one page in that region is allocated and has a corresponding non-zero entry storing translation information on the page table. At each level of the page table hierarchy—PGD, PUD, PMD, and PTE (Appendix B)—empty entries corresponding to unused and unmapped regions in the virtual address space are represented as an 8-byte zero. During the translation of a guest virtual address (GVA), the page walker extracts a slice of the GVA, a specific range of bits in the GVA (Table 1), to index the entries at each level of the page table hierarchy. These entries point to the guest page of the next level, as illustrated in Figure 3. Thus, the offsets of the mapped regions are linked to the GVA slice value.

Table 1: The size of the region an entry and encryption block represents in the page table page. The GVA slice bits are used to index the entries in each page table level.

Page Table Level	Page table entry	Encryption block	GVA slice bits
PGD	512 GiB	1 TiB	39-47
PUD	1 GiB	2 GiB	30-38
PMD	2 MiB	4 MiB	21-29
PTE	4 KiB	8 KiB	12-20

Figure 3 demonstrates how to decode the GVA from the sparsity information in the page table pages. Using the primitive described in the previous section, the attacker can detect whether two adjacent mappings are empty within each 16-byte

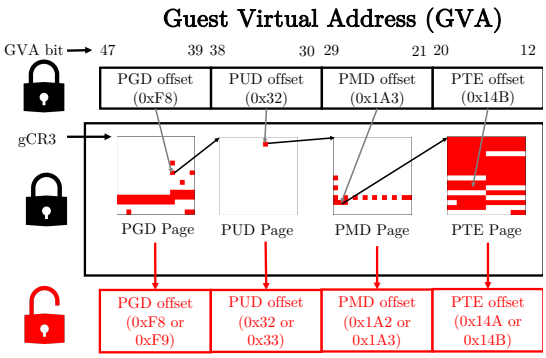


Figure 3: Page walk for translating a GVA and GVA slices are leaked via offsets of pointer entries.

encryption block on the encrypted page table pages. Furthermore, if the attacker knows which non-zero blocks are indexed by the GVA slice, they can use the offsets of the blocks to recover the values of the GVA slices. If an encryption block has an offset of n, its corresponding GVA slice is $2n$ or $2n+1$ because one encryption block can accommodate two page table entries. Thus, the intuition of the attack is to identify the page table pages involved in translating a symbol within a library or code segment that the attacker aims to de-randomize. By finding the non-zero encryption blocks indexed to find the next-level page table and obtain their in-page offsets, the attacker reconstructs the symbol's possible GVAs. Ultimately, this enables de-randomization of the base address of the code or library containing the symbol.

We assume the attacker aims to determine the base address of a victim application's `glibc` library, a common target due to its widespread use and rich set of functions that facilitate further exploitation, such as return-oriented programming (ROP) attacks [8]. Furthermore, although the attacker cannot execute any code inside the victim CVM [17], it can query a service that invokes at least one `glibc` symbol, such as `accept` for network function. This invoked symbol becomes the target symbol. In the offline phase, in their own VM, the attacker profiles the target applications and uses the profiled data to train classifiers that can recognize candidate pages for each page table level and that can infer the possible GVA slices corresponding to the target symbol. In the online phase, the attacker mounts a controlled-channel attack and leverages the page table accesses during the page walk to resolve the translation of the target symbol. By identifying candidate page table pages and offsets, the attacker reconstructs possible GVAs for the target symbol and thereby de-randomizes the base address of `glibc`. Our evaluation shows that on average, this reduces the search space from 2^{23} (8388608) possible addresses to just 35–104 candidates on a set of popular long-running server applications. The reduction stems from knowing exact offsets of non-zero encryption blocks in each page table level (Figure 3). We note that the attack could be made even more effective if false positives—non-page-table pages mistakenly contributing possible offsets— could be reduced, as these inflate the number of possible addresses.

4.1 Offline-Phase: Classifiers Preparation

The attacker prepares four classifiers—$\mathcal{C}_{\text{PGD}}$, $\mathcal{C}_{\text{PUD}}$, $\mathcal{C}_{\text{PMD}}$, and $\mathcal{C}_{\text{PTE}}$—for each level of the page table during the offline phase to identify page table pages and infer possible values of the corresponding GVA slices in the online phase. These classifiers are trained using attacker-controlled CVMs before attacking the victim CVM by profiling how mapped regions are accessed across different page table levels. Each classifier takes as input a 256-bit bitmap representing whether each 16-byte encryption block on a page is zero or non-zero. The training process leverages common global memory layout conventions and the interleaving of mapped and unmapped regions near the target symbol.

In a system, every process follows a standard layout for its virtual address space, including text, heap, and stack segments, as well as user and kernel spaces. This relative position of each segment is constant across all processes and ASLR simply modifies the relative offsets between segment, which manifests as shifts in the non-zero encryption blocks in the encrypted PGD pages, since each PGD page represents the mappings of the entire virtual address space. We provide visualizations illustrating the consistent structure in Appendix C. Since the high-level address space layout is consistent across all applications, we train a single classifier $\mathcal{C}_{\text{PGD}}$ to identify PGD pages for any application. This classifier uses a simple CNN [21] that performs binary classification on 256-dimensional inputs corresponding to the input bitmap. The CNN consists of two convolutional layers with ReLU activations and max-pooling, followed by a fully connected layer and a single-neuron output for prediction.

If a page is identified as a PGD page, the attacker needs to determine the encryption block that represents the `mmap` region where the `glibc` library is mapped to produce the possible values for the GVA slice of bits 39–47 (Table 1). Typically, at least three distinct mapped regions exist in the first half of the PGD page (Figure 4a): the text/heap segment, the `mmap` region and the stack segment. The bits representing the `mmap` region are the middle ones. In some cases, however, only two distinct regions are represented in the first half of the PGD page (Figure 4b). This occurs when the `mmap` region is close to the other segments and becomes merged with them. In this case, both relevant bit offsets in the first half of the bitmap are used to generate possible values of the GVA slice.

In contrast to the PGD, the lower page table levels (PUD, PMD, and PTE) describe increasingly smaller regions of the virtual address space. Although ASLR randomizes the base addresses of these segments, it does not alter the offsets between allocated and unallocated mappings within each segment. By exploiting these consistent offsets, the attacker con-

(a) At least three mapped regions in the first half

(b) Only two mapped regions in the first half

Figure 4: PGD pages: different mappings in the first half.

structs application-specific classifiers for each page table level that can identify PUD, PMD, and PTE pages based on the intra-segment pattern of mappings surrounding a target symbol, which are robust to ASLR's randomization. The attacker accomplishes by profiling the application to obtain sequences of binary bits, with each bit representing a region whose size is determined by the information represented by an encryption block a the corresponding page table level, as specified in Table 1 (e.g., 4 MiB at the PMD level). By examining the `pagemap` in `/proc`, the attacker records whether each region hosts at least one virtual to physical address translation ('1' for presence and '0' for absence).

Recall that our primitive can recover a 256-bit bitmap of allocated and unallocated regions for each page table page. Due to ASLR, this sequence shifts within the address space. A page whose 256-bit bitmap matches any subsequence of the profiled sequences becomes a candidate for the corresponding page table level. The offset of the bit that represents the encryption block containing the target symbol is then used to determine the values of the GVA slice. The bit corresponding to the target symbol may shift across the 256-bit bitmap of a page due to ASLR, necessitating an expansion of the sampling range by 255 bits on each side of the central bit representing the region encompassing the target symbol. This creates a 511-bit fingerprint as exemplified in Figure 5, capturing the mappings around the target symbol, where each bit is linked to a memory region defined by the page table level. The central bit denotes the region containing the target symbol.

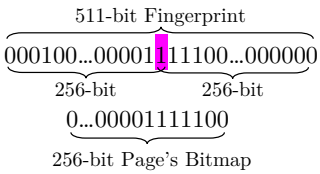


Figure 5: A fingerprint and a majority bitmap matching it.

