SNPeek: Side-Channel Analysis for Privacy Applications on Confidential VMs

Ruiyi Zhang[1,2], Albert Cheu[2], Adria Gascon[2], Daniel Moghimi[2], Phillipp Schoppmann[2], Michael Schwarz[1], and Octavian Suciu[2]

[1]CISPA Helmholtz Center for Information Security
[2]Google

***Abstract*—Confidential virtual machines (CVMs) based on trusted execution environments (TEEs) enable new privacy-preserving solutions. Yet, they leave side-channel leakage outside their threat model, shifting the responsibility of mitigating such attacks to developers. However, mitigations are either not generic or too slow for practical use, and developers currently lack a systematic, efficient way to measure and compare leakage across real-world deployments.**

In this paper, we present SNPeek, an open-source toolkit that offers configurable side-channel tracing primitives on production AMD SEV-SNP hardware and couples them with statistical and machine-learning-based analysis pipelines for automated leakage estimation. We apply SNPeek to three representative workloads that are deployed on CVMs to enhance user privacy—private information retrieval, private heavy hitters, and Wasm user-defined functions—and uncover previously unnoticed leaks, including a covert channel that exfiltrated data at 497 kbit/s. The results show that SNPeek pinpoints vulnerabilities and guides low-overhead mitigations based on oblivious memory and differential privacy, giving practitioners a practical path to deploy CVMs with meaningful confidentiality guarantees.

I. Introduction

Cloud providers now offer *confidential virtual machines* (CVMs) based on hardware architectures such as AMD SEV-SNP, and Intel TDX. These CVMs encrypt guest memory and enforces security via the hardware, allowing a tenant to run unmodified binaries while keeping data hidden from the hypervisor and other co-tenants. Unfortunately, Intel, AMD, and ARM (e.g., CCA) *explicitly exclude* leakage through page-table activity and processor-cache state from their CVM's threat model. Therefore, such side-channel attacks based on page tables [1] and caches [2], [3] can track memory accesses at 4 kB and 64 B granularities, respectively. They have been very successful at, e.g., inferring cryptographic keys [4], [5]. That setting matches the traditional notion of (architectural) side-channel attack, where the adversary's goal is to extract a private key, e.g., a signing key, from a confidential VM by exploiting or inducing side-channel leakage. While implementing successful mitigations has proven challenging, best practices such as constant-time code are well understood for *concrete* cryptographic applications, e.g., RSA-based signature schemes. This is the result of a fruitful line of security research that provided a feedback mechanism to chip manufacturers.

From cryptographic applications to privacy-preserving data analyses. A recent trend in industry involves adopting CVMs for user data processing, e.g., computing user statistics, distributed secure computation, and oblivious data retrieval [6], [7], [8], [9]. Just like with cryptographic code, these data-driven applications inherit a large attack surface when deployed in CVMs [10], [11], and mitigations are delegated to the application developers.

Privacy threats in these data-driven applications are less precisely defined and more challenging to quantify compared to those in cryptographic applications. An attacker clearly defeats a cryptographic application when they recover a pseudorandom key. In contrast, when an attacker recovers an input to a data-processing application, their success must be evaluated in light of both their prior knowledge and the goals of the application itself. Without **a rigorous formalization of the threat model and attack success**, it will be tempting to fall back to impractical mitigations like constant-time code for *general* software [12], [13], [14], [15], instead of optimizing defenses based on the specific workload. Aside from a strong definition, practitioners also need a **systematic way to quantify, measure, compare, and reduce the leakage of real workloads in the deployed scenario**.

In this paper, we provide formal definitions and an evaluation framework called **SNPeek**. Our goal is to enable the systematic investigation of side-channel leakage in privacy-preserving workloads, and potential mitigations.

Quantifying Attacker's Success (Section III). Taking inspiration from the literature on cryptography and differential privacy, we define and measure success of privacy attacks in a relative sense: assuming the attacker has some prior knowledge about a target's data, we measure success by comparing the attacker's probability of correctly guessing the data before and after the attack. If the prior probability of a successful guess is already high, our formulation captures the intuition that there is not much more information an attacker can learn. Additionally, our model accounts for the fact that the party connecting the CVM to the outside world can introduce

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240699
www.ndss-symposium.org

Sybils, i.e., arbitrarily well-crafted values that can trigger more leakage than naturally-occurring inputs.

Automated Side-channel Extraction and Analysis (Section IV). Our open-source toolkit, currently implemented for AMD SEV-SNP, consists of a *trace extraction* and a *trace analysis* phase. In the former, SNPeek records low-noise page-table [1], [16], [17] and cache traces [18], [2], [3], and optionally also performance-counter values [19], [20] and ciphertexts [21], [22], from commodity SEV-SNP guests. We introduce a noise-free and efficient Multi-Prime+Probe attack targeting 64 cache sets on AMD CPUs with a non-inclusive last-level cache by exploiting model-specific registers that allow restriction of the L3 cache. Our optimizations significantly reduce measurement noise and improve the practicality of the cache attack. Additionally, we devise filtering strategies that restrict trace collection to the relevant part of the application, thereby speeding up measurement. In the analysis phase, SNPeek analyzes those traces with a set of predefined statistics and machine-learning models for automated side-channel traces analysis. These predefined models allow for easy pinpointing of the leakage source and experimentation with attackers of various capabilities.

Evaluation on Real-World workloads (Sections V, VI, VII). We apply SNPeek to evaluate side-channel leakage of three real-world privacy applications that are executed on top of CVMs: Private Information Retrieval (PIR), Private Heavy Hitters (PHH), and User Defined Functions (UDF). In PIR, a party wishes to retrieve an element from a remote database without letting the database's maintainer(s) learn which element was accessed. TEE-based PIR implementations are available as open-source projects such as Project Oak [23] and the Signal messenger [24]. We analyze the effectiveness of mitigations such as ORAM [25], [24] and demonstrate how our framework can uncover and pinpoint subtle leakage. Surprisingly, we show that even constant-time ORAM may exhibit leakage when deployed on AMD SEV-SNP due to ciphertext side channel leakage.

We additionally demonstrate how SNPeek can evaluate the privacy guarantee of a PHH application as implemented by the TensorFlow Federated project [26] and deployed by Google [8]. In this application, a large number of personal devices (e.g., smartphones) hold sensitive data, such as location or browser history, and the service provider wishes to identify frequently occurring entries in this distributed data store in a differentially private manner. We show that the Tensorflow-Federated implementation [27], which is not leakage-aware, is vulnerable to a privacy attack due to its data-dependent execution behavior when deployed on AMD SEV-SNP. We present a series of examples to illustrate the use of our framework to detect issues, develop and evaluate defenses, and advanced attacks. Along the way, we also introduce a partial mitigation based on differential privacy that might be of independent interest.

Finally, we demonstrate how SNPeek can evaluate private user-defined functions. In-memory data stores such as those used in PIR or PHH may also support custom queries via a user-defined function (UDF) [28], [29]. For example, in the context of Google's Privacy Sandbox [11], UDFs based on the Wasm language are written by AdTechs to customize higher-level (privacy-preserving) aggregations about end-users' web browsing behavior [30], [23]. Private UDFs introduce an attack scenario where the attacker can not only collect side-channel traces outside the CVM, but also introduce new queries on processed data sources and efficiently steal data via a covert channel. Our results show that a covert-channel attack can leak data from a UDF inside the Wasm language runtime [29] to a colluding hypervisor at a rate of at least 497 kbit/s.

Contributions. We summarize our contributions as follows.

1) We introduce SNPeek, a modular framework that gathers various side-channel traces, including a novel noise-resilient Prime+Probe, from unmodified AMD SEV-SNP guests and provides different trace filters to reduce collection overhead. The framework includes statistical and machine-learning models to automatically analyze gathered side-channel traces to enable leakage estimates for non-domain experts.
2) We introduce and motivate rigorous quantitative notions of privacy leakage via side-channel in the presence of a malicious attacker, i.e., the attacker's advantage. Our notion is inspired by the privacy attacks literature and can be easily estimated empirically using SNPeek's ML components.
3) We evaluate three representative privacy workloads—PIR, private heavy hitters (PHH), and user-defined functions (UDFs) based on Wasm—and show how SNPeek guides the design of effective, low-overhead mitigations.

Although our toolkit is currently implemented on AMD SEV-SNP, we anticipate many of our attacks carry over to other vendors with minimal changes.

Responsible Disclosure. Our research follows established responsible disclosure guidelines. We notified maintainers of all open-source projects whose applications showed vulnerabilities under our framework—specifically Project Oak [23], TensorFlow Federated [26], and the Privacy Sandbox [29]. Each project acknowledged the security impact of software-based side-channel attacks and indicated ongoing work to strengthen its privacy protections.

Availability. The source code of SNPeek is open-sourced at https://github.com/google-parfait/cvm-side-channel-analysis.

II. Preliminaries

A. Confidential VMs

Confidential VMs are based on hardware-based trusted execution environments, such as AMD SEV-SNP [31] or Intel TDX [32]. They rely on hardware-based access control and memory encryption to prevent other VMs and privileged software (hypervisor, BIOS) from accessing the memory of a trusted domain (a CVM instance). Additionally, memory is encrypted as soon as it leaves the CPU. The operating system is only responsible for the availability of trusted workloads

(e.g., scheduling workloads, mapping memory, and handling I/O), and can thus be untrusted.

CVMs additionally rely on a hardware-based root of trust (external to the CPU core) and a remote-attestation protocol to guarantee the integrity of the software and hardware components responsible for executing a trusted domain. Therefore, before a user sends encrypted data to the CVM, they can verify that the data is processed by genuine hardware and the right software components, including the latest firmware and microcode security patches.

B. Software-Based Side Channels

Software-based side-channel attacks exploit shared resources and exposed system interfaces to leak information about computation of other users on the system. Some of the attack primitives that are relevant to our work focusing on CVMs include:

Page table. Controlled-channel attacks target page tables, allowing a malicious hypervisor to track a program's memory access pattern at page-level granularity (typically 4 kB). This attack vector applies to various CVM platforms. For instance, a hypervisor can unmap a guest memory page on AMD SEV-SNP. While Intel TDX (and ARM CCA) restricts such direct page-table manipulation, similar leakage is achievable: privileged software can leverage the TDX module to block and unblock memory ranges, resulting in a similar VM exit as soon as a trusted domain accesses the memory ranges [1].

Cache. Cache attacks, such as the Prime+Probe technique [33], target shared caches in modern CPUs. These remain applicable to CVM platforms like AMD SEV-SNP, Intel TDX, and ARM CCA, given their reliance on shared-cache architectures. In a Prime+Probe attack, the attacker fills a cache set with known addresses and waits for the victim to access data mapping to the same cache set. After the victim's access, the attacker detects which parts have been evicted by measuring timing differences of re-accessing its own addresses.

HPC. Hardware performance counters (HPCs) are special registers in modern CPUs that track various microarchitectural events, such as cache hits, misses, and branch predictions. Privileged attackers can use performance counters to gather detailed information to infer sensitive information, such as cryptographic keys or execution-flow patterns [19].

Ciphertext visibility. Ciphertext side-channel attacks [21] exploit the memory encryption scheme in AMD SEV-SNP. Each 16-byte-aligned memory block is encrypted individually, using a tweak value derived from its physical address. At a specific address, the same plaintext always produces the same ciphertext. Although SEV-SNP aims to provide confidentiality and integrity, a malicious hypervisor can read the encrypted memory. By observing changes in ciphertexts, the attacker can infer changes in the underlying plaintexts, beyond learning that a given region in memory did change.