The attacker collects N fingerprints by repeatedly running the target application, forming a fingerprint set. This prepares for the online phase, where the attacker checks if the 256-bit bitmap of a faulting page matches a subsequence of any fingerprint for each faulting page. Matches identify candidate pages, and the offset of the central bit provides possible values of the GVA slices for the target symbol. For example, in Figure 5, a matching offset 251 of the central bit in the 256-bit majority bitmap corresponds to GVA slices 502 or 503. By combining these slices across all page table levels, the attacker can recover the full GVA of the target symbol. The base address of the `glibc` library is then deduced from the internal offset of the target symbol.

To evaluate whether N fingerprints sufficiently cover possible bitmap sequences, we use the Good-Turing estimator [34]. The probability of unseen patterns is given by $P_{\text{unseen}} = \frac{N_1}{N}$, where N_1 represents fingerprints observed only once, and N is the total number of observations. A large P_{unseen} suggests the probability of encountering an unseen sequence during the online attack is high, potentially failing to produce the correct possible GVAs by missing matches with the actual page table pages. Thus, the attacker must sample more fingerprints until P_{unseen} is lower than an empirically pre-defined threshold. To expedite this process, the attacker can benefit from parallel collection by deploying multiple fingerprint-generation machines simultaneously. In our experiment, collecting fingerprints with P_{unseen} achieving lower than 1% for all evaluated applications takes around one day.

4.2 ASLR Online-Phase Attack Flow

After building the classifiers for each page table level, the attacker performs the online-phase of the attack, leaking the target symbol's GVA by triggering access to the target symbol, thus de-randomizing the base address of the `glibc` library. Since the empty entries are represented by zero (i.e., a global prevalent value), the corresponding ciphertexts can be learned via random sampling from CVM's pages and collecting ciphertext frequency information as described in Section 3.1. After learning the prevalent value ciphertexts, the attacker can distinguish the zeros and non-zero encryption blocks for each page in the victim CVM. With this capability, the attacker tracks all pages in the CVM, which is achieved by clearing the W (write) bit in the host page table for all guest pages to enforce write protection. The page walk in the CVM will violate page write protection, as the hardware will try to set the access bit in the guest page table entries (detailed in Appendix B). The attacker then queries the service that accesses the target symbol, causing a series of page faults. For each page fault, the attacker records the guest page frame and produces its 256-bit bitmap by relocating it to the target page frame with known ciphertexts of the prevalent values and lifts the write protection on the page by setting the W bit to allow the CVM to continue execution. The tracking ends when the service completes the query. Since responding to the query requires resolving the target symbol's translation, the page table pages involved also get recorded in the page-fault events.

Page-faults can occur on both page table pages and non-page table pages. To distinguish between them, the attacker uses the classifiers derived from the offline phase to determine whether a page-fault corresponds to a page table page, and

if so, identify the specific level of the page table that was accessed (e.g., PUD, PMD, or PTE). The attacker then identifies all valid quadruples, $Q(A,B,C,D)$, in the sequence of faulting pages, forming a set S_Q: each quadruple $Q(A,B,C,D)$ satisfies the condition that A, B, C, and D occur in order and correspond to PGD, PUD, PMD, and PTE candidates, and also produces possible GVAs by combining the possible GVA slices. The pages forming S_Q produce a set of candidates, S_c. This process represents one iteration of tracking. The attacker can perform multiple iterations to obtain several sets of quadruples S_Q and candidate pages S_c. By intersecting these sets and the possible GVAs across iterations, the attacker reduces the sizes of the sets and the possible GVAs, as the ASLR offset is not changed as long as the process is not restarted.

If intersecting the sets cannot further reduce the size of the sets, the attacker can opt in a more fine-grained approach we call *windowed tracking*. In windowed tracking, a window size w is defined, where each window contains w page-fault events. At the beginning of each window, the attacker clears the W bit for pages in S_c that faulted in the previous window, thereby bringing them back to the *tracking set*—the set of pages currently being monitored for write activity via induced page faults. This approach guarantees that the target symbol's translation is resolved within a single window: if the target symbol is accessed and a page walk is required, the pages involved in its translation must appear within the same window to access the symbol. The attacker identifies quadruples within each window and intersects them with the original S_Q, further refining the set of possible GVAs.

4.3 ASLR Leakage Results

For evaluation, we use nginx [32], Apache [16], MySQL [33], Redis [36], and Memcached [27], which provide long-running network and data management services. The victim CVM runs Ubuntu 24.04.1 LTS with one core and 4 GiB of memory.

Offline-phase: CNN-based classifier for PGD. The training set for the PGD classifiers comprises 568 PGD pages and 410481 non-PGD pages with multiple reboots. Considering the significant class imbalance between PGD and non-PGD pages, SMOTE [15] is employed to generate synthetic samples for PGD pages. The trained classifier successfully classifies 102613 non-PGD pages and 142 PGD pages in the test set, with 8 non-PGD pages identified as PGD pages.

Offline-phase: Fingerprint-based classifiers for other levels. We collected $N = 150000$ fingerprints per page table level for each application, using 50 attacker-launched fingerprint-generation CVMs (each with 1 core and 4 GiB memory). Each CVM completed fingerprint collection in 23.75 ± 2.41 hours on average. We then applied the Good-Turing estimator on 1%, 5%, and 50% of the fingerprints as the sample set to predict the coverage, defined as the proportion of fingerprints in the remaining dataset also appearing in the sample set. The predicted coverage is computed as $1 - \frac{N_1}{N}$ described in Section 4.1. The results demonstrate that the Good-Turing estimator accurately predicts coverage for each page table level and each application in each sample set (detailed in Appendix D). We selected the 50% sample set of the 150000 fingerprints, as it achieves over 99% coverage in all applications and page table levels, indicating the comprehensiveness of the fingerprint collection.

Online-phase attacks. We conducted online-phase attacks on the victim CVM. The tracking iterations were terminated when the number of possible GVAs for the `glibc` base address did not decrease over five consecutive iterations. We run both non-windowed tracking (non-WT) and windowed tracking (WT) with window sizes of 128 and 64. We repeat the attack 10 times with 10 random memory layouts for each application.

Table 2: Online ASLR attack results: ASR and **Average number of possible GVAs** / Average number of page-fault events.

Application	ASR	Non-WT	WT ($w = 128$)	WT ($w = 64$)
nginx	10/10	**321**/1988	**106**/2565	**104**/2840
apache	10/10	**214**/1954	**74**/2315	**62**/2706
mysql	9/10	**264**/3139	**54**/3473	**52**/3975
redis	10/10	**58**/1916	**38**/2288	**35**/2643
memcached	10/10	**75**/1841	**48**/2355	**41**/2656

Table 2 includes the attack success rate (ASR), defined as the proportion of runs where the actual GVA of the `glibc` base is included in the set of possible GVAs derived from tracking, the number of possible GVAs for the `glibc` base address at termination, and the average number of page-fault events triggered per iteration. The results demonstrate that our attack achieves a high success rate, failing only once to generate the potential base address of `glibc` in MySQL. The evaluation system features 2^{23} (8388608) possible addresses for the `glibc` base, with entropy derived from 5 bits in the PGD page, 9 bits in the PUD page, and 9 bits in the PMD page. However, the evaluation system has no entropy at the PTE page level, as the `glibc` base address aligns with a 2 MiB boundary. The results demonstrate that the set of possible GVAs can be reduced to 35–104, depending on the application, representing a significant reduction.

5 OpenVDB Leakage

OpenVDB [30] is a library designed for managing sparse volumetric data, enabling efficient representation and manipulation of three-dimensional grids with attributes such as distance and density. OpenVDB employs voxels, which represent points in 3D space, analogous to pixels in 2D space. To efficiently store and manage sparse data for 3D objects, OpenVDB stores voxels in a hierarchical tree structure with multiple levels of nodes.

In OpenVDB, the default background value is used to fill node buffers during initialization. As a result, it becomes the prevalent value, creating sparsity patterns when interleaved with non-background values, including pointers to child nodes and actual voxel data. In this section, we show how an attacker correlates the fundamental operations of the construction and the traversal process with the underlying tree structure. Since the tree structure reflects the spatial distribution of voxels, which is sensitive, the attacker can reconstruct the shape of 3D objects by analyzing the interleaving of background and non-background values in node buffers.