System-level mitigations. Table I presents the current state of system-level mitigation for our attack vectors. Ciphertext side-channel attacks [21] are specific to SEV-SNP, and AMD has provided software workarounds that make constant-time code even harder to implement [39]. Intel TDX and ARM CCA prevent the hypervisor from accessing guest-encrypted memory [36]. AMD plans to address the leaks from ciphertext and HPCs on Zen 5 processors using ciphertext hiding [34] and PMC virtualization [35], respectively. Currently, neither is supported in KVM. For ARM CCA, which defines a broader architecture, performance monitoring virtualization for the trusted realm is platform dependent. However, cache attacks and page-level leakage remain unmitigated across SEV [31], TDX [36], [37], and CCA [40]. They are considered out of scope by the vendors. Thus, from the vendor's perspective, attacks on a given workload leveraging these side-channels are the responsibility of the application developer.

TABLE I: ● : vulnerable, ○ : mitigated, ◐ : partially, ⊜ : mitigation planned

Platform	Page-level	Cache-level	Ciphertext	HPC
SEV-SNP Zen3/4	●	●	●	●
SEV-SNP Zen5	●	●	⊜ [34]	⊜ [35]
TDX	● [36]	● [36], [37]	○	◐ [38]
CCA	●	●	○	◐

C. Differential Privacy (DP)

Suppose there are n distinct inputs $X = X_1, \ldots, X_n$ to a computational service S, where each input X_i may be a sensitive value (i.e., proprietary information or personal attribute) of a distinct input provider. Let $V_S^A(X)$ denote attacker A's view of that service when X is given as input. In the textbook *central model*, the view is the output of the service, like an estimate of a mean or a table of synthetic data [41]. S ensures (ε, δ)-DP against A if, for any X, X' that differ on any one input and any possible Y, $\mathbb{P}\left[V_S^A(X) \in Y\right] \leq e^{\varepsilon} \cdot \mathbb{P}\left[V_S^A(X') \in Y\right] + \delta$.

We emphasize that the guarantee must hold for *all* neighboring inputs X, X'. This effectively means that the attacker has narrowed down a target's input to one of two different values and controls all other inputs (*Sybils*).

There is considerable risk in underestimating the scope of an adversary's view. As highlighted in prior work [42], [43], consider a service that outputs the same mean estimate on two neighboring datasets but whose running time differs dramatically: an adversary can deduce the target's input whenever it can measure elapsed time. Haeberlen et al. [42] attempt to mitigate this by modeling the adversary as only able to access S via a network connection and a restricted query language. Meanwhile, Ratliff and Vadhan [43] carefully reason about sensitivity and inject random-length delays, following the pattern of the Laplace and Gaussian mechanisms. We note that our definition of DP is a strict generalization of the one by Ratliff and Vadhan [43], since our adversary's view can encompass more than the timing side channel (e.g., memory access patterns).

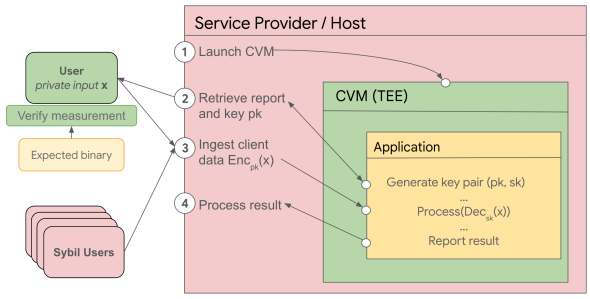


Fig. 1: The user requests an attestation report and a public key before sending encrypted data to the isolated CVM. The service provider and other users are untrusted. The CVM and application binary is trusted, but it may have the ability to execute user-defined queries from an untrusted source.

III. Defining Side-Channel Privacy Attacks

Here, we formalize how an attacker in the CVM threat model can recover information about sensitive inputs using side channels [44]. This includes personal attributes that are processed by software running in the CVM (e.g., location, webpage visits), not just cryptographic keys.

We assume a shared cloud environment where the hypervisor and other VMs are untrusted. Figure 1 shows the structure of a CVM-based data analysis system. Data and/or custom queries are ingested from *input providers*, also known as *users*. An untrusted *service provider*—the party operating the data analysis service—seeks to reconstruct more information about the inputs than what is contained in the output. We use the term *attacker* as a short synonym for the service provider. We assume no collusion between the service provider and the hardware manufacturer. In Section VII, we extend the model to assume that the attacker can provide custom queries to the protected key-value service.

Sybils [45] are a significant feature in our model. We do not assume public key infrastructure free from the influence of the service provider, which means the attacker can generate fake identities to take over a service: they can inject maliciously-generated inputs, and suppress honest inputs. Sybil attacks in distributed data analysis are a known issue, particularly in the context of federated learning [7]. The attacks we design reaffirm this deep challenge. However, we acknowledge a practical limitation against hardened systems using third-party privacy gateways [46]. To guarantee a victim's input is routed to a compromised machine for side-channel analysis, an *attacker* must collude with the gateway, as monitoring all endpoints is undesirable. This countermeasure is thus effective against *attackers* when the external party gateway remains uncompromised.

Like previous work on software-based attacks, we leave physical attacks out of the scope [47], [48], assuming appropriate physical security is in place. Likewise, we exclude CPU bugs such as transient-execution attacks [49], [50], [51], [52] and CacheWarp [53], and software-based fault attacks like Rowhammer [54] and Plundervolt [55]. We also assume that all applications are protected against rollback attacks, and therefore, honest client contributions cannot be duplicated or replayed by the attacker without aborting the application.

We sketch the steps of a generic data-analysis service by breaking it into an *offline* and *online* phase. The offline phase describes what happens before any interactions.

Offline phase:

1) Input Preparation: The input providers generate their inputs. The attacker does not know any target's input with complete certainty, but might have some prior knowledge; we model that input as being drawn from a probability distribution known to the attacker.
2) Setup: The guest VM binary is made reproducible for the purposes of verifiability. Here, an attacker has full control to assess the behavior of the binary on their TEE-supporting hardware (e.g., debug mode) but does not have access to the (secret) data. In particular, the attacker can obtain statistical information to characterize secret inputs given side-channel information, either by "manual" inspection, or by training ML models.

Online phase:

1) Launch of CVM: The service provider triggers the initialization of the trusted environment.
2) Establishing trust: The CVM's attestation report and public key are forwarded by the service provider to the users, who validate the report.
3) Input Ingestion: The service provider forwards a stream of encrypted user data and/or user-defined functions for the CVMs. Here,
   a) the attacker can drop honest inputs and insert *Sybils*, specially-crafted inputs that are meant to trigger more side-channel leakage. However, we assume they do not duplicate or replay honest inputs.
   b) the attacker monitors side channels of the ingestion computation.
4) Report: the guest VM computes a plaintext output that the service provider relays to its recipient. The attacker monitors side channels of the report computation.

This gives a high-level idea of the actions an attacker can perform. Next, we provide a notation for the attacker's knowledge and define what it means for an attack to succeed.

A. Attacker Knowledge & Success

The attacker is not necessarily limited to the knowledge gleaned from the service's execution: they may have *prior information* about a target input provider. For example, they may know Alice is contributing URLs as input and that she only reads English. This means the attacker can rule out strings that are not URLs while also weighing URLs with a "jp" extension as less likely to have been visited by Alice. We will use W to denote the probability distribution that describes this prior knowledge about a target.

Although it is tempting to use the likelihood of reconstructing secrets (e.g., pseudorandom keys) as the metric of success, this is not always the correct choice: if the prior W is sufficiently skewed, the attacker may have a high probability

of a correct guess *even without looking at any side channels or CVM output*. In the case of URL visits, they are **not** uniformly random: if `example.com` is known to be $p\%$ of all visits, then the baseline "attack" that simply outputs `example.com` has a $p\%$ chance of being correct for a target. Thus, a high probability of a correct guess could be due to using leakage or reporting an argmax of a heavily-skewed W. As such, it is a poor metric to gauge success.

We take the stance that the quantity to measure is the *advantage* afforded by the attack over the baseline argmax-of-W strategy, the improvement in the probability of a correct guess[1]. We note that $\Omega(1)$ advantage is permissible in some applications; for example, DP computations already grant a "privacy budget" ε and we show $\varepsilon = \Omega(1)$ permits $\Omega(1)$ advantage (see next section). Otherwise, the advantage should be bounded by a negligible function.

B. Pairwise Distinguishability Attacks

Let us consider a specific class of prior W: those distributions over two values that give equal probability to each. This occurs when the attacker is evenly split between, say, whether the last URL entered by a target individual was `example1.com` or `example2.com`. Without the side-channel leakage, the baseline argmax-of-W strategy results in a guess that is right $1/2$ of the time. With the side-channel leakage, we would like to bound the *advantage* over that baseline chance. The adversary observes these side channels via the pairwise distinguishability game:

Definition 1 (Pairwise Distinguishability Game)**.** *Let $\mathcal{A}$ be an attacker, and let B be a binary executed in a CVM. Let $\mathsf{SC}^B_{\mathcal{A}}(D)$ denote the leakage function for $\mathcal{A}$ when running B on input D. Assuming uniform prior W over $\{x_0, x_1\}$, the distinguishability game proceeds as follows:*

1) *(Input Preparation) c is chosen randomly from $\{0, 1\}$, such that user's data x_{c} is x_0 or x_1 with a 1/2 chance*
2) *(Before Input Ingestion) Attacker chooses Sybils X*
3) *(Input Ingestion) Whole dataset D is formed by appending x_{c} to X*
4) *(After Report) $\mathsf{output}_{\mathcal{A}} := \mathcal{A}(\mathsf{SC}^B_{\mathcal{A}}(D))$*

The distinguishability advantage of attacker $\mathcal{A}$ is $\mathsf{Adv}_{\mathcal{A}} := \max(0, \Pr[\mathsf{output}_{\mathcal{A}} = \mathsf{c}] - 0.5)$. The maximum value for this is 0.5; we compute a *normalized advantage* $\mathsf{Adv}_{\mathcal{A}}/(0.5)$ that ranges from 0 to 1 (least to most successful attack). Note that we can derive a bound on this advantage when leakage $\mathsf{SC}^B_{\mathcal{A}}(D)$ satisfies DP:

Lemma 1 (DP bounds advantage)**.** *If the leakage of B guarantees (ε, δ)-DP, a pairwise distinguishability attack against B has advantage bounded by $(e^{\varepsilon} - 1)/4 + \delta/2$.*

The proof can be found in Appendix A. For an example, consider $\varepsilon = 0.5, \delta = 0.01$: advantage is bounded by < 0.17 which normalizes to < 0.34.

[1]For cryptographic keys, the baseline is close to zero, so the advantage of an attack is close to the probability of reconstructing keys.

Our definition of distinguishability attack can be compared to membership inference attacks [56], [57], [58]. In both cases, an attacker wants to learn a binary predicate about the target. The predicate is membership for membership inference, while our predicate concerns value.

C. Fingerprinting Attacks

One way to generalize pairwise distinguishability is k-wise distinguishability, where prior knowledge W covers a large set $\{x_1, x_2, \ldots, x_k\}$. We additionally refer to an *interest set* I. To continue our URL example, the adversary may be interested in URLs that end in country codes. The adversary has two objectives: to determine whether the target's URL is interesting and, if it is, to identify which interesting URL it is. The country code can serve as a hint about the target's location or language. Reconstruction of uninteresting URLs is not a priority.

Similar to the pairwise distinguishability attack, the prior W grants the adversary baseline strategies that do not involve side channels at all. Specifically, to determine whether the target x is in I, the baseline strategy is to report "interesting" if and only if the mass placed on set I by W is larger than the mass placed outside it. In our example, this amounts to comparing the prior probability of visiting a URL with a country code against that of visiting a URL without one. To fingerprint x assuming it is in I, the baseline strategy is to report the likeliest element according to the distribution W_I, which is W conditioned on I; this amounts to reporting the most frequently visited URL ending in a country code. Note that the success rate of the baseline interesting/not-interesting classifier is $s_c := \max(\mathop{\mathbb{P}}_{x \leftarrow W}[x \in I], \mathop{\mathbb{P}}_{x \leftarrow W}[x \notin I])$, while the baseline fingerprinting success rate is $s_f := \max_{i \in I} \mathop{\mathbb{P}}_{x \leftarrow W_I}[x = i]$.

With side-channel leakage, we again would like to bound the advantage over these baseline rates. The adversary observes side channels via the fingerprinting game:

Definition 2 (Fingerprinting Game)**.** *Assuming prior W, the fingerprinting game proceeds as follows:*

1) *(Input Preparation) Target user's data x randomly chosen according to W*
2) *(Before Input Ingestion) Attacker chooses Sybils X and chooses I*
3) *(Input Ingestion) Whole dataset D is formed by appending x to X*
4) *(After Report) $\mathsf{output}_{\mathcal{A}} := \mathcal{A}(\mathsf{SC}^B_{\mathcal{A}}(D), W, I)$*

The interest-classification advantage is $\mathsf{Adv}_{\mathcal{A}} := \max(0, \mathop{\mathbb{P}}_{x \leftarrow W}[(\mathsf{output}_{\mathcal{A}} == \text{"interesting"}) = x \in I] - s_c)$. We can normalize this advantage to the range $[0, 1]$ by dividing by its maximum value $1 - s_c$. The fingerprinting advantage is $\max(0, \mathop{\mathbb{P}}_{x \leftarrow W_I}[\mathsf{output}_{\mathcal{A}} = x] - s_f)$. We can again normalize by dividing by its maximum value $1 - s_f$.

Example 1 (Fingerprinting Game)**.** *Suppose an attacker picks I = {any example site $\neq$ 'example.com'} and the prior probability distribution for visiting a website is*

60% example.com	*10% example.co.jp*
10% example.co.uk	*20% other URLs*

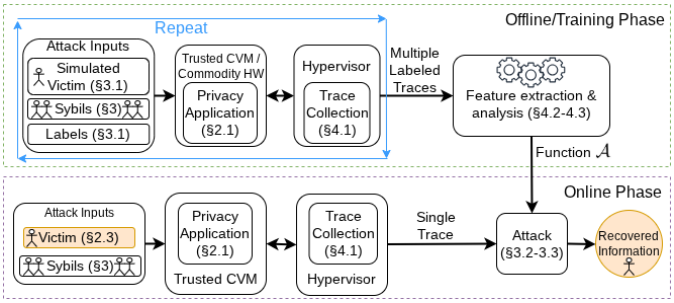


Fig. 2: The overview of SNPeek. The corresponding section numbers § are listed for each component. The offline phase consists of building an ML attacker model using labeled traces (Section IV-C). The online phase uses this model on a single CVM trace that includes victim data, to launch a privacy attack $\mathcal{A}$ that reveals this victim data.

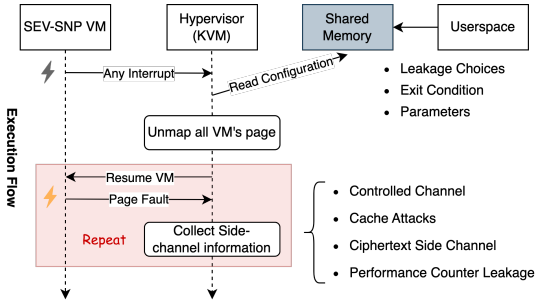


Fig. 3: The overview of SNPeek trace collection. A user-space controller uses shared memory to configure each collection run with a modified KVM module. Grey lighting marks any interrupt allowing the hypervisor to read the configuration; yellow lighting is a page fault interrupt as the hypervisor clears the present bit of guest pages.

The real site visited (x) is drawn randomly from W. The baseline chance of guessing (non-)membership in I is $s_c = \max(0.2, 0.8) = 0.8$; absent side-channels, the best guess is that the target is not in I. The baseline chance of reconstructing an element of I is 0.5; absent side-channels, there is an even chance between `example.co.uk` *and* `example.co.jp`.

If the side-channel leakage observed by the attacker ($\texttt{output}_{\mathcal{A}}$) leads to a 0.9 probability of guessing whether $x \in I$, then the interest-classification advantage is $0.1 = 0.9 - 0.8$. If the leakage grants a 0.7 chance of recovering an element of I, the attacker has a fingerprinting advantage of $0.2 = 0.7 - 0.5$.

In our empirical evaluation of TF-Federated's PHH implementation (Section VI), we set W to be the actual distribution of the data going into the CVM, thus assuming the adversary has perfect prior knowledge about the distribution. Note that this sets a high bar for what constitutes a successful side-channel attack to reconstruct a victim's input.

IV. SNPeek Framework

Figure 2 presents an overview of SNPeek. The offline phase allows developers to model attacks of various strengths and capabilities, and analyze privacy leakage under these instantiations. This is achieved by collecting a labeled dataset of traces corresponding to the attack, and using statistics and ML tools to build leakage-analysis tools. In the online phase, the leakage-analysis tool is used to conduct an attack $\mathcal{A}$ on a single trace containing the victim data, to quantify privacy leakage. This resembles an attacker who collects side-channel information offline to build a leakage detector for online use, on a victim CVM through a malicious hypervisor. We present how SNPeek implements the collection of different side-channel signals, feature extraction, and the leakage-analysis tools.

A. Trace Collection

Figure 3 gives an overview of the side-channel trace collection in SNPeek, which consists of a modified KVM module and a user-space controller, communicating through shared memory. Besides configuring trace collection runs (e.g., via number of traces, optimizations), the shared memory enables a developer to control the two key components of trace collection: *temporal resolution* – determining the frequency of side-channel event collection, and *spatial resolution* – the side-channels with different granularities.

Nevertheless, in contrast to cryptographic targets, there are significant challenges in automating trace collection and analysis in privacy applications. This requires strategies to improve collection speed by ignoring uninteresting parts of the execution flow, mapping only relevant code pages, and supporting generic analysis through various attack primitives. SNPeek achieves this through two *key insights* during temporal and spatial resolution, which allow developing general and platform-specific optimization strategies.

1) Temporal Resolution: We introduce our technique to synchronize the hypervisor's side-channel collection with the target running inside the SEV-SNP VM. The hypervisor requires a trigger point to halt the VM execution and control its execution. Well-known methods include inducing page faults [1], triggering interrupts with APIC timers [59], [60], or using a combination of both [61], [53]. APIC interrupts allow the attacker to pause the VM at short intervals, ensuring that only one instruction is completed after each context switch, known as single-stepping [59].

While single-stepping might seem like an obvious solution for temporal resolution, it is suboptimal for privacy attacks for two reasons. First, single-stepping is inherently slow. Each step requires at least one context switch, and it takes even longer when zero stepping occurs (i.e., no progress after a context switch). For example, our fingerprinting testcase (see Section VI-B) completes in $0.46\,\mathrm{ms}$ without monitoring. With a filtered controlled channel, SNPeek collects 1.06×10^5 entries in $0.56\,\mathrm{s}$, incurring approximately 1217x overhead, which

remains acceptable for offline auditing. In contrast, after integrating single-stepping, SNPeek generates 2.8×10^5 zero-step, 1.6×10^6 single-step, and 6.7×10^5 two-instruction entries in 8.67 s, yielding an approximately 18 848x overhead. Such high overhead typically necessitates narrowing analysis to a limited code section, as in cryptographic libraries [22], [62], [63], [64]. However, privacy-preserving applications often have code bases much larger than cryptographic libraries, making such an analysis prohibitively time-consuming. Importantly, unlike in cryptographic libraries – where the attacker's goal can be clearly identified (i.e., stealing the secret keys), privacy-preserving applications generally do not reveal where leakage may occur, making binary analysis much more challenging.

Second, single-stepping could be restricted in future architectures. While current tools can use performance counters to detect if a single-step was successful [60], AMD claims to prevent the hypervisor from reading performance counters for guest events starting with Zen 5 [35]. Moreover, Intel TDX mitigates single-stepping by ensuring sufficient VM progress between context switches [65]. Therefore, although supporting single-stepping is not a limitation of our framework, it was not included in our evaluation.

Key Insight 1: Filtered Controlled-Channel. We choose nested page faults for SNPeek, which, besides being much faster, represent a design choice that cannot be mitigated without major architectural changes. As shown in Figure 3, the untrusted hypervisor clears the present bit of all the VM pages at the beginning of the execution. When the guest VM triggers page faults, the error code reveals whether the faulting page is used for instruction or data and whether it is encrypted. If the page is not encrypted, the hypervisor considers it uninteresting and retains its mapping in subsequent executions. Similarly, if a faulted page address belongs to the reserved memory of the guest system, it is likely associated with kernel activity and can be skipped. To ensure accurate control and data flow tracking, SNPeek maps the newly accessed page and conditionally unmaps the previous one at each page fault.

Platform-specific optimizations. To further refine page-level analysis, SNPeek also supports additional platform-specific optimizations. For AMD EPYC CPUs, we leverage performance counters that allow the hypervisor to monitor guest events in either user-space or OS-space. Our implementation uses two of these: one tracking retired instructions from guest user-space and another tracking retired micro-operations (uops) from the guest OS. As a result of these optimizations, pages containing only kernel code are labeled irrelevant and excluded from analysis. This optimization is optional to improve runtime on AMD EPYC, and is not necessary for page-level analysis. As a result, SNPeek also generalizes to other architectures [36]. We analyze the collection speed in Table V (Appendix A).

2) Spatial Resolution: We use SNPeek to collect runtime side channels of the target program at granularity levels ranging from 4 kB to 16 B. These include both generalizable platform-independent ones (controlled-channel and cache), and platform-dependent ones (ciphertext and PMCs). Figure 10 in Appendix A shows an example trace with collected side channels.

Platform-independent: Controlled-channel. We monitor access patterns of the victim at page granularity with controlled-channel techniques, distinguishing between code fetches and data accesses. SNPeek follows a principle of mapping only one *interesting* code page at a time, unmapping the current code page whenever the guest jumps to a new one[2]. For data accesses, SNPeek manages a queue for mapping data pages. The queue size is adjusted dynamically to avoid deadlocks when a single instruction accesses more pages than the queue size. This ensures precise control over memory access patterns, without losing track of any accessed pages.

Platform-independent: Cache attacks. While controlled-channel techniques have a 4 kB page granularity, access patterns with a 64 B granularity are possible via cache attacks. When handling an NPF, the hypervisor iterates the nested page table and maps the faulted page before returning control to the VM. As an attacker cannot predict which 64 B blocks of a page the victim accesses, we mount a Multi-Prime+Probe on all 64 cache sets before the context switch. At the next NPF, the hypervisor probes all cache sets to identify the accessed 64 B blocks.

Developing a precise Multi-Prime+Probe attack poses significant challenges. On newer AMD CPUs, such as EPYC, the shared L3 last-level cache is non-inclusive. Therefore, performing L3 Prime+Probe on each cache set requires accessing at least 24 addresses, accounting for both L2 and L3 cache set entries [66]. Although the untrusted hypervisor shares an internal L2 cache with the target VM, the timing difference between L2 hits and misses is only about 40 cycles. This small difference introduces significant noise when attempting to probe across 64 different cache sets, as we will see below.