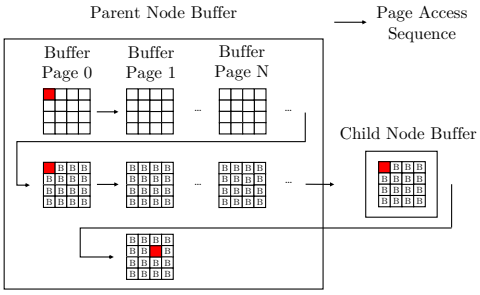


Figure 6: Buffer initialization and parent-child link.

5.1 Offline-Phase Application Analysis

The OpenVDB tree structure consists of a root node, multiple internal layers, and a leaf layer. The root node represents the entire 3D space and creates child nodes only for regions with actual data. Internal nodes recursively subdivide the regions their parent represents into smaller regions, forming a hierarchical structure. Leaf nodes store the voxel data. The tree elements correspond to coordinates in 3D space. The root node uses a map data structure to directly map coordinates to child node pointers. Internal and leaf nodes use fixed-size arrays, determined by the layer, to store pointers to child nodes or values. The index of an element in the array corresponds to the local 3D coordinates of a subregion within the space covered by the node, providing a mapping between a 1D offset and 3D coordinate information.

Object construction. An OpenVDB object is constructed by incrementally adding voxels into the 3D space. Initially, the tree is empty, consisting only of a root node with no entries but containing a specified background value for the object. When a voxel is added, internal and leaf nodes are lazily created as necessary to store the voxel data. Lazy appending of child nodes involves two key steps, as shown in Figure 6: node initialization and non-background value insertion. The node creation involves filling the buffer with the background value. A child node must be created after its parent node; after the child node buffer is initialized, the corresponding element in its buffer is updated to point to this new child node. The buffer offset updated in the parent node is determined by the relative 3D coordinates of the child node, indicating where the active subregion is added. As voxels are inserted, the buffers in internal and leaf nodes become populated with interleaved background and non-background values, maintaining spatial information through offsets corresponding to relative 3D coordinates of the regions they represent.

Object data traversal. OpenVDB also provides native iterators that perform depth-first-search (DFS) traversal to process object data. As illustrated in Figure 7, the iterator sequentially accesses entries in a parent buffer and descends into a child node whenever an entry points to one. Unlike object construction, where voxels are inserted into the tree in arbitrary order, such iterator proceeds in a deterministic order. For example, the iterator visits the first child of a node before moving to the second, implicitly revealing parent-child relationships. An attacker exploits this behavior by leveraging (1) the ability to distinguish background values using our primitive to reveal buffer contents, and (2) the ordering of page faults, which reflects the access sequence of nodes during traversal. The attacker combines this information to recover the positions of the voxels in 3D space.

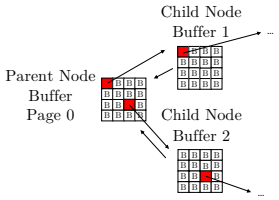


Figure 7: Tree traversal in OpenVDB.

5.2 Online-phase Tracking

Learning ciphertexts of the background value. In both object construction and traversal, the background value associated with the object becomes the local prevalent value. Its ciphertexts remain learnable even when the background value is non-zero. First, they can be leaked from earlier processing of objects with the same background value, even if the objects are different and stored at different memory locations, by leveraging memory snapshots from prior processing and using the Relocate-Vote primitive to inspect their re-encrypted contents on the target page frame. Second, since a tree structure often spans many pages filled predominantly with the same background value, the attacker can exploit ciphertext frequency of the background value across multiple pages within the same object during a single execution to infer the corresponding ciphertexts on the target page frame.

Tracking during object construction. The attack relies on detecting events corresponding to two key operations during online tracking of object construction: (1) node initialization, where buffers are filled with the background value, and (2) insertion of non-background values. Unlike the ASLR scenario, where pages containing secret information remain unchanged

before and after tracking, this tracking requires observing multiple versions of values on the same pages, as illustrated in Figure 6. First, the attacker monitors dynamically allocated buffers during node initialization. Second, for a page containing an internal node buffer, the attacker must track intermediate updates after initialization, as any overwrite of a background value indicates the insertion of a child node.

We implemented a fine-grained tracking method allowing the attacker to observe intermediate memory updates while still allowing the victim CVM to make forward progress. This method begins by tracking all guest pages by clearing the W (Write) bit in the host page table entries, enabling the observation of ciphertext changes caused by writes. Specifically, it captures the memory state resulting from the last write to the page that triggers a fault and any updates made to that page before a subsequent page fault occurs on a different page. For each faulting page, the attacker relocates it on the fly to the target page frame and identifies ciphertexts matching those associated with the background value. The range of the identified blocks reveals the region initialized with that value. Upon detecting a buffer initialization, the attacker records the pages involved and the range of background values within each page. It then adds the page back to the tracking set after w page faults have been triggered by other pages to capture further updates. Later, a page fault in the parent node's buffer is triggered when a child node is added. By observing the update, the attacker links the newly created child node to the corresponding element in the parent node, thereby forming an edge in the tree structure.

For tracking efficiency, the attacker can progressively eliminate pages not initialized with node buffers from the tracking set. Each guest page maintains two separate counters: T_{active} and T_{silent}, tracking active and silent write events, respectively. An active write refers to a page fault that results in a memory content update, while a silent write does not change the contents. If either counter reaches its threshold and the page has not exhibited any node buffer initialization, it is permanently removed from the tracking set and no longer triggers page faults. The T_{silent} counter accounts for frequent writes that repeat previously stored values. In contrast, T_{active} is intended to filter out pages that undergo frequent updates inconsistent with node buffer behavior.

Tracking during object traversal. As an alternative, the attacker can also attack OpenVDB after the tree structure and data have already been populated in memory. Therefore, unlike during object construction, the attacker does not need to track intermediate states caused by write operations. However, since multiple buffers may reside on the same page, the attacker must still monitor repeated page faults on those pages as different buffers are accessed. Assuming the traversal is read-only, the attacker begins by clearing the P (Present) bits in the host page table entries to trigger page faults on CVM's page accessing events. Upon encountering a fault of a new page, the attacker relocates it to the target page frame and records its ciphertexts. If the page contains background values—indicated by memory blocks matching the corresponding ciphertexts—it is added to the tracking set to be revisited on subsequent faults triggered by co-located node buffers. The tree structure corresponding to the processed object can then be recovered by analyzing the sequence of page faults in conjunction with the sparsity information embedded in the node buffers.

5.3 Object Recovery

Once the tree structure is reconstructed in the online phase, the attacker extracts active regions in the 3D space, beginning with the leaf buffers. Non-background values in the leaf buffers represent the relative 3D coordinates of active voxels. These offsets are mapped to coordinates using the library's intrinsic mapping method, revealing the distribution of active voxels within the 3D block the leaf buffer represents. When a leaf buffer is linked to an element in an internal node (its parent), the relative coordinates within the block covered by the parent node are inferred based on the offset of the corresponding element in the parent buffer. This recursive process propagates spatial information up the hierarchy through the internal nodes until the top-level internal layer is reached. By traversing this hierarchy, the attacker reconstructs the spatial distribution of voxels in the 3D object. However, the attacker faces challenges with the map-based data structure of the root node, which uses key-value pairs to map 3D coordinates to subregion pointers. In our experiment, these pieces can be assembled manually, which is feasible because each root node corresponds to a large, box-shaped subregion of the 3D space, resulting in only a small number of distinct components. These subregions have clearly defined boundaries as flat faces and right angles that naturally suggest how they align with neighboring regions. Moreover, when adjacent blocks are placed together, the continuity and smoothness of the underlying surface geometry across block boundaries serve as visual cues for correct alignment.

5.4 OpenVDB Leakage Results

To demonstrate the extraction of sensitive objects, we simulate (1) construction: converting DICOM format data—widely used standard in medical imaging for storing and transmitting data, including image slices and associated metadata—into an OpenVDB object and extracting the object under construction during this process, and (2) traversal: a read-only traverse with the native iterator `cbeginValueAll`. We use the first scan, `CQ500CT0/CT Plain`, from the CQ500 dataset [35]—a publicly accessible collection of 491 anonymized head CT scans curated for medical imaging research. This scan comprises 30 slices in DICOM format.

To simulate the construction operation, we apply a threshold of 20 to the slices' values, a common medical imaging

technique to isolate specific tissues or structures [4]. The filtered voxel data is inserted into an OpenVDB grid, specifically a default `FloatGrid`, which consists of a 4-layer tree with two internal layers (Internal-1 and Internal-2), uses float as the data type, and has a background value of -1000 to represent air. For traversal, we simulate a victim CVM traversing the constructed grid using the code in Listing 1, a minimal yet representative style for examining values across the grid, introducing two challenges: (1) the constant iterator performs no writes, and (2) voxel values do not affect access patterns when processing each of them. Still, such exploitation generalizes to more complex processing scenarios.

Listing 1: Tree traversal with DFS-style iterator

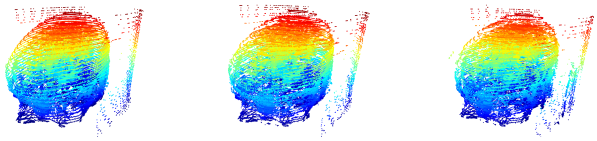
```
for (auto iter = tree().cbeginValueAll(); iter; ++iter) {
  float value = iter.getValue(); // Processing not voxel-dependent
} // Processing code omitted
```

Table 3: Node discovery information (construction, traversal).

Internal 1	**Internal 2**	**Leaf**
# Found / # Total	# Found / # Total	# Found / # Total
8/8, 8/8	8/8, 8/8	4576/4626, 4616/4626

The source object is illustrated in Figure 8a, consisting of 71349 active voxels spanning on 3045 pages. We run each type of tracking three times and take the average value to report as shown in Table 3, including the number of discovered node buffers during the trackings. For tracking the construction, we set $w = 8$, $T_{\text{active}} = 128$, and $T_{\text{silent}} = 1024$. On average, the active threshold T_{active} caused one buffer page to be prematurely removed, as it reached the threshold before its initialization was observed. The silent threshold T_{silent} led to an average of 14 such removals. In the traversal scenario, a subset of node buffers was missed due to access patterns being masked by co-located buffers on the same page. Each attack instance generates 1522035 page-fault events for construction and 48547 page-fault events for object data traversal on average. The significant reduction in page faults in the traversal scenario is attributable to the fact that tracking intermediate updates for potential buffer allocation is unnecessary. The number of page faults is even smaller than the number of voxels because of the primitive's ability to directly extract value-based information from static memory snapshots containing populated secrets. Since each page leaks multiple secret values, the extraction cost is effectively amortized.

The extracted object is visualized in Figure 8b and 8c, after manually assembling the eight pieces derived from the online phase, as described in Section 5.3. The visualization is based on the worst-case tracking scenario, where the largest number of pages containing node buffers were missed, demonstrating that the attacker can still infer the structural information of the object. Although most tree nodes are successfully discovered, some inaccuracy arises due to the 16-byte encryption granularity, as each encryption block may span multiple elements (e.g., two pointers or four floating-point numbers). To simplify handling this inaccuracy, for non-background values in leaf nodes, we populate only the first voxel represented by each non-background encryption block. For internal nodes, which store parent-child relationships as pointers, we adopt different strategies based on the extraction scenario. During construction-based extraction, if one child is present, we assign it to the first entry; if two children are detected within the same block, we randomly assign parent-child relationships, since the ordering of pointers within an encryption block is non-deterministic. In traversal-based extraction, the order of children can be inferred from the sequence of accesses to the child nodes. However, missing a child node may still disrupt the insertion of subsequent children sharing the same parent buffer page. Despite these inaccuracies, our measurement of the distribution of nearest-neighbor distances between source and recovered voxels (Figure 9) shows a clear advantage over randomly populated points within the same bounding box. This shows that the attacker can recover high-quality approximations of the protected data in the CVM.



(a) Source (b) Construction (c) Traversal

Figure 8: Extract 3D object constructed from CT scanning. Color is not recovered, only added for better visualization.

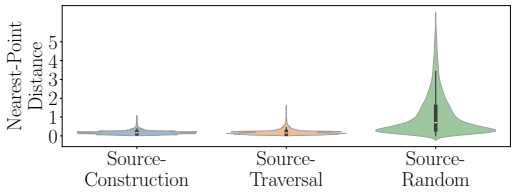


Figure 9: Distributions of nearest-neighbor distances from source voxels to recovered and randomly populated points.

6 Sparse LLM Leakage via ReLU Activation

Recent works in LLM inference focus on reducing the memory and computational requirements by leveraging sparsity in neuron activations [26,29,38], specifically in the self-attention and multi-layer perceptron (MLP) layers. For example, researchers at Apple [29] proposed architectural modifications to Llama by replacing its SiLU [13] gating function in its feed-forward network—defined as $\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1+e^{-x}}$)—with ReLU. This line of work has led to the development of

artifact models like ReLULlama [39], a set of Llama-2-based models utilizing ReLU activation.

Our attack exploits sparsity information leaked by ReLU activations during sparse LLM inference. During inference, both user-provided prompt tokens and auto-regressively generated tokens are processed by the `decoding` function, which performs a forward pass through all model layers, each containing a feed-forward network with ReLU activations. ReLU generates input-dependent patterns of interleaving zero and positive values and can reveal information about the processed tokens. To exploit this leakage, the attacker trains an *activation probe* during the offline phase—a lightweight model designed to perform prediction or classification—based on the model's internal state. This approach is inspired by prior work on probing classifiers, which use internal representations to infer external properties [3, 7, 18]. However, unlike these works, our attacker observes (1) the ReLU activation information instead of the layer's output (i.e., the hidden states, see Figure 10) and (2) only limited information (i.e., the knowledge of zero ciphertexts at the granularity of encryption). Thus, the attacker can only observe *blurred* activation information: they can determine only whether at least one activation within a 16-byte block (encryption granularity) is non-zero.

We demonstrate a scenario where the attacker aims to learn the geographic coordinates of the LLM inputs [18]—specifically latitude and longitude—based on blurred activation information obtained when the victim processes the user's prompt. Specifically, we assume the victim processes a prompt in a victim CVM with the format: "What are the lat/lon coordinates of `<place_name>`". The attacker trains a probe model in the offline phase and measures the activation information of the last token in the user input to feed into the probe in the online phase. Our results show that, even with blurred visibility, the attacker can infer the geographic coordinates with performance comparable to scenarios where full hidden states are available in the original non-sparse model.

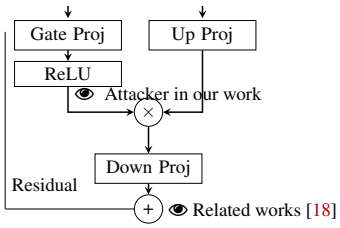


Figure 10: The attacker targets the ReLU's buffer.

6.1 ReLU Activation Signals

We studied the PowerInfer [38] serving framework, which is based on `llama.cpp` and supports sparse LLM inference. The implementation optimizes memory efficiency and utilizes in-place ReLU operations, overwriting the output of the preceding operator by zeroing out negative values. Element-wise multiplication is performed in place within the same buffer after ReLU activation and is expected to update only non-zero blocks. However, due to hardware or compiler implementation, zero elements sometimes produce negative zeros during multiplication. Knowledge of the ciphertexts corresponding to positive zeros as a global prevalent value helps the attacker identify ReLU activation buffers and their activation results. Additionally, it enables the attacker to disambiguate behaviors across two operations: In ReLU activation, a block may remain unchanged because it contains either all positive values or all zeros; in multiplication, a block may be updated because it contains either at least one positive value or a zero that is converted into a negative zero. Identifying encryption blocks filled with all positive zeros allows the attacker to distinguish these cases more precisely, beyond relying solely on ciphertext collisions with previously observed ciphertexts [44].