Key Insight 2: Noise reduction via MSRs. We introduce a novel approach to improve L2 Prime+Probe by exploiting model-specific-registers (MSRs) to reserve L3 cache usage. The MSRs `0xC001_1095` and `0xC001_1096` define a memory range for which the number of L3 ways can be configured via the MSR `0xC001_109A`. We allocate our buffer for eviction set at a high memory address beyond 30 GB, and use the MSRs to reserve all L3 ways for memory addresses below 30 GB. Consequently, the addresses from our eviction set cannot be cached in the last-level cache, as there is no available way. Hence, if an address in the eviction set is evicted from the L2 cache, it is directly evicted to the main memory, increasing the measured timing difference for Multi-Prime+Probe to more than 700 cycles. This approach improves the speed of the prime stage from 24 distinct addresses accessed to only 8 (the capacity of an L2 cache set), without introducing noticeable performance overhead, as the memory space above 30 GB is rarely used.

[2]Corner cases, such as a single instruction spanning a page boundary, can be handled by detecting a threshold of two repeated page faults without execution progress, using PMCs.

While these specific MSRs were introduced in the Zen3 microarchitecture, similar cache reservation technologies (e.g., Intel's Cache Allocation Technology, or CAT) can achieve a comparable effect on older AMD CPUs and Intel CPUs. We demonstrate a proof-of-concept improving Prime+Probe using Intel CAT in Appendix C.

Platform-specific: Ciphertext. Similar to cache attacks, before the guest writes to an unmapped page, the hypervisor reads all 256 ciphertext blocks of the page [21]. At the next NPF, SNPeek compares all blocks to their prior values to pinpoint the ones modified by the victim, and highlight ciphertext differences. This exploit provides insights into the victim code at an even deeper spatial side-channel level.

Platform-specific: PMC leakages. In addition to the two events we introduce for optimizing temporal resolution on AMD EPYC, SNPeek uses three other events used by Gast et al. [19], *Retired Branch Instructions*, *Retired Taken Branch Instructions*, *Retired Near Returns*, that leak the control flow of the victim. The former is updated at each NPF, and the latter during instruction fetch NPFs. This method leverages performance counters to provide detailed insights into the execution flow, further enriching the side-channel data available for analysis.

3) Targeted Trace Collection: In addition to tracking across the entire target call flow, SNPeek enables precise leakage analysis by offering configurable controls over the monitoring phase. This allows developers to instrument applications with code pages containing specific assembly instructions, provided these instructions can be tracked by performance counter events. In the offline phase, the developer wraps the target code snippet with two code pages that execute the `clflush` instruction multiple times, enabling the framework to treat these pages as signals to start or stop tracking. As we show in Section VII, in real-world exploits, this is equivalent to an attacker who would monitor specific I/O, network traffic, and access patterns in the online phase.

B. Feature Extraction

Given a set of execution traces, SNPeek extracts features to model privacy attacks as discriminative machine learning problems, which aim to separate traces depending on the value of the target. The types and complexity of features chosen often result in a trade-off between interpretability and utility. Simpler features tend to be more suitable for pinpointing the source of the leakage, while high-level features are capable of producing a tighter empirical lower bound of the side channel. To support the automated audit process and balance this trade-off, SNPeek employs both handcrafted features and automatic feature learning.

1) Handcrafted features: We engineer five sets of handcrafted features as summarized in Table II. These features do not exhaustively capture all the distinguishing patterns that can be extracted from traces, but, as we show in Sections V and VI, highlight the utility of the framework in pinpointing sources of leakage. Feature set $\mathcal{F}_1$ focuses on a page-level granularity and counts the number of total and distinct pages observed for the controlled channel, across a particular trace, while $\mathcal{F}_2$ operates at block- and cache-level granularity. $\mathcal{F}_3$ captures a lower level of spatial granularity by computing histograms for the number of times individual cache lines and page blocks are accessed in cache and ciphertext side channels, respectively. $\mathcal{F}_4$ looks at the side channel for each individual page level, computing how many data accesses, cache lines, and blocks are being accessed, and providing statistics for these across a trace. $\mathcal{F}_5$ captures aspects of the control and data flow by counting how many times individual pages are accessed during the execution.

2) Automatic feature learning: Alternatively, we rely on representation learning, allowing analysts to train deep learning models for automated analysis of the leakage. This involves feeding the trace information to the models and relying on their representational power to expose discriminative features from the sequences encoded in the traces. To aid learning, we pre-process the traces by abstracting the memory space and ciphertext information. More precisely, we extract only the distinct page-table accesses (code and memory pages) and the ciphertext changes observed through the ciphertext visibility channel. This transformation is described in Appendix A.

C. Leakage Analysis

To conduct a leakage discovery task, one can build different analysis models using the framework. We implement several analytics and machine learning tools for evaluating features against datasets and identifying leakage. Collected traces first need to be separated into different classes to define a classification problem. The labeling depends on the threat model. A distinguishing attack (Section III-B) can be modeled using two classes, indicating whether the targeted entity is present in the input set for a particular trace. In contrast, fingerprinting attacks (Section III-C) can be modeled as two consecutive distinguishing problems: a two-class setting reflecting whether the targeted entity is part of a set of labels of interest, followed by a multi-class setting that identifies which of the labels the entity corresponds to.

After collecting and labeling the traces, the frameworks can discover the source and severity of the leakage across different labels. This is achieved by choosing the feature sets, localizing the portion of the program suspected of leakage, and computing features over the collected traces. After feature extraction, the leakage-analysis tools are applied. To identify leakage sources, we provide statistical tests and visualization tools to verify whether the distributions of features across labels are distinct. For quantifying the leakage, we implement supervised learning models that allow measuring leakage in the online phase. Our implementation relies on scikit-learn [67] for classifiers based on handcrafted features, and on TensorFlow [68] for feature learning through sequence models.

Evaluation To measure the leakage and highlight the empirical advantage through SNPeek, in Sections V and VI we use an L2-regularized logistic regression classifier trained for 1000 iterations with L-BFGS. For feature learning, we implement

Name	Feature Set	Description	Count
CF Count	$\mathcal{F}_1$	Number of total & unique code pages fetched	2
DA Count	$\mathcal{F}_1$	Number of total & unique data pages accessed	2
Cache Count	$\mathcal{F}_2$	Number of total & unique intercepted cache lines in cache attacks	2
CI Count	$\mathcal{F}_2$	Number of total & unique modified ciphertext blocks in memory	2
Cache Frequency	$\mathcal{F}_3$	Number of times each of the 64 4 kB cache lines was accessed	64
CI Frequency	$\mathcal{F}_3$	Number of times each of the 256 blocks of any page was modified	256
DA Stats	$\mathcal{F}_4$	Stats over the number of data page accesses following a code page fetch	11+N
Cache Stats	$\mathcal{F}_4$	Stats over the number of total & unique cache lines accessed for a page	2*(11+N)
CI Stats	$\mathcal{F}_4$	Stats over the number of total & unique blocks accessed for a page	2*(11+N)
CF Page Frequency	$\mathcal{F}_5$	Frequency of code fetches for individual code pages	M_{CF}
DA Page Frequency	$\mathcal{F}_5$	Frequency of page accesses for individual data pages	M_{DA}

TABLE II: Summary of handcrafted features in SNPeek. $\mathcal{F}_4$ Stats correspond to features describing the distribution: min, max, first to ninth quantiles, and a histogram with N bins. $\mathcal{F}_5$ frequencies are over the M first seen pages in a trace.

a bidirectional LSTM with attention (see Appendix A). The model has 751 554 parameters and is trained with Adam using a learning rate of 2e-5. We compute the empirical advantage by training and validating on 80% of the samples and testing on the remaining 20%. For the logistic regression, we report the average over 5 trials.

V. Oak Private Information Retrieval

In this section, we evaluate SNPeek on *Oak* [23], [69] private information retrieval (PIR). Project Oak is a software platform developed by Google for constructing distributed systems with built-in transparency and guarantees of confidentiality and integrity. It provides core components for developing enclave applications and supports remote attestation. Oak includes an untrusted launcher on the host and uses a Wasm runtime to execute Wasm enclave applications within a CVM. The launcher handles requests using gRPC, providing end-to-end encryption for data in transit. This architecture supports PIR for in-memory key-value lookups, enabling sensitive data queries while preserving the confidentiality of data and queries.

PIR generally allows a client to retrieve an element from a (typically public) database, without revealing the accessed element to the server that hosts the database. Cryptographic solutions to PIR are well-established [70], but even the most efficient constructions fundamentally require the server to scan the entire database to answer a query, which limits the scalability. To overcome these issues, many solutions [23], [24], [71] instead use a TEE with the goal of protecting the queried index from the server.

A. Distinguishing Attack on Oak PIR

As a motivating example, we distinguish the request of a missing key and a key with a value, capturing the resulting traces on the hypervisor. Figure 4 shows sequences of the number of data page accesses at each code page. The traces are clearly distinguishable, and we omit a more detailed analysis using machine learning.

For a more realistic scenario, we expand the dataset to include 1,000 key pairs. The keys are the strings `key0` through `key999`, resulting in lengths of 4 to 6 bytes. Each

Fig. 4: Traces of the Wasm runtime for the oak `key_lookup` module with a missing and a test key [72].

corresponding value is randomly generated with a size of up to 1,000 bytes. Although the sequences shown in Figure 4 remain the same across these keys, we can still distinguish individual lookups by observing variations in page and cache accesses at specific code pages, as shown in Table III. The lookup module must load the key-value pairs from different pages containing distinct cache lines for the second to fifth data page accesses from this code page. The number of cache-line accesses also reflects the size of the values, which can be up to 1,000 bytes and span multiple cache lines. Note that we use manual examination to showcase and validate this leakage. We collected the traces using Targeted Trace Collection (Section IV-A3). Fully automating the end-to-end exploit would require an attacker to recognize a pattern targeting the vulnerable code and train a new model offline.

Following Definition 1, the goal for the attacker is to distinguish two sequences of memory accesses into a database of 1000 elements: one consisting of 10 identical PIR retrievals of the first element, and the other consisting of 9 identical retrievals of the first element, followed by a single retrieval of the last element. We collect 1500 traces for each case and use them to evaluate the effectiveness of using SNPeek to evaluate PIR mitigations.

B. Evaluating Mitigations

To mitigate the above leakage, we can apply a linear scan, i.e., a `std::vector` accessed through a scan, using a constant-time compare-and-swap, and the PathORAM [73] used by Signal [24]. We run both applications inside a CVM. We collect traces with SNPeek and evaluate the collected traces using the methods described in Section IV.

	key0	key1	key60	key61	key998	key999
1st	123dce *31*	123dce *31*	123dce *31*	123dce *31*	123dce *31*	123dce *31*
2nd	106835	106aea	106ae9	106ae9	106835	106835
3rd	106b9e	123e6e	123e6e	123e6e	106b9d	106b9d
4th	123e6e	10696c	10696b	10696b	123e6e	123e6e
5th	**139213** ***48,49,63***	**142816** ***7,16-31***	**13a018** ***32,50-52***	**139010** ***58,60,63***	**136617** **0-6,22,62**	**136a1a** **6,32**
6th	123dce *31*	123dce *31*	123dce *31*	123dce *31*	123dce *31*	123dce *31*

TABLE III: An example of distinct page- and *cache-access* sequences appears in one of the code pages (index 1,642) within the oak `key_lookup` trace for six different key lookups. The page number refers to the guest's physical page number (gPN), followed by cache-line accesses within this page. We repeat the lookup on each key five times.

Distinguishing attack via handcrafted features. First, we analyze the leakage exposed via handcrafted feature sets summarized in Table II. We train a logistic regression model on each of the feature sets, as well as their union. The results are summarized in Table IV. While none of these features reveals a significant advantage against the constant-time linear scan implementation, feature sets $\mathcal{F}_5$ reveal leakage in Signal's ORAM. The leakage originates from a ciphertext side-channel related to how zero-value items are handled. When an item with a zero value is retrieved, it is added back to memory, leaving the ciphertext unchanged. In contrast, retrieving a non-zero value causes the plaintext to change, which results in a changed ciphertext. This is consistent with the observation that retrieving different non-zero values produces no discernible difference in the trace.