When processing a prompt, the sequence of N input tokens is decoded as a batch. In each layer's ReLU operator in the feed-forward network, each token is associated with a d-element vector, forming an $N \times d$-element buffer laid out contiguously in memory. The value of d is model-specific, e.g., in `ReLULlama2-13b`, $d = 13824$.

6.2 Leveraging ReLU Activation Information

We assume that the elements in the ReLU activation buffer are `float32`. We demonstrate the attack with the `ReLULlama2-13b` model, with a 13824-dimensional ReLU vector in each layer. We use the experiment from prior work [18] to demonstrate the ability to extract useful information from blurred activation information. In this study, they use prompts ending with place names as input and collect hidden states (layer output) for each model layer on the last token. They then train linear regression probes for each layer on the collected hidden states to predict the corresponding latitude and longitude, using a dataset of 39585 place-name entities to form prompts. 20% of the data was held out as a test set, while the remaining 80% was used for training with Leave-One-Out Cross-Validation (LOOCV) [19] to select probe parameters. Their results show that spatial information is linearly decodable in the tested models, with high R^2 across layers and various prompts.

In our attack, instead of collecting the hidden states, we collected the activation information from the dataset consisting of 39585 prompts from running the PowerInfer framework, where each prompt was processed to extract ReLU activation results at each layer for *the last token*. The activations were transformed into a 3456-dimensional binary vector by grouping every four elements into a single bit to simulate the attacker's visibility: 0 if all were zero, and 1 otherwise. A linear regression probe was trained on this data with a training set of size 31668, and evaluated on 7917 test prompts, simulating victim-side inference of processing prompts formed

with the place-name entities in the test set on a CVM.

Results. The probe's performance on the test data is in Figure 11, including the R^2 and the proximity error. The proximity error is defined in the original work and quantifies the proportion of entities predicted to be closer to the target than the predicted point. This metric, ranging from 0 to 1 with a random performance of 0.5, measures the relative spatial accuracy of the predictions. The original work reports the R^2 and the proximity error at the 60% layer depth as 0.896 and 0.068, and our variant yields 0.80 and 0.13. The results demonstrate that even with blurred activation information, the attacker achieves a high correlation and a low proximity error, highlighting its utility. Figure 12 visualizes the predicted coordinates at the 60% layer-depth and their actual continent, illustrating that continent-level spatial information is well-preserved.

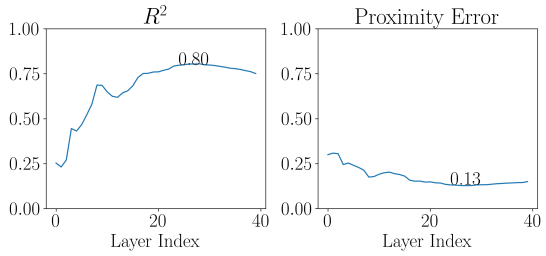


Figure 11: R^2 and the proximity error in each layer.

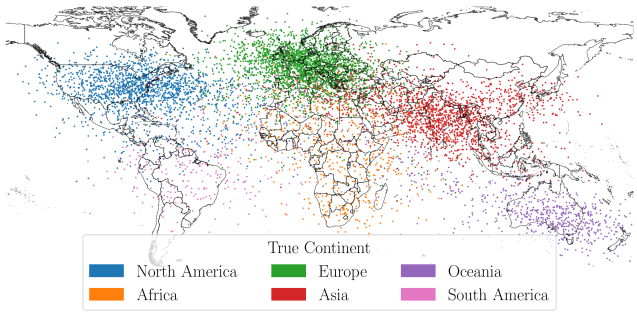


Figure 12: Predicted coordinates of the places in the test set with the attack's blurred information.

7 Implementation

We implement our attack on a 3rd-generation AMD EPYC 7543 with the SEV-SNP firmware version 1.55.21.

7.1 Relocate-Vote Implementation

We implemented Relocate-Vote as a procedure in the attacker's kernel including the following modifications.

Implementing `SNP_PAGE_MOVE`. We added kernel functions for the hypervisor to request `SNP_PAGE_MOVE` as it has not been officially implemented in AMD's Linux kernel fork yet [6]. Our implementation follows the official ABI specification: The source page of relocation should be immutable, and the CVM should own the target page [2]. These requirements cause two `rmpupdate` requests for updating the page's metadata stored in the RMP table [2] maintained by the hardware. `SNP_PAGE_MOVE` is then called to request re-encryption of the contents in the target page while invalidating the source page.

Recovering kernel's map of guest private pages. We allocate the target page and move pages to it to collect re-encrypted ciphertexts. By default, the kernel mappings to CVM are removed once they are assigned to a CVM. We modify the kernel to retain such mappings so that our malicious hypervisor can maintain access to guest CVM pages.

Host cache coherence on guest private pages. When the hypervisor reads guest CVM's private pages, it brings the corresponding cache lines into the cache. However, during hardware-initiated page relocation onto the same page frame, these cache lines belonging to the hypervisor are not invalidated. Consequently, the hypervisor may access stale data from the cache. To avoid this, the hypervisor issues `clflush` instructions on the frame region of the target page before re-accessing it.

After the preparation above, we implement Relocate-Vote (Figure 1). The hypervisor first allocates a target page frame, enabling iterative relocation of victim pages to this frame. In each iteration, a page is relocated to the target frame, its ciphertext is collected after re-encryption, and the page is then moved back to its original frame. This process—consisting of two relocations, a page flush, and ciphertext recording—takes 2.02 milliseconds on systems running firmware version 1.55.20 or later, where AMD addressed performance issues in the `SNP_PAGE_MOVE` implementation.

7.2 Implementation of Online Attacks

Implementation of ciphertext learning. We implement ciphertext learning of global prevalent values (i.e., zeros) by randomly sampling private pages from the CVM and relocating them to a designated target page frame. Specifically, we relocate 4096 private pages (taking approximately 8 seconds) out of every 1 million, demonstrating that even a small subset of pages can be used to reliably recover the global prevalent values from the ciphertext frequency distribution. We record the most frequent ciphertexts observed for each encryption block for the target page frame, enabling the collision status test by comparing against them. As part of our proof-of-concept to validate correctness of learned ciphertexts, we have the victim CVM populate a page with zeros, relocate it to the target page frame, and check if the resulting ciphertexts match the learned ciphertexts. For local prevalent values (e.g., the background value in the OpenVDB scenario), we assume that an object using the same background value was processed earlier at different memory locations, leaving

Table 4: Comparison of the capabilities of the hypervisor in different confidential computing platforms.

Platform	Address-based tweak	Relocation	Controlled Channels	Ciphertext access from hypervisor
AMD SEV-SNP	Yes	`SNP_PAGE_MOVE` `SNP_PAGE_SWAP_IN/OUT`	Directly manipulable from host page table	No when ciphertext-hiding enabled (5th-generation or later EPYC with DDR-BF mode required)
Intel TDX	Yes	`TDH.MEM.PAGE.RELOCATE`	`TDH.MEM.RANGE.BLOCK` `TDH.EXPORT.BLOCKW`	No
ARM CCA	Yes	No	No	No

its pages populated in memory. This enables the attacker to relocate those previously accessed pages and learn the corresponding ciphertexts via frequency. This reflects the core idea of the primitive that analyzes ciphertext frequency across pages. As described in Section 5.2, the attacker can also infer frequency information through ciphertext collisions among pages of a single object during tracking, which we leave as future work.

Implementation of exploits. The development of our exploits is based on the SEV-Step framework [42] on Linux 5.19 to utilize its functionality of sending page fault events from the kernel space to the userspace. The framework implements a kernel space part based on the KVM kernel module for a malicious hypervisor and a user space part. The kernel space component modifies the `ioctl` interface provided by the `kvm` device to support additional commands, such as tracking all pages requested from the user space to mount the controlled-channel attacks. When a page fault occurs, the kernel space sends the information about this event such as the faulting GPA to the user space component. The user-space component acknowledges the event, allowing the CVM to resume execution. We extend the framework to include the contents of faulting pages when sending page fault events. Each faulting page is relocated to the target page frame to capture its re-encrypted contents, which are then compared against the ciphertexts of prevalent values. This comparison yields a bitmap of 256 bits, indicating which blocks match the prevalent values. If a page triggers a subsequent page fault, we also compare its new ciphertexts with the previous version to identify which encryption blocks have been updated.

8 Discussion

8.1 Confidential Computing Platforms

Table 4 compares the hypervisors' capabilities across different confidential computing platforms in relation to our attack primitives. AMD SEV-SNP allows ciphertext access to the hypervisor in 4th generation or older EPYC processors and only enables ciphertext hiding in 5th generation or newer EPYC processors with DDR-BF mode enabled DRAM [9, 11, 12, 23]. Moreover, ciphertext hiding is an opt-in policy for guest CVMs, which creates opportunities for misconfiguration, as CVM owners may not enforce it correctly or at all. SEV-SNP also allows the hypervisor to directly manipulate permission bits in the host page table, enabling controlled-channel attacks, and its page relocation mechanism remains exploitable at the time of writing. In contrast, Intel TDX enforces ciphertext access control on memory reads (Section 2) and prevents the hypervisor from manipulating page table permission bits, instead delegating these abilities to its secure runtime [24]. However, it exposes resource management commands such as `TDH.EXPORT.BLOCKW` and `TDH.MEM.RANGE.BLOCK` that a malicious hypervisor can misuse to simulate the effects of W or P bit clearing [37, 41]. This design still permits future firmware patches to restrict these interfaces due to its delegation model. TDX also supports page relocation for memory management, using `TDH.MEM.PAGE.RELOCATE`, similar to `SNP_PAGE_MOVE` in SEV-SNP. ARM CCA adopts stricter limits on hypervisor's control over guest resources. To our knowledge, we are unaware of comparable mechanisms to enable controlled-channel attacks or relocation primitives.

Confidential computing platforms in Table 4 encrypt memory using address-based tweaks [23]. Even if they restrict the hypervisor's access to ciphertexts via access control, recent works [23, 45] hypothesized that ciphertext side-channel attacks remain feasible through off-chip bus snooping [22], which gives the ability to read ciphertexts back to the adversary. We argue that the relocation primitive also amplifies such attacks. While traditional bus snooping captures only data-in-motion [14] during memory transactions, relocation actively exposes data-at-rest without the victim CVM's accessing memory. By forcibly relocating idle pages, the attacker can trigger re-encryption and force cache eviction, ensuring the ciphertexts are written back to DRAM and thus exposed on the bus.

8.2 Mitigation

Hardware implementation: Hardware vendors implementing confidential computing can mitigate issues caused by arbitrary page relocation by restricting the hypervisor's abil-

ity to perform such operations—for example, by disabling related management commands. This mitigation is expected to be backward-compatible, as it operates at the hypervisor management interface level by introducing additional checks before executing these commands. It mitigates attacks enabled by our primitive, even when ciphertext collisions do not occur at the same memory locations. However, it impacts the hypervisor's ability to manage resources. In particular, it prevents the hypervisor from moving pages for de-fragmentation and swapping pages to disk to reclaim memory, and does not address prior ciphertext side-channel attacks [23, 44].

Cloud service providers (CSPs) and tenants: The guest CVM should enforce the ciphertext-hiding guest policy. However, its enforcement requires specific hardware support (Section 8.1), which limits the availability of such instances. We surveyed the major CSPs: Microsoft Azure, AWS, and GCP, to ascertain whether their deployments of SEV-SNP restrict the hypervisor from accessing encrypted memory. At the time of writing, AWS confidential computing instances (`c6a`, `m6a`, and `r6a`) are provisioned with 3rd generation EPYCs, making it impossible to enable ciphertext-hiding due to unmet hardware requirement as specified in Table 4. Azure and GCP provision a mix of 3rd and 4th-generation EPYCs. However, the attestation report shows that ciphertext-hiding is not enabled on any of the instances for similar hardware constraints with AWS. In addition, GCP's 4th-generation EPYC platforms lack SNP support. At the time of writing, major CSPs do not support ciphertext-hiding and provide no interface for clients to request this protection through specifying corresponding guest policy, leaving them reliant on the platform's default configuration. We have also reported our primitive and the associated issue to the affected CSPs. In the future, CSPs should offer instances where ciphertext-hiding is both supported and configurable by the client.

Software mitigation: Patch the sparsity's encoding. Our attacks imply that the secrets already populated in the memory are still not safe, as the malicious hypervisor can force re-encryption to learn the population of the prevalent values on them. One can imagine modifying applications to exhibit less or no sparsity, but such changes are non-trivial, and can have significant implications on performance and compatibility.

8.3 Limitations

The current implementation uses only a single target page frame for simplicity, requiring each page to be relocated for majority status testing and then moved back. To improve scalability and eliminate repeated relocations, the attacker can learn ciphertexts for multiple page frames and pin private pages directly on them for in-place testing. To infer the ciphertexts of prevalent values across multiple frames, the attacker can either reapply Relocate-Vote or propagate known ciphertexts by relocating pages containing known prevalent value blocks. If these pages are idle and not actively accessed by the victim CVM, the propagation can proceed efficiently and stealthily without interrupting CVM execution. Furthermore, controlled-channel attacks inherently interrupt the CVM's execution. While our fine-grained tracking approach (Section 5.2) enables general-purpose monitoring of intermediate write operations while still allowing CVM progress, it incurs substantial performance overhead in scenarios such as OpenVDB construction. We leave the development of application-specific tracking strategies as future work, aiming to reduce the number of page fault events by adapting to the specific behavior of the target.

9 Related Work

Recent studies categorized ciphertext side-channel attacks into dictionary and collision attacks. Dictionary attacks [23, 25] build plaintext-ciphertext pairs for specific pages, including VM Save Area (VMSA) pages storing VM states during `VMEXIT`. However, the dictionary building can be mitigated by enhancing the security of these pages, such as adding a nonce for each encryption of the VMSA page. From the perspective of dictionary attacks, our primitive does not target specific pages in the CVM; instead, the attacker can learn the ciphertexts of prevalent values on arbitrary page frames, and the ciphertexts are learned with randomly sampled pages belonging to the CVM. Thus, the dictionary building cannot be mitigated by previous fixes for the dictionary attacks. From the perspective of collision attacks, previous works rely heavily on ciphertext collisions at precise locations. For example, CipherSteal [44] also exploits the ReLU activation information but relies on collision with previous zero-initialization behavior. On the contrary, our primitive can use zeros across the system to generate collision. Since co-located legacy software can widely utilize sparsity and may have the behavior of zeroing pages, only randomizing the initialization of the ReLU's buffers does not prevent the leakage. In summary, our attack introduces another dimension of collision by exploiting frequency information spatially, detecting collisions with prevalent values without observing repeated values written into specific locations.

10 Conclusion

In this paper, we present a novel primitive, Relocate-Vote, enabling the frequency analysis for encrypted values across memory pages. Using this primitive, we launch attacks to leak sensitive data on three different targets: ASLR, the OpenVDB library, and sparse LLMs. We also propose mitigations to alleviate the impact of this attack. Our works highlights that ciphertext side channels continue to pose a risk to confidential computing architectures and effective mitigation requires collaboration between hardware vendors, CSPs, and tenants.

Acknowledgments

We thank our anonymous reviewers and our shepherd for their improvement suggestions during the peer-reviewing process. We would also like to acknowledge the in-depth technical advice by Andy Jones (AMD), as well as Andrew Paverd and Giovanni Cherubin (Microsoft Research), Andrew Warfield (AWS), and Andrew Baumann (Google) for their invaluable assistance with the responsible disclosure process.

This research was supported in part by funding from NSERC Discovery Grants RGPIN-2018-05931 and RGPIN-2023-04796, and NSERC-CSE Research Communities Grant ALLRP 588144-23. Researchers funded through the NSERC-CSE Research Communities Grants do not represent the Communications Security Establishment Canada or the Government of Canada. Any research, opinions or positions they produce as part of this initiative do not represent the official views of the Government of Canada.