Distinguishing attack via feature learning. To explore what information can be observed without any feature engineering, we also train an LSTM model on the pre-processed traces. The advantages obtained through the LSTM on the test dataset are shown in Table IV. Using only page-level information, the LSTM cannot get any significant advantage against the two mitigations. However, once we add the ciphertext block-level visibility side channel, we observe that both linear scan and ORAM are distinguishable. This finding aligns with the handcrafted features-based analysis, reinforcing the vulnerability in these implementations: side-channel visibility into the exact ciphertext changes reveals more than just the number of changes. Nevertheless, the larger attacker advantage obtained through automatic feature learning over handcrafted features highlights the complementary role of the two in SNPeek: while handcrafted features are useful for pinpointing the source of the leakage, feature learning provides tighter empirical advantage estimates.

Remark. While our experiments highlight leakage through the ciphertext channel, this is due to our experiments using AMD SEV-SNP, which is known to be vulnerable. In contrast, Signal's ORAM was implemented with Intel SGX as its target architecture, which does not suffer from this side channel in the same way. We therefore cannot confirm any vulnerability in Signal's deployment of ORAM.

	Logistic Regression						LSTM	
	$\cup$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$	$\mathcal{F}_4$	$\mathcal{F}_5$	Page	Page+Block
Linear Scan	**0.02**	0.00	0.00	0.01	0.01	**0.02**	0.03	**0.80**
Signal ORAM	**0.19**	0.03	0.04	**0.18**	0.02	0.07	0.03	**0.32**

TABLE IV: Normalized advantage (Definition 1) obtained through a Logistic Regression on all ($\cup$) and sets of ($\mathcal{F}$) handcrafted features, and an LSTM on sequence-based features at page- and block-level, across PIR implementations.

VI. Private Heavy Hitters in TF-Federated

The private heavy hitters (PHH) problem has received immense research attention, with cryptographic solutions deployed under multiple threat models. PHH aims to compute a histogram of the user data, while providing a (differential) privacy guarantee to individual users, with n users $1, \ldots, n$ each holding one datapoint from some large domain. Recent deployments by Google [8] and Meta [9] leverage AMD SEV-SNP and Intel SGX for this task, respectively.

In this section, we evaluate SNPeek on the TensorFlow-Federated [27] implementation of Private Heavy Hitters in Confidential VMs by Google, as deployed in Gboard via AMD SEV-SNP [8]. The specific application is out-of-vocabulary word discovery: "discovering new common words to incorporate them into the typing model, without revealing any uncommon private words." The work of Srinivas et al. [9] describes a deployment of the same algorithm based on SGX, but does not discuss mitigations to architectural side channels nor make any code available.

Algorithm for DP Heavy Hitters. TEE-based solutions for PHH [7], [8], [9] apply a textbook DP algorithm inside a TEE [74] (the so-called stability-based histograms or noise-and-threshold), and rely on the TEE to safeguard inputs and keep sampled DP noise confidential. In particular, the TensorFlow-Federated implementation [27], [8] closely follows the textbook DP mechanism for large domain histograms. Figure 5 shows a basic C++ reference implementation, for illustrative purposes.

Case Study: Counting Common URLs. A typical application of PHH is in browser telemetry [75], [76], where clients report URLs that crashed their browser and the related context (see Network Error Logging [77]). In the rest of this section, for illustrative purposes, we consider a DP algorithm that attempts to identify frequent URLs submitted by devices. The goal of the attacker is to extract additional information (beyond the result histogram) about the URLs submitted by individual devices, as formalized in Definitions 1 and 2. This attack model also incorporates Sybil attacks (Section III).

```
// batch, epsilon, and threshold in the context
std::unordered_map<string, int> hist;
std::vector<std::pair<string, int>> result;
...
while(!batch.empty()){ // Aggregate inputs
  hist[batch.front()]++;
  batch.pop();
}
for (auto [k, v]: hist) { // Noise and Threshold
  auto noisy_val = v + sample_centered_laplace(epsilon);
  if (noisy_val >= threshold)
    result.push_back(std::make_pair(k, noisy_val));
}
```

Fig. 5: Baseline unprotected PHH application example code. "batch" refers to user data to be protected. The variance of the noise and the threshold are set according to ϵ and δ, to achieve (ϵ, δ)-DP.

A. TF-Federated Evaluation

We analyze the leakage of the aggregation and noise-and-threshold phases of the DP algorithm[3]. As in the snippet in Figure 5, the aggregation phase accumulates inputs into a hash map and places the keys into a vector. The noise-and-threshold phase iterates through the vector, adding DP noise to each histogram entry, and then thresholding.

Findings. In both parts of the code (aggregation and noise-and-threshold), the code leaks sufficient information to enable a distinguishing attack (Definition 1). Figures 6a and 6b show page-level leakage in the application. We plot the count of data page accesses after each code page fetch (feature set $\mathcal{F}_4$) in two neighboring executions: in the first one, we ingest "normal.com" 10 times, while in the second one, we ingest into the application "normal.com" 9 times, followed by "embarrassing.com". This corresponds to an instance of a distinguishing attack from Definition 1. The plots show that the execution length is data-dependent for both the aggregation and noise-and-threshold phases. We examine this further, along with a potential mitigation, in the following section.

B. Advanced Attacks and Mitigations

In the rest of this section, we discuss different flavors of attacks and mitigations, and how their effectiveness can be evaluated with SNPeek. Instead of working with the TF-Federated codebase from the previous section, the results in this section are with respect to a smaller example (partially reported in Figure 5) and included with our library. This simpler example is less leaky than a larger codebase, and allows us to better identify the origin of the leakage when using automatic ML approaches to exploit it. Moreover, any attack found in the simpler codebase translates to the more complex TF-Federated implementation, and mitigations are easier to implement and evaluate.

We start by discussing an attack based on data-dependent execution. This attack is analogous to the one reported in Figure 6b, and mentioned above.

[3]Source code available at: google-parfait/tensorflow-federated, file: dp_open_domain_histogram_test.cc, line 655, commit e245ed4

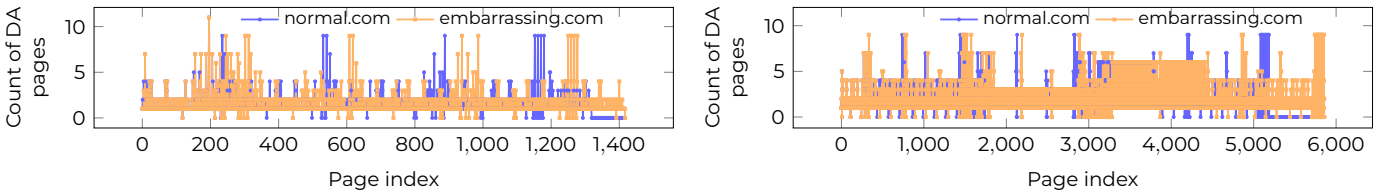
Distinguishing attack via simple features. Assume the target device has either URL0 or URL1. The attacker injects 99 Sybils with URL0 before the target, so the input is either 100 copies of URL0 or 99 copies of URL0 and one URL1. The attacker obtains 1500 traces for each case and uses them to learn how to distinguish URL0 and URL1. One leakage source in Figure 5 is the noise-and-threshold loop iteration count, which depends on the number of keys in the input set.

If the target visited URL0, there is just one key in the map, while a URL1 visit results in 2 keys and therefore one more iteration. Figure 7 highlights how we capture this leakage: The distributions of code fetches and data accesses in the noise-and-threshold phase are different across the two labels. This leakage can also be confirmed through a logistic regression classifier trained on the $\mathcal{F}_1$ feature set with perfect accuracy, giving the attacker a normalized advantage of 1.0 (Definition 1).

Mitigation. While padding the unordered map to the maximum number of possible keys is an obvious mitigation, it is extremely inefficient. Thus, we propose a DP-based mitigation. Observe that the noise-and-threshold loop iteration count—the quantity used by the previous attacker—has limited sensitivity to the target's value: switching from URL0 to URL1 changes it by only 1. If there is just enough variance in the number of iterations, the attacker will have a hard time distinguishing the two cases. We accomplish this by injecting a random number of distinct "dummy" elements. More precisely, we add a number of dummies distributed as a discrete shifted Laplace random variable with parameters ϵ, δ, to ensure that the number of loop iterations is (ϵ, δ)-DP and the corresponding leakage is bounded. Figure 9 in Appendix A shows model code. The idea of DP-fying side-channel leakage by adding dummy contributions appears in related work [78], [79]. We show that SNPeek can help determine appropriate values for DP parameters.

We evaluate this mitigation by computing the empirical advantage of the attacker over the noise-and-threshold stage, across a range of ϵ values for the dummies, and comparing it with the analytical lower bound computed in Appendix A (Figure 8, top). The results show that the mitigation succeeds for the noise-and-threshold phase, dropping the attacker advantage below and bringing it close to the analytical lower bound for sufficiently small ϵ, across all feature sets described in Table II. This highlights that SNPeek can evaluate defenses and also discover parameters for effective mitigations. Regarding performance, the expected number of dummies for $\delta = 10^{-9}, \epsilon = 0.1$ is about 200, offering a very good tradeoff between privacy and performance for large enough deployments, e.g., with $n \geq 10000$.

Distinguishing attack via advanced features. Despite the above mitigation, there could be leakage in the aggregation stage. In Figure 8 (bottom), we evaluate the mitigation against attacks that use *all* feature sets over the aggregation subroutine. The input-dependent memory usage—code fetches and data accesses per page—expressed through $\mathcal{F}_5$ suffices for a logistic



(a) Last iteration of the accumulation phase

(b) The noise-and-threshold phase

Fig. 6: Page-level leakage of TensorFlow Federated PHH code during accumulation and noise-and-threshold phases. We ingest "normal.com" 9 times, then either "normal.com" or "embarrassing.com". Adding a new key triggers a longer code path in accumulation and an extra iteration in the noise-and-threshold phase.

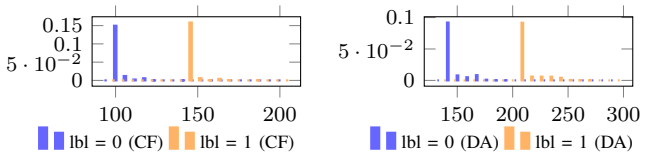


Fig. 7: The separable distribution of the CF and DA Count features across labels in the noise-and-threshold phase of the vanilla PHH implementation.

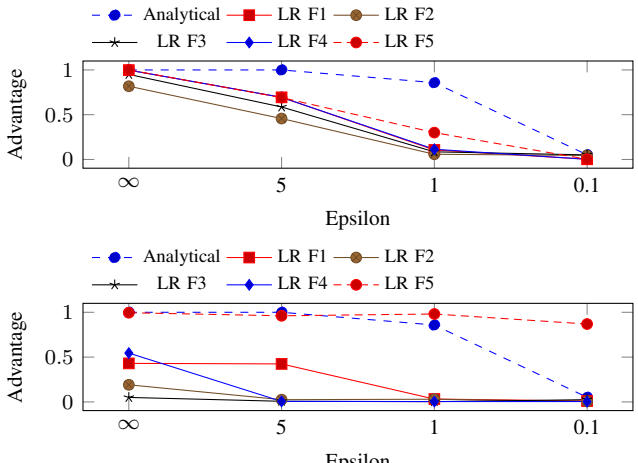


Fig. 8: Advantage of the $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$, $\mathcal{F}_4$ and $\mathcal{F}_5$ Distinguishing attacker for PHH protected by dummy operations for various ϵ. The Empirical advantage is computed using a logistic regression for the noise-and-threshold (top) and Aggregate (bottom) stages, averaged over 5 trials. We compute the analytical upper bound using the formula in Appendix A.

regression classifier to bypass the mitigation. The attacker bypasses the analytical upper bound because it is computed under the assumption that such features were hidden. Mitigations for this attack would need to randomize a histogram of accesses, rather than just one count. We leave this for future work.

Fingerprinting attack. Recall that fingerprinting attacks are applicable when the attacker knows the distribution W of a target's data over a (possibly large) domain and is interested in a (possibly small) set I. In our example, we instantiate W with a power-law distribution with parameter 0.5 over the list L of the top 1000 most common sites in Alexa Top 1 Million Sites dataset [80]. The choice of 0.5 is arbitrary but realistic, in that it corresponds to a skewed distribution. The list L contains URLs such as "google.com" and "wikipedia.org", but also URLs that might leak additional information, such as "google.co.jp". I are the 301 URLs in L not ending in ".org", ".com" or ".net", and thus often carrying information about the user's language or location.

The offline phase of the attack trains two classifiers: the membership *binary* classifier to identify whether or not the target's value is in I and the fingerprint classifier to predict which of the 301 values in I the target had, assuming it had one. The offline phase has the following steps

1) Create a dataset as follows:
   a) Sample target's data x from W.
   b) Create one instance of each element in I.
   c) To boost fingerprinting success, create enough Sybil data that is out-of-domain (e.g., not URLs) such that it is unlikely that the memory locations maintaining counts of elements in I will be close to each other. This permits control over the granularity of side channels needed to effectively fingerprint I.
   d) To improve membership detection in I, create enough Sybils such that a rehash event is guaranteed when $x \notin I$. This can easily be done by inspecting the hashtable code, which is available to the attacker.
2) Capture side-channel traces produced by running PHH algorithm on the above input.
3) Repeat above to create sufficiently many unlabeled examples (traces).
4) Train the membership classifier on *all examples*, each labeled by the bit indicating if $x \in I$.
5) Train the fingerprint classifier on *only examples where* $x \in I$, each labeled by x.

In the online phase, the attacker runs the membership classifier to determine whether or not the target has interesting data ($\in I$). If the target is deemed interesting, they will run the fingerprint classifier.

We evaluate the fingerprinting attack by building a dataset of 5000 instances, corresponding to traces of samples collected from W. We instantiate the membership and fingerprinting classifiers as logistic regression models based on all $\mathcal{F}_1$-$\mathcal{F}_5$ features. The membership classifier obtains a *normalized*

advantage of 1.0 (Definition 1), which is a perfect score. This underscores the power of Sybils to exploit the vulnerability in the `std:unordered_map` implementation. The fingerprinting classifier yields a *normalized* advantage of 0.44 (Definition 2), highlighting attack feasibility due to substantial leakage of the victim URL label through the PHH workflow.

VII. User-Defined Functions in Privacy Sandbox

TEE-isolated user-defined functions aim to enable new privacy-preserving applications such as data source verification [81], outsourcing computation [82], private function-as-a-service [83], and ads targeting [29]. UDFs run atop a language sandbox (e.g., JavaScript/Wasm) to enable third-party queries on user data. The language sandbox restricts third-party code from extracting user data, limits interfaces, and enforces constraints like time limits and accounting [82]. We focus on the side-channel evaluation of an example UDF that is implemented by the Protected Auction Key/Value service, part of the Privacy Sandbox [30]. Privacy Sandbox, as an alternative to third-party cookies, enables third-party advertisers (AdTechs) to access advertising signals stored in an in-memory TEE-protected key/value database. They use UDFs to run custom queries without direct access or logging. The query's output is aggregated and protected by DP techniques.

However, the requirement to keep the attacker out of the hypervisor requires special care for secure on-premise deployments of such privacy-preserving systems. Consider a hypothetical scenario where a malicious AdTech tries to run their own deployment of Privacy Sandbox on machines they fully control, including the hypervisor. Although the TEE would properly attest their key-value service deployment with the isolated UDF, they can still exploit side-channels to extract raw data from the key-value service. Hence, specialized mitigations intended to prevent or detect side-channel signaling behavior by a UDF would be advisable.

A. Stealing User Data via Covert Channel

We evaluate a covert channel attack where a malicious UDF steals user query arguments. Because the UDF is maliciously constructed, the attacker does not need to train a model to learn correlations. The UDF can use a deterministic encoding that gives a clear view of the data. As a result of this noiseless recovery, the normalized advantage is 1.0.

Profiling UDF runtime. We analyze the UDF runtime to identify a trigger point for our covert-channel attack—when the receiver expects to see data from the sender. The UDF in the Protected Auction Key/Value service [30] is based on the V8 engine [84]. It supports JavaScript or inline Wasm, where the Wasm code must be invoked by JavaScript driver code, ensuring the UDF entry point remains in JavaScript. When the V8 engine creates a typed array, it first executes one code page, writes to this page, and then executes this page again. This $X{+}W{+}X$ access pattern serves as an indicator of the start and end of the UDF execution. We mark the inline Wasm by surrounding it with two typed arrays, enabling SNPeek to only track the Wasm execution.

Encoding data over ciphertext. In the inline Wasm, we choose the ciphertext side channel to encode secret data. Specifically, we create a 4 kB memory buffer that contains 256 ciphertext blocks. As we iterate over the data stream byte by byte, we write to one of the 16-byte blocks in the 4 kB buffer. The index of the block depends on the value of each secret byte. We observe that in each iteration, the V8 engine consistently accesses four distinct memory pages in addition to our 4 kB encoding buffer. Among these, two page faults occur with a single specific block modification, which we attribute to updates in the loop counter and a temporary variable. The other two page faults are caused by memory reads. Therefore, we check the ciphertext changes of the previous faulted page every time a new memory page fault occurs. Thus, we can encode one byte with only five page faults, i.e., five context switches.

Evaluation. We evaluate the performance of our covert channel by transmitting 48 bytes of user-supplied input arguments 100 times. We ignore three page faults when transferring each byte, as only one additional page fault is enough to trigger checking the ciphertext changes. Since these faulted pages have distinct page numbers, SNPeek can simply skip unmapping them. Our covert channel achieves an average transmission rate of 497 kbit/s with an error rate of 0. The speed of this attack can be further increased by combining page numbers and ciphertext to encode secrets more efficiently. For example, using 256 pages, an attacker can encode an extra byte per access through a controlled channel. We verify this with a page fault-based covert channel spanning 256 pages. To apply it, the attacker only needs a profiling step to map each page to its corresponding byte value, since the guest operating system controls the gPN, resulting in a non-contiguous mapping. Given that the memory limit of the V8 engine is 4 GB [29], this optimization meets such constraints.

By enabling this practical exploit, SNPeek reveals that simply placing the UDF in a language sandbox and restricting access to logging interfaces does not prevent the covert transfer of sensitive data across isolation boundaries.

VIII. Related Work

SGX-STEP [59] is a framework for rapid prototyping of side-channel attacks in SGX [85], [65]. Similarly, Stacco [86] offers a framework for collecting and differentially analyzing multiple side-channel traces (e.g., page faults, cache attacks) against SGX enclaves. While Stacco focuses on using differential analysis to automatically detect vulnerabilities in cryptographic implementations such as SSL/TLS, our work aims to quantify information leakage in data-privacy applications. Within this evolving threat model, attackers use system interfaces to construct new side channels [1], [17] and improve the reliability and bandwidth of side channels [61], [59], [87]. Unlike traditional side channels, these attacks can completely circumvent system noise. We provide a more detailed discussion in Appendix D.

SEV-SNP relies on encryption for hiding memory, but does not protect the ciphertext, which enables the new class of

ciphertext side channels [21], [22]. Attackers can also exploit privileged interfaces such as performance counters [19] and power reporting [88] to leak side-channel information from CVMs. TDXDown [37] exploits gaps in system-level countermeasures against timer interrupt attacks. SEV-STEP [60] is a framework for prototyping single-stepping and L1 cache attacks on SEV-SNP.

Software-based side-channel attacks have impacted TEEs in real products to steal cryptographic keys [89], [90], [86]. The industry consensus to mitigate these attacks is to apply constant-time coding practices [91]. Previous work has proposed automated tools to test such implementations [92], [93], [94], [95]. However, these tools and constant-time coding practices are not applicable and practical for general-purpose programs. Yuan et al. applied manifold learning to evaluate side-channel attacks on media software [12]. Ciphertext side channels have been demonstrated as an effective technique to steal ML models' inputs and hyperparameters [96], [97]. Further, side-channel-assisted information retrieval has been demonstrated against SQLite [13]. We focus on automated side-channel testing of privacy-preserving applications in CVMs.

Haeberlen et al. argue that differentially private query release may be vulnerable to covert channel attacks via side-channel leakage [42]. Their threat model explicitly separates the service provider from the adversary and leaves only the privacy budget and query time as side-channels. Jin et al. demonstrate that the running time of noise sampling algorithms could be used to circumvent DP guarantees [98]. Ratliff & Vadhan formalize DP against adversaries observing that side-channel and propose padding-based methods for achieving that objective [43].

IX. Conclusion

We conclude that automated analysis of side-channel leaks is crucial to improve the privacy guarantees of applications running within CVMs. The status quo of relying solely on software techniques to mitigate side-channel attacks is impractical, and developers of privacy-preserving applications need to constantly evaluate an app's threat model and execution traces to ensure sufficient mitigation. Toward this goal, a comprehensive framework like SNPeek can significantly help developers assess their threat model and mitigation strategy. In the future, defense-in-depth mitigations such as reducing side-channel information at the architecture level, preventing Sybil attacks, and carefully applying data-oblivious data structures like ORAM are promising but require further investigation.

X. Ethics Considerations

This work builds upon the observation that Confidential Virtual Machines (CVMs) are not invulnerable to side-channel attacks. Although hardware vendors generally consider these attacks outside their threat models, practitioners deploying privacy-preserving solutions using CVMs are responsible for mitigating such attacks. Our proposed tool, SNPeek, is designed to assist security researchers in identifying and mitigating side channels rather than serving as a tool for malicious exploitation. Although it could theoretically support attackers, our intent is to facilitate better defensive measures by quantifying risks and evaluating countermeasures.

We performed all experiments on local systems, did not use personal data or involve human subjects, and thus did not encounter any additional ethical concerns regarding privacy or user consent. By openly reporting vulnerabilities to the affected projects and contributing a framework for enhanced side-channel analysis, we aim to improve the overall security posture of CVM-based privacy solutions.

Acknowledgements

We would like to thank Kobbi Nissim, Jonathan Katz, Sarah Meiklejohn, Marco Gruteser, Peter Kairouz, Daniel Ramage and Shabsi Walfish for their constructive feedback and support.

References

[1] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015.

[2] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015.

[3] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Springer, 2017.

[4] D. F. Aranha, F. R. Novaes, A. Takahashi, M. Tibouchi, and Y. Yarom, "Ladderleak: Breaking ecdsa with less than one bit of nonce leakage," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[5] Y. Lyu and P. Mishra, "A survey of side-channel attacks on caches and countermeasures," *Journal of Hardware and Systems Security*, 2018.

[6] R. Li, Q. Wang, Q. Wang, D. Galindo, and M. Ryan, "Sok: Tee-assisted confidential smart contract," *arXiv preprint arXiv:2203.08548*, 2022.

[7] H. Eichner, D. Ramage, K. Bonawitz, D. Huba, T. Santoro, B. McLarnon, T. Van Overveldt, N. Fallen, P. Kairouz, A. Cheu *et al.*, "Confidential federated computations," *arXiv preprint arXiv:2404.10764*, 2024.

[8] D. Ramage and T. Van Overveldt, "Discovering new words with confidential federated analytics," https://research.google/blog/discovering-new-words-with-confidential-federated-analytics/, accessed: 2025-05-28.