Ethics Considerations

We have responsibly disclosed our findings to the affected hardware vendor (Section 1) and cloud service providers (Section 8.2). All our attacks are demonstrated with open-source datasets or models (e.g., `CT 500` dataset[1] in Section 5.4 and the `ReLULlama2-13b` model[2] in Section 6.2) in local systems, avoiding harms to any user data or production systems.

Open Science

Description. The artifact includes the following: (1) a kernel implementation based on SEV-Step, supporting page relocation as described in Section 7.1, and extended with functionality for online tracking, as detailed in Section 7.2; (2) a kernel module that demonstrates online-phase ciphertext learning of zero as a global prevalent value (Section 7.2); (3) implementations of the exploits presented in Section 4, Section 5, and Section 6. In this paper, Table 2, Table 3, Table 6, Figure 8, Figure 9, Figure 11, and Figure 12 are produced by our artifacts. Figure 8 requires manual assembling of automatically generated pieces of objects according to Section 5.3. External datasets referenced in this paper, including the `CT 500` dataset and the `ReLULlama2-13b` model, are packaged with the artifact under the terms of their respective licenses.

Accessibility. We make the artifacts publicly available upon the lifting of the embargo on August 12th, 2025 at https://doi.org/10.5281/zenodo.15609905.

[1] https://www.kaggle.com/datasets/crawford/qureai-headct

[2] https://huggingface.co/PowerInfer/ReluLLaMA-13B-PowerInfer-GGUF

References

[1] Academy Software Foundation. ACM SIGGRAPH recognizes Ken Museth for OpenVDB. https://www.aswf.io/blog/acm-siggraph-recognizes-ken-museth-for-openvdb/, 2023.

[2] Advanced Micro Devices, Inc. *SEV Secure Nested Paging Firmware ABI Specification (Revision 1.56)*, 2024.

[3] Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. 2018. *arXiv preprint arXiv:1610.01644*, 2018.

[4] Yahya Alzahrani and Boubakeur Boufama. Biomedical image segmentation: a survey. *SN Computer Science*, 2(4):310, 2021.

[5] Amazon Web Services. AMD SEV-SNP for Amazon EC2 instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html, 2024.

[6] AMD SEV Team. SEV-SNP Support in the Linux Kernel. https://github.com/AMDESE/linux/tree/snp-host-latest, 2023.

[7] Yonatan Belinkov. Probing classifiers: Promises, shortcomings, and advances. *Computational Linguistics*, 48(1):207–219, 2022.

[8] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, 2014.

[9] 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.

[10] Google Cloud. Confidential VM overview. https://cloud.google.com/confidential-computing/confidential-vm/docs/confidential-vm-overview, 2024. Accessed: 2025-06-10.

[11] Kjersten Criss, Kuljit Bains, Rajat Agarwal, Tanj Bennett, Terry Grunzke, Jangryul Keith Kim, Hoeju Chung, and Munseon Jang. Improving memory reliability by bounding dram faults: DDR5 improved reliability features. In *Proceedings of the International Symposium on Memory Systems*, pages 317–322, 2020.

[12] Dong Du, Bicheng Yang, Yubin Xia, and Haibo Chen. Accelerating extra dimensional page walks for confidential computing. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 654–669, 2023.

[13] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural networks*, 107:3–11, 2018.

[14] Ali Fakhrzadehgan, Prakash Ramrakhyani, Moinuddin K Qureshi, and Mattan Erez. SecDDR: Enabling low-cost secure memories by protecting the ddr interface. In *Proceedings of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 14–27. IEEE, 2023.

[15] Alberto Fernández, Salvador Garcia, Francisco Herrera, and Nitesh V Chawla. SMOTE for learning from imbalanced data: progress and challenges, marking the 15-year anniversary. *Journal of artificial intelligence research*, 61:863–905, 2018.

[16] Apache Software Foundation. Apache HTTP server. https://httpd.apache.org/, 2025.

[17] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *Proceedings of the Symposium on Network and Distributed System Security*, volume 17, page 26, 2017.

[18] Wes Gurnee and Max Tegmark. Language models represent space and time. *arXiv preprint arXiv:2310.02207*, 2023.

[19] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[20] Buddhi Herath, Markus Laubach, Sinduja Suresh, Beat Schmutz, J Paige Little, Prasad KDV Yarlagadda, Dietmar W Hutmacher, and Marie-Luise Wille. The development of a modular design workflow for 3D printable bioresorbable patient-specific bone scaffolds to facilitate clinical translation. *Virtual and Physical Prototyping*, 18(1):e2246434, 2023.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[22] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[23] 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 *Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P)*, pages 337–351. IEEE, 2022.

[24] Mengyuan Li, Yuheng Yang, Guoxing Chen, Mengjia Yan, and Yinqian Zhang. SoK: Understanding design choices and pitfalls of trusted execution environments. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1600–1616, 2024.

[25] 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 *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, pages 717–732, 2021.

[26] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja Vu: Contextual sparsity for efficient LLMs at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[27] Memcached. Memcached. https://memcached.org/, 2025.

[28] Microsoft Azure. Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute.

[29] Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo C Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. ReLU strikes back: Exploiting activation sparsity in large language models. *arXiv preprint arXiv:2310.04564*, 2023.

[30] Ken Museth, Nick Avramoussis, and Dan Bailey. OpenVDB. In *ACM SIGGRAPH 2019 Courses*, pages 1–56. 2019.

[31] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[32] nginx. nginx. https://nginx.org/, 2025.

[33] Oracle. MySQL. https://www.mysql.com/, 2025.

[34] Alon Orlitsky and Ananda Theertha Suresh. Competitive distribution estimation: Why is Good-Turing good. *Advances in Neural Information Processing Systems*, 28, 2015.

[35] Qure.ai. Development and validation of deep learning algorithms for detection of critical findings in Head CT scan. https:

//web.archive.org/web/20220926004214/http://headctstudy.qure.ai/explore_data, Sep 2022. Accessed: 2025-06-10. Archived at Wayback Machine.

[36] Redis. Redis. https://redis.io/, 2025.

[37] Pradyumna Shome. Closing the Intel TDX page fault side channel, or, the case for TDEXIT notify. https://collective.flashbots.net/t/closing-the-intel-tdx-page-fault-side-channel-or-the-case-for-tdexit-notify/3775, 2024. Accessed: 2025-06-10.

[38] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade GPU. *arXiv preprint arXiv:2312.12456*, 2023.

[39] SparseLLM. SparseLLM/ReluLLaMA-13B. https://huggingface.co/SparseLLM/ReluLLaMA-13B, 2024. Accessed: 2025-06-10.

[40] Confidential Computing Summit. Confidential computing summit 2024 agenda. https://www.confidentialcomputingsummit.com/agenda, June 2024.

[41] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. TDXDown: 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.

[42] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A single-stepping framework for AMD-SEV. *arXiv preprint arXiv:2307.14757*, 2023.

[43] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*, pages 640–656. IEEE, 2015.

[44] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. CipherSteal: Stealing input data from TEE-shielded neural networks with ciphertext side channels. In *Proceedings of the 2025 IEEE Symposium on Security and Privacy (S&P)*, pages 79–79. IEEE Computer Society, 2024.

[45] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. HyperTheft: Thieving model weights from TEE-shielded neural networks via ciphertext side channels. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4346–4360, 2024.

A The Page State Machine Design in AMD SEV-SNP

Secure Encrypted Virtualization with Secure Nested Paging (SEV-SNP) [2] enhances security by providing integrity protection through additional page-level metadata, which tracks page ownership and enforces guest-side validation before a guest CVM can use a page. These mechanisms prevent unauthorized memory modifications to CVMs and mitigate re-mapping attacks. The ownership tracking and validation mechanisms operate as a state machine, with various metadata fields defining the page states. Table 5 shows several states involved in the discussion later. "—" indicates that the field does not affect the state.