[9] H. Srinivas, G. Cormode, M. Honarkhah, S. Lurye, J. Hehir, L. He, G. Hong, A. Magdy, D. Huba, K. Wang *et al.*, "Federated analytics in practice: Engineering for privacy, scalability and practicality," *arXiv preprint arXiv:2412.02340*, 2024.

[10] Google, "Confidential federated compute," https://github.com/google-parfait/confidential-federated-compute.

[11] privacysandbox.com, "Protecting your privacy online," https://privacysandbox.com/intl/en_us/.

[12] Y. Yuan, Q. Pang, and S. Wang, "Automated side channel analysis of media software with manifold learning," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[13] A. Shahverdi, M. Shirinov, and D. Dachman-Soled, "Database reconstruction from noisy volumes: A cache {Side-Channel} attack on {SQLite}," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[14] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[15] H. Wang, S. M. Hafiz, K. Patwari, C.-N. Chuah, Z. Shafiq, and H. Homayoun, "Stealthy inference attack on dnn via cache-based side-channel attacks," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022.

[16] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page {Table-Based} attacks on enclaved execution," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[17] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[18] C. Percival, "Cache missing for fun and profit," 2005.

[19] S. Gast, H. Weissteiner, R. L. Schröder, and D. Gruss, "Counterseveillance: Performance-counter attacks on amd sev-snp," in *Network and Distributed System Security Symposium 2025: NDSS 2025*, 2025.

[20] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2008.

[21] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "{CIPHERLEAKS}: Breaking constant-time cryptography on amd sev via the ciphertext side channel," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[22] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang, "A systematic look at ciphertext side channels on amd sev-snp," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.

[23] google.com, "Project oak," https://github.com/project-oak/oak.

[24] G. Connell, "Technology deep dive: Building a faster oram layer for enclaves," https://signal.org/blog/building-faster-oram/, 2022.

[25] E. Stefanov, M. van Dijk, E. Shi, T. H. Chan, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," *J. ACM*, 2018.

[26] google.com, "Tensorflow federated," https://github.com/google-parfait/tensorflow-federated.

[27] ——, "Tensorflow Federated – DP Open Domain Histogram," https://github.com/google-parfait/tensorflow-federated/blob/main/tensorflow_federated/cc/core/impl/aggregation/core/dp_open_domain_histogram.cc.

[28] H. Aksu, B. Ghazi, P. Kamath, R. Kumar, P. Manurangsi, A. Sealfon, and A. V. Varadarajan, "Summary reports optimization in the privacy sandbox attribution reporting api," *arXiv preprint arXiv:2311.13586*, 2023.

[29] google.com, "Protected auction key/value service," https://github.com/privacysandbox/protected-auction-key-value-service.

[30] ——, "Key/value service user-defined functions (udfs)," https://github.com/privacysandbox/protected-auction-services-docs/blob/main/key_value_service_user_defined_functions.md.

[31] A. Sev-Snp, "Strengthening vm isolation with integrity protection and more," *White Paper, January*, 2020.

[32] intel.com, "Intel trust domain extensions (intel tdx)," https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html.

[33] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*. Springer, 2006.

[34] A. Kalra, "Add SEV-SNP CipherTextHiding feature support," 2024. [Online]. Available: https://lwn.net/Articles/985386/

[35] amd.com, "Performance counter side channel," https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3013.html.

[36] E. Aktas, C. Cohen, J. Eads, J. Forshaw, and F. Wilhelm, "Intel trust domain extensions (tdx) security review," *Google security review*, 2023.

[37] L. Wilke, F. Sieck, and T. Eisenbarth, "Tdxdown: Single-stepping and instruction counting attacks against intel tdx," in *ACM CCS 2024*, 2024.

[38] U. Mandal, S. Shukla, N. Mishra, S. Bhattacharya, P. Saxena, and D. Mukhopadhyay, "Exploring side-channels in intel trust domain extensions," Cryptology ePrint Archive, Paper 2025/079, 2025. [Online]. Available: https://eprint.iacr.org/2025/079

[39] amd.com, "Ciphertext side channels on amd sev," https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1033.html.

[40] Arm Architecture & Technology Group, "Arm CCA Security Model 1.0," 2021. [Online]. Available: https://documentation-service.arm.com/static/610aaec33d73a34b640e333b

[41] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*. Springer, 2006, pp. 265–284.

[42] A. Haeberlen, B. C. Pierce, and A. Narayan, "Differential privacy under fire," in *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011. [Online]. Available: http://static.usenix.org/events/sec11/tech/full_papers/Haeberlen.pdf

[43] Z. Ratliff and S. Vadhan, "A framework for differential privacy against timing attacks," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 3615–3629. [Online]. Available: https://doi.org/10.1145/3658644.3690206

[44] kernel.org, "Confidential computing vms," https://docs.kernel.org/virt/hyperv/coco.html.

[45] J. R. Douceur, "The sybil attack," in *IPTPS*, ser. Lecture Notes in Computer Science. Springer, 2002.

[46] security.apple.com, "Private cloud compute security guide," https://security.apple.com/documentation/private-cloud-compute/requestflow.

[47] J. De Meulemeester, L. Wilke, D. Oswald, T. Eisenbarth, I. Verbauwhede, and J. Van Bulck, "BadRAM: Practical memory aliasing attacks on trusted execution environments," in *46th IEEE Symposium on Security and Privacy (S&P)*, May 2025.

[48] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia, "{VoltPillager}: Hardware-based fault injection attacks against intel {SGX} enclaves using the {SVID} voltage scaling interface," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[49] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security Symposium*, 2018.

[50] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *CCS*, 2019.

[51] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-flight Data Load," in *S&P*, 2019.

[52] D. Moghimi, "Downfall: Exploiting speculative data gathering," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[53] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based fault injection using selective state reset," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[54] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.

[55] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," in *S&P*, 2020.

[56] N. Homer, S. Szelinger, M. Redman, D. Duggan, W. Tembe, J. Muehling, J. V. Pearson, D. A. Stephan, S. F. Nelson, and D. W. Craig, "Resolving individuals contributing trace amounts of dna to highly complex mixtures using high-density snp genotyping microarrays," *PLoS genetics*, 2008.

[57] S. Sankararaman, G. Obozinski, M. I. Jordan, and E. Halperin, "Genomic privacy and limits of individual detection in a pool," *Nature genetics*, 2009.

[58] C. Dwork, A. Smith, T. Steinke, J. Ullman, and S. Vadhan, "Robust traceability from trace amounts," in *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*. IEEE, 2015.

[59] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017.

[60] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, "Sev-step: A single-stepping framework for amd-sev," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023.

[61] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "{CopyCat}: Controlled {Instruction-Level} attacks on enclaves," in *29th USENIX security symposium (USENIX security 20)*, 2020.

[62] S. Florian, Z. Zhang, S. Berndt, C. Chuengsatiansup, T. Eisenbarth, and Y. Yarom, "TeeJam: Sub-Cache-Line Leakages Strike Back," in *TCHES*, 2024.

[63] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "Platypus: Software-based power side-channel attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2021.
[64] I. Puddu, M. Schneider, M. Haller, and S. Capkun, "Frontal attack: Leaking Control-Flow in SGX via the CPU frontend," in *30th USENIX Security Symposium (USENIX Security 20)*, 2021.
[65] S. Constable, J. Van Bulck, X. Cheng, Y. Xiao, C. Xing, I. Alexandrovich, T. Kim, F. Piessens, M. Vij, and M. Silberstein, "{AEX-Notify}: Thwarting precise {Single-Stepping} attacks through interrupt awareness for intel {SGX} enclaves," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.
[66] R. Zhang, T. Hornetz, L. Gerlach, and M. Schwarz, "Taming the Linux Memory Allocator for Rapid Prototyping," in *DIMVA*, 2025.
[67] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011.
[68] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/
[69] google.com, "Bringing transparency to confidential computing with slsa," https://security.googleblog.com/2023/06/bringing-transparency-to-confidential.html.
[70] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *Journal of the ACM (JACM)*, 1998.
[71] S. B. Mokhtar, A. Boutet, P. Felber, M. Pasin, R. Pires, and V. Schiavoni, "X-search: revisiting private web search using intel sgx," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017.
[72] google.com, "Oak functions containers," https://github.com/project-oak/oak/tree/main/oak_functions_containers_launcher.
[73] E. Stefanov, M. v. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: an extremely simple oblivious ram protocol," *Journal of the ACM (JACM)*, 2018.
[74] S. P. Vadhan, "The complexity of differential privacy," in *Tutorials on the Foundations of Cryptography*. Springer International Publishing, 2017, pp. 347–450.
[75] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Lightweight techniques for private heavy hitters," Cryptology ePrint Archive, Paper 2021/017, 2021. [Online]. Available: https://eprint.iacr.org/2021/017
[76] D. Mouris, C. Patton, H. Davis, P. Sarkar, and N. G. Tsoutsos, "Mastic: Private weighted heavy-hitters and attribute-based metrics," *Proceedings on Privacy Enhancing Technologies*, 2025.
[77] W3C, "Network error logging," W3C Working Draft, 2023. [Online]. Available: https://www.w3.org/TR/network-error-logging/
[78] J. Bell, A. Gascón, B. Ghazi, R. Kumar, P. Manurangsi, M. Raykova, and P. Schoppmann, "Distributed, private, sparse histograms in the two-server model," in *CCS*. ACM, 2022, pp. 307–321.
[79] D. Bogatov, G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "ϵpsolute: Efficiently querying databases while providing differential privacy," in *CCS*. ACM, 2021, pp. 2262–2276.
[80] S. Ghodke, "Alexa top 1 million sites," Kaggle, 2018. [Online]. Available: https://www.kaggle.com/datasets/cheedcheed/top1m
[81] I. E. Akkus, I. Rimac, and R. Chen, "Praas: Verifiable proofs of property as-a-service with intel sgx," in *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2024.
[82] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "AccTEE: A webassembly-based two-way sandbox for trusted resource accounting," in *Proceedings of the 20th International Middleware Conference*, 2019.
[83] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-faas: Trustworthy and accountable function-as-a-service using intel sgx," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019.
[84] Google, "What is v8?" https://v8.dev/.
[85] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "Lvi: Hijacking transient execution through microarchitectural load value injection," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.
[86] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves," in *CCS*. ACM, 2017.
[87] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.
[88] W. Wang, M. Li, Y. Zhang, and Z. Lin, "Pwrleak: Exploiting power reporting interface for side-channel attacks on amd sev," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2023.
[89] K. Ryan, "Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.
[90] F. Dall, G. De Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 2, 2018.
[91] I. Corparation, "Guidelines for mitigating timing side channels against cryptographic implementations," 2021.
[92] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.
[93] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "{CacheD}: Identifying {Cache-Based} timing channels in production software," in *26th USENIX security symposium (USENIX security 17)*, 2017.
[94] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying {High-Performance} cryptographic assembly code," in *26th USENIX security symposium (USENIX security 17)*, 2017.
[95] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017.
[96] Y. Yuan, Z. Liu, S. Deng, Y. Chen, S. Wang, Y. Zhang, and Z. Su, "Ciphersteal: Stealing input data from tee-shielded neural networks with ciphertext side channels," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024.
[97] ——, "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*, 2024, pp. 4346–4360.
[98] J. Jin, E. McMurtry, B. I. P. Rubinstein, and O. Ohrimenko, "Are we there yet? timing and floating-point attacks on differential privacy systems," *CoRR*, vol. abs/2112.05307, 2021. [Online]. Available: https://arxiv.org/abs/2112.05307
[99] M. Rainer, L. Hetterich, F. Thomas, T. Hornetz, L. Trampert, L. Gerlach, and M. Schwarz, "Rapid Reversing of Non-Linear CPU Cache Slice Functions: Unlocking Physical Address Leakage," in *S&P*, 2025.
[100] ——, "Rapid reversing of non-linear cpu cache slice functions: Unlocking physical address leakage artifact repository," 2025. [Online]. Available: https://github.com/cispa/LLCSliceReversing

APPENDIX

A. Connecting DP & Advantage

Lemma 2 (DP bounds advantage)**.** *If the leakage of B guarantees (ε, δ)-DP, a pairwise distinguishability attack against B has advantage bounded by $(e^{\varepsilon} - 1)/4 + \delta/2$.*

Let c be the bit that indicates whether the target has x_0 or x_1. Observe that

$$\Pr[\mathsf{output}_{\mathcal{A}} = \mathsf{c}] = \frac{1}{2}\big(\Pr[\mathsf{output}_{\mathcal{A}} = 0 \mid \mathsf{c} = 0] + \Pr[\mathsf{output}_{\mathcal{A}} = 1 \mid \mathsf{c} = 1]\big)$$

because c is uniform over $\{0,1\}$.

Because $\Pr[\mathsf{output}_{\mathcal{A}} = \mathsf{c}] > 1/2$, there must be at least one $i \in \{0,1\}$ where $\Pr[\mathsf{output}_{\mathcal{A}} = i \mid \mathsf{c} = i] > 1/2$ otherwise the mean would be $\leq 1/2$. We rewrite the above equality using i:

$$\begin{aligned}
&\Pr[\mathsf{output}_{\mathcal{A}} = \mathsf{c}] \\
=\ &\frac{1}{2}\big(\Pr[\mathsf{output}_{\mathcal{A}} = i \mid \mathsf{c} = i] \\
&+ \Pr[\mathsf{output}_{\mathcal{A}} = 1-i \mid \mathsf{c} = 1-i]\big) \\
\leq\ &\frac{1}{2}\big(\Pr[\mathsf{output}_{\mathcal{A}} = i \mid \mathsf{c} = i] \\
&+ e^{\varepsilon}\Pr[\mathsf{output}_{\mathcal{A}} = 1-i \mid \mathsf{c} = i] + \delta\big)
\end{aligned}$$

The inequality comes directly from the definition of DP.

We can continue the analysis by adding and subtracting the quantity $\Pr[\mathsf{output}_{\mathcal{A}} = 1-i \mid \mathsf{c} = i]$:

$$\begin{aligned}
=&\frac{1}{2}\big(\Pr[\mathsf{output}_{\mathcal{A}} = i \mid \mathsf{c} = i] + \Pr[\mathsf{output}_{\mathcal{A}} = 1-i \mid \mathsf{c} = i] \\
&+ (e^{\varepsilon}-1)\Pr[\mathsf{output}_{\mathcal{A}} = 1-i \mid \mathsf{c} = i] + \delta\big) \\
=&\frac{1}{2}\big(1 + (e^{\varepsilon}-1)\Pr[\mathsf{output}_{\mathcal{A}} = 1-i \mid \mathsf{c} = i] + \delta\big)
\end{aligned}$$

Now notice that $\Pr[\mathsf{output}_{\mathcal{A}} = 1-i \mid \mathsf{c} = i]$ is at most $1/2$ by virtue of the definition of i. Hence,

$$\Pr[\mathsf{output}_{\mathcal{A}} = \mathsf{c}] \leq \frac{1}{2}\left(1 + \frac{e^{\varepsilon}-1}{2} + \delta\right)$$

which in turn means that the advantage is $\leq \frac{e^{\varepsilon}-1}{4} + \frac{\delta}{2}$. We remark that this analysis is only useful for $\varepsilon < \ln(3-2\delta)$; otherwise, the advantage would be bounded by a number $\geq 1/2$.

Figure 9 shows our proposed DP-based mitigation.

B. Trace and Collection Speed

	Time per NPF	NPF Tracked/sec
Page-level	18,841 CPU cycles	159,227
Ciphertext	22,543 CPU cycles	133,079
Cache attacks	248,568 CPU cycles	12,069

TABLE V: The collection speed with different leakage choices with a 3.0 GHz CPU.

Figure 10 shows an example of the raw trace, where *MA 141b69 CL 60* indicates access to the 60th 64B of the guest page at `0x141b69`. *ci_bk* is followed by the 16B index in the page and the ciphertext value before and after the change. Table V shows the collection speed of SNPeek under different leakage choices, averaged over 10,000 NPF using the *Vanilla* application in Section V as the benchmark. In cache attacks, we disable the hardware prefetcher and ensure a clean cache state at the prime state by executing the `wbinvd` instruction [53].

Automatic Feature Learning To aid the sequence model, we pre-process the traces by abstracting the memory space, and maintaining information about the ciphertext. In Figure 11 we show the result of pre-processing the trace in Figure 10.

```
// batch, epsilon, and threshold are defined in
// the context
std::unordered_map<string, int> hist;
std::vector<std::pair<string, int>> result;
...
// Mitigation params
double kEpsilonDummies = atof(argv[1]);
double kLaplaceScaleDummies = 2 / kEpsilonDummies;
double kTailBound = log(1 / kDelta) / kEpsilonDummies;
std::exponential_distribution<double>
    expDummies(1 / kLaplaceScaleDummies);

// Aggregate inputs
...

// Add dummies
int nDummies = 0;
if (kEpsilonDummies > 0) {
    nDummies = 1 +
        int(sample_centered_laplace(expDummies)) +
        kTailBound;
}
for (int i = 1; i <= nDummies; ++i) {
    hist[-i]++;
}

// Noise and Threshold
...
}
```

Fig. 9: A mitigation for the obvious leakage in the Noise & Threshold phase of the vanilla PHH implementation.

```
CF <gpn> npf_num:<num> ---------- CF 10abda npf_num:1926

MA <gpn> CL <64B_idx> ... ---------- MA 141b69 CL 01 60 61 62 63
                                     MA 1a35f9  CL 26 35 36 37
64/16B_idx: Memory block index accessed / changed

MA <gpn> ci_bk_<16B_idx> B:<Ci_b> A:<Ci_a> ------- 141b69 ci_bk_240 B:c030 A:ab0b
                                                   141b69 ci_bk_244 B:020a A:faff
Ci_b/a: First two-byte ciphertext before / after change  1a35f9  ci_bk_144 B:1482 A:8479

PMCs <pn1> <pn2> <pn3> pn4> <pn5> <pn6> --------- PMCs 40 0 7 6 0 0
                                                  CF 10abf7 npf_num:1927
```

Fig. 10: The syntax and example of side-channel trace. "gpn" represents the guest physical page number and "num" records the number of code pages monitored. "pn" represents the performance counter values of attacker-chosen events.

```
CF 1
MA 1
MA 2
MA 1 CI_BK 240
MA 1 CI_BK 244
MA 2 CI_BK 144
CF 2
```

Fig. 11: The trace in Figure 10 after pre-processing for sequence model.

Sequence Model We implement an LSTM model according to the architecture described in Table VI. The vocabulary size is set to 10,000, and the traces are truncated to the last 5000 tokens. The model uses a batch size of 32 and is trained for 50 epochs using early stopping.

Layer	Dimm	# Params
Input	10,000	0
Embedding	64	640,000
Bidirectional	128	66,048
Bidirectional	64	41,216
Dense	64	4,160
Dropout	64	0
Dense	2	130

TABLE VI: The architecture of the LSTM model used throughout the case studies.

C. Intel CAT

We evaluate our optimized Prime+Probe attack on an Intel Xeon E5-2697 v4. This platform features a 20-way inclusive L3 cache, where a "perfect" eviction set normally requires 22 (20+2) candidates mapping to the same slice and set [99]. We employed the `pqos` utility to leverage Intel CAT, defining a new Class of Service (CLOS) with an L3 Capacity Bitmask (CBM) of `0xff`. This restricted assigned cores to 8 L3-cache ways (down from the default 20). We then associated the core executing the AES T-table benchmark with this restricted CLOS [100]. This intervention yielded the expected outcome, reducing the required perfect eviction set size from 22 to just 10 (8+2) candidates. This finding confirms that a privileged attacker can co-opt hardware-based cache reservation mechanisms (e.g., Intel CAT or AMD MSRs) to force a victim into a shared, low-associativity partition. This makes a successful Prime+Probe attack quantifiably less complex and resource-intensive.

D. System Noise Circumvention

In this section, we discuss the noise sources for the four side-channel types and the hypervisor's ability to circumvente them.

- **Page Faults and Ciphertext:** Both of these channels are inherently deterministic. The sequence of page accesses is dictated by the victim program's execution flow, just as the ciphertext leakage is dictated by its memory write operations. Consequently, these channels are not susceptible to external system noise.
- **PMCs:** PMCs are a per-core resource. A hypervisor can configure them to record only events originating from the guest VM. By scheduling other VMs (if any) to different cores, the hypervisor can dedicate a core to the victim, effectively eliminating any cross-VM interference and ensuring a noise-free PMC trace.
- **Cache Attacks:** The noise profile for cache attacks depends on the targeted cache level. We target the L2 cache, which is shared only within the physical core. Importantly, the shared L3 cache in the targeted AMD microarchitecture is non-inclusive. This means memory activities on other physical cores do not cause evictions in the L2 cache where the victim is sharing, thus preventing them from interfering with our L2-based Prime+Probe. The only remaining potential noise source is the sibling hyper-thread (SMT) on the same core. However, a privileged hypervisor can easily eliminate this by disabling SMT, granting the victim VM exclusive access to all L2 cache ways.

APPENDIX A
ARTIFACT APPENDIX

A. Description & Requirements

This artifact contains the framework for auditing privacy applications running inside Confidential VM, described in the paper "SNPeek: Side-Channel Analysis for Privacy Applications on Confidential VMs". It includes everything needed to reproduce the experiments in the paper, such as the customized hypervisor with instrumentation, data collected from confidential VMs, and the scripts for processing that data.

1) How to access: The artifact is publicly available in a GitHub repository: https://github.com/google-parfait/cvm-side-channel-analysis. It is also available on Zenodo at: https://doi.org/10.5281/zenodo.17542706.

2) Hardware dependencies: This artifact requires AMD SEV-SNP VMs, and thus an AMD EPYC server (Zen 3, 4, or 5 generation) with SEV-SNP support is necessary. The experiments in the paper were conducted on a 16-core AMD EPYC 9124 CPU. Root privileges are required to install and run a customized kernel, along with the ability to reboot the machine, in order to fully evaluate the artifact.

3) Software dependencies: All our experiments are tested on Ubuntu 24.04 LTS (Linux kernel 6.11.0). QEMU and OVMF are required to launch SEV guest VMs, as referenced from AMD's official GitHub repository[4]. Finally, we use the libtea framework to modify page table entries and configure the APIC timer[5].

4) Benchmarks: None.

B. Artifact Installation & Configuration

This section describes how to install the software components for inspection, though running the experiments requires the specific hardware listed above.

C. Experiment Workflow

To access the artifact, clone the repo.

```
git clone https://github.com/google-parfait/cvm-side
    -channel-analysis.git
cd cvm-side-channel-analysis
```

To prepare the host OS and guest VM, following the guidance under *trace_collection /kernel_patch/*:

```
cd trace_collection/kernel_patch
```

To collect traces from applications running inside the guest VM, you can follow the step-by-step instructions in the README under *trace_collection* . The script for setting up the environment and batching the collection process is located in *trace_collection/sca_dp*.

The *trace_processing* folder contains basic pipelines for processing traces, extracting features, analyzing data, and applying machine learning.

[4]https://github.com/AMDESE/AMDSEV
[5]https://github.com/libtea/frameworks