Each page is associated with an entry in the reverse-mapping (RMP) table, which tracks its state. The state transition can be achieved by hypervisor-issued `RMPUPDATE`, CVM-issued `PVALIDATE`, or hardware-assisted commands such as `SNP_PAGE_MOVE`. The hardware needs to assist some operations because the hypervisor cannot access the necessary information to perform the operations, such as relocating a page, since the hypervisor cannot access the encryption key of the CVMs. The ownership tracking mechanism ensures that only the designated owner can modify a page, thereby preventing the hypervisor from tampering with pages assigned to a guest CVM. This is implemented by requiring the hypervisor to explicitly assign a page through the `RMPUPDATE` operation, which updates the `assigned` bit in the RMP table. Once assigned, the page's ownership transitions to the guest CVM, rendering it cannot by modified by the hypervisor. To counter re-mapping attacks, SEV-SNP enforces a guest-side validation mechanism. This process mandates the guest CVM to validate any new page assigned to it before use. This ensures that a page cannot be remapped without the guest's knowledge or consent since the `validated` bit of the new page can only be set by the CVM through `PVALIDATE`, thereby upholding the integrity of the CVM's memory.

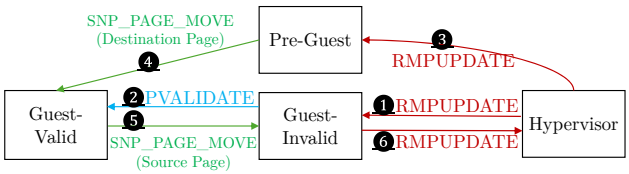


Figure 13: A subset of transition functions in the page state machine of AMD SEV-SNP.

Figure 13 illustrates a part of the transitions in the state machine. In the state machine, if a hypervisor page is assigned to a guest by issuing the `RMPUPDATE` command to update its `assigned` bit and the `ASID` field, it is transitioned into the

Table 5: Page state definitions.

Page State	Assigned	Validated	ASID	Immutable	GPA	VMSA
Hypervisor	0	0	0	0	—	—
Pre-Guest	1	0	>0	1	—	—
Guest-Invalid	1	0	>0	0	—	—
Guest-Valid	1	1	>0	0	—	—
...	...					

Guest-Invalid state. The guest CVM must validate it by triggering the `PVALIDATE` instruction during its execution before it can use the page ❶-❷. The official documentation suggests that the guest CVM only validates a guest physical address (GPA) once, and the subsequent validation requirements for the same GPA could indicate a potential attack, where a malicious hypervisor maps the guest page of GPA to a different invalid page.

With this suggestion, a GPA is expected to be validated only once, which implies that valid operations, including hardware-assisted commands, should not set a validated GPA to an invalidated state. Under this implication, a page frame initially belonging to the hypervisor can bypass the guest VM's validation if it is assigned as the destination of a page move of a validated private page, supported by management command `SNP_PAGE_MOVE`. The hypervisor can compel a page in the Hypervisor state to transition into the Pre-Guest state using `RMPUPDATE` and subsequently to the Guest-Valid state via `SNP_PAGE_MOVE` ❸-❹. The hardware re-encrypts the page with the new tweak values derived from the addresses in the new page frame. The source page is then transitioned to the Guest-Invalid state with the effect of the `SNP_PAGE_MOVE` command ❺ and can be further turned into the Hypervisor state via `RMPUPDATE` ❻.

In summary, once the guest validates a guest page, the hypervisor can arbitrarily move it to arbitrary hypervisor-owned page frames while maintaining its validated state without requiring the guest's permission or validation. This allows the attacker to control the bits in each encryption block's address except for the low 12 bits. Another command, `SNP_PAGE_SWAP_IN`, has similar effects and can be utilized similarly.

B Guest Page Walk and Attacker's Visibility

On the x86 architecture, the page table hierarchy typically comprises four levels: the Page Global Directory (PGD), Page Upper Directory (PUD), Page Middle Directory (PMD), and Page Table Entry (PTE). Each entry at these levels holds the physical address of the subsequent level's page, thus forming a chain of references to progressively deeper levels. Each page level contains 512 entries. Virtual address translation necessitates a page walk when there is a TLB miss, and on x86, this operation is handled by the hardware. During this page walk, the hardware attempts to set the access bit of each accessed page table entry, regardless of whether it has already been set. Thus, if the hypervisor clears the W bit in the host page table, the hardware write triggers a page fault of write protection violation. The CVM halts execution to report the corresponding faulting GPA of the page table page with the lowest 12 bits cleared, requesting the hypervisor to lift the write protection by setting the W bit. In conclusion, the page walk sequence can be exposed by the controlled-channel-based method by clearing the W bit.

C Common Layout of PGD Pages

Figure 14 shows the typical layout of the PGD pages, illustrating the 16x16 bitmaps of PGD pages of 100 random processes on a CVM. Each bitmap has 256 blocks representing a PGD page, and each encryption block accommodates two 8-byte page table entries. The white blocks represent 16-byte zeros, and the red blocks indicate at least one mapped entry in an encryption block. The second half represents the mappings in the kernel space, so they share the layout.

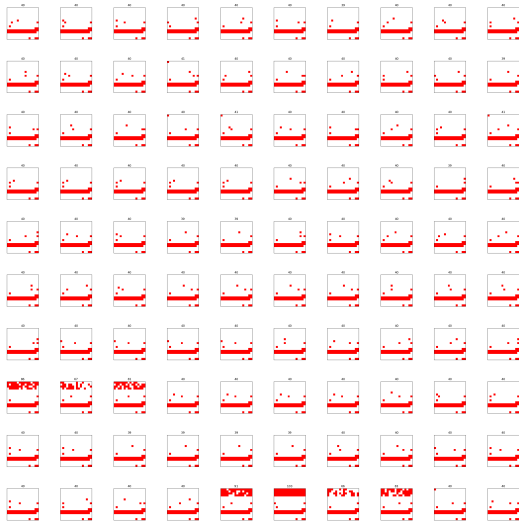


Figure 14: The memory layouts on 100 randomly sampled PGD pages.

Table 6: $\frac{m}{n}$: Predicted coverage (m) and actual coverage (n).

	1%			5%			50%		
	PUD	PMD	PTE	PUD	PMD	PTE	PUD	PMD	PTE
nginx	$\frac{97.46}{97.45}$	$\frac{100}{100}$	$\frac{98.93}{99.37}$	$\frac{98.77}{98.74}$	$\frac{100.0}{100.0}$	$\frac{100.0}{99.99}$	$\frac{99.93}{99.94}$	$\frac{100.0}{100.0}$	$\frac{99.99}{99.99}$
apache	$\frac{96.73}{97.24}$	$\frac{89.93}{87.60}$	$\frac{88.80}{88.20}$	$\frac{98.78}{98.71}$	$\frac{98.24}{97.81}$	$\frac{99.99}{99.99}$	$\frac{99.95}{99.94}$	$\frac{99.25}{99.10}$	$\frac{99.99}{100.0}$
mysql	$\frac{97.27}{97.20}$	$\frac{96.47}{97.12}$	$\frac{99.93}{100.0}$	$\frac{98.10}{98.10}$	$\frac{99.26}{99.60}$	$\frac{100.0}{100.0}$	$\frac{99.90}{99.92}$	$\frac{99.88}{99.77}$	$\frac{100.0}{100.0}$
redis	$\frac{98.00}{97.34}$	$\frac{100.0}{100.0}$	$\frac{100.0}{100.0}$	$\frac{98.60}{98.72}$	$\frac{100.0}{100.0}$	$\frac{100.0}{100.0}$	$\frac{99.93}{99.91}$	$\frac{100.0}{100.0}$	$\frac{100.0}{100.0}$
memcached	$\frac{97.07}{97.36}$	$\frac{99.47}{99.19}$	$\frac{100.0}{100.0}$	$\frac{98.37}{98.28}$	$\frac{99.85}{99.75}$	$\frac{100.0}{100.0}$	$\frac{99.90}{99.90}$	$\frac{99.94}{99.95}$	$\frac{100.0}{100.0}$

D Good-Turing's Prediction and Actual Coverage

Table 6 presents the predicted coverage by the Good-Turing estimator and actual coverage for each page table level of PUD, PMD, and PTE across applications with different sample sets. It demonstrates that the Good-Turing estimator provides accurate predictions of the comprehensiveness of the fingerprint sets at different stages of fingerprint collection.