



TDXRay: Microarchitectural Side-Channel Analysis of Intel TDX for Real-World Workloads

Tristan Hornetz^{*‡}, Hosein Yavarzadeh^{*†§},
Albert Cheu[§], Adria Gascon[§], Lukas Gerlach[‡],
Daniel Moghimi[§], Phillipp Schoppmann[§], Michael Schwarz[‡], Ruiyi Zhang[‡]
**Equal contribution joint first authors †University of California San Diego ‡CISPA §Google*

Abstract—Confidential computing with VM-based trusted execution environments (TEEs) promises to protect code and data from a privileged cloud operator, enabling privacy-preserving workloads ranging from medical analytics to AI inference. However, most deployments exclude microarchitectural side channels from their threat model, shifting the burden to application developers who lack practical, general-purpose tools to assess (let alone mitigate) leakage. In particular, it remains unclear which host-observable signals persist under TDX’s strict isolation and whether these signals can reveal sensitive information about confidential workloads.

In this paper, we systematically investigate the side-channel attack surface in Intel TDX. We identify four new side-channel primitives: SEPTrace, Load+Probe, TSX-Probe, and MWAIT-Probe. Together, they expose page-level and cache-level activity with varying temporal precision. By combining these primitives, we construct TDXRay, a host-side measurement framework that produces highly accurate, cache-line-granular memory access traces of unmodified confidential VMs. Using TDXRay, we build two case studies: (1) a classic AES T-table attack in which a malicious hypervisor recovers the secret key from access-pattern leakage, and (2) an attack against large language models in which the host infers user prompts by monitoring memory accesses during tokenization. Our evaluation demonstrates that TDXRay can reliably recover user prompts from a single memory access trace, thus posing a severe threat to private LLM inference.

Finally, we investigate and discuss mitigation strategies at system and application level. While effective countermeasures based on ORAM can be a short-term solution, our results highlight the need for long-term investment in improving Intel TDX and similar architectures against this class of attacks.

1. Introduction

Confidential compute architectures protect data in use by executing workloads inside hardware-enforced isolated environments. With this goal, modern CPUs encrypt memory and restrict access from higher-privileged software, thereby reducing the trusted computing base to the processor package. Recent offerings such as AMD SEV-SNP [1], Intel TDX [2], and ARM CCA [3] have made confidential virtual machines (CVMs) accessible to cloud users, promising that

a malicious cloud operator cannot inspect or alter guest-private memory or CPU state. Cloud providers now offer CVM instances on commodity infrastructure, which are increasingly used to host sensitive workloads, including private data analytics [4], financial applications [5], [6], and large language model (LLM) inference [7]–[9].

While these technologies significantly strengthen security, addressing side channel attacks remains an open problem. The host and guest continue to share physical resources such as caches, interconnects, and memory controllers, and the host retains operational control over scheduling, memory management, and I/O. These components can transmit information about guest activity through indirect effects such as cache contention [10], [11] or address translation events [12]. Although such side channels are often excluded from official threat models, they remain a realistic vector for information leakage in multi-tenant cloud environments. Understanding these residual risks is essential for both cloud operators and developers of confidential workloads.

However, quantifying the leakage that persists under modern CVM architectures remains challenging. The strong isolation enforced by TDX and SEV-SNP prevents many attack vectors known from native code [13]–[16]. At the same time, new architectural interfaces, such as restricted page management [2] or CPU hints for entering efficient idle states [17], introduce subtle new observables that a host can still monitor. These observables are often low-level, undocumented, and hardware-dependent, making it difficult to assess whether they can meaningfully reveal information about guest execution. Moreover, the boundary between “in-scope” and “out-of-scope” threats in confidential computing deployments remains ambiguous. Vendors typically exclude microarchitectural side channels from their formal threat model [1], [18], even though they still pose a risk to users, especially in the cloud. Thus, as a middle ground, Intel has taken some measures to make side-channel attacks against its CVM solution, Intel TDX, more difficult. These measures include a mitigation against single stepping [2], [19], and the blocking of direct access to page tables from the host [2], preventing controlled-channel attacks [20].

In this paper, we ask the following research question:

What is the remaining side-channel attack surface for complex workloads running inside Intel TDX?

We focus on Intel TDX as a representative confidential computing platform with specific measures against side-channel attacks. TDX isolates guest memory from the host by encrypting all guest-private pages and executing the guest in a so-called trusted domain (TD). The TDX module mediates all interactions between host and guest, exposing only a narrow set of management operations required for regular VM operation, such as page acceptance and promotion, and vCPU scheduling. In principle, this design prevents the host from directly observing guest activity. In practice, however, the host still shares the processor’s caches and interconnects with the guest. Additionally, the limited TDX management interfaces can still leak measurable feedback about guest-physical memory behavior.

We introduce four new host-observable primitives that enable side-channel leakage from Intel TDX:

- (a) **SEPTrace** captures guest page activity through controlling page access. Intel TDX supports a standard API that allows the untrusted host to selectively block or unblock access to a TD’s memory pages. SEPTrace shows that this interface allows the host to efficiently track access to multiple memory pages by continuously blocking access to all but one page within a critical section of a target program.
- (b) **Load+Probe** detects the cache state of a target memory address via a timing-based side channel. While TDX prevents a host from reading a TD’s memory, the host can still directly measure the execution time of such memory accesses to determine if a target cache line is cached, allowing a malicious host to perform precise cache attacks.
- (c) **TSX-Probe** employs hardware transactions to detect cache state without precise timers. If a request to read private memory is wrapped in a TSX transaction, the transaction’s success depends on the cache state. Attackers can exploit this behavior to check whether a target memory location is cached.
- (d) **MWAIT-Probe** uses the `mwait` instruction, which is designed for hardware sleep/wake-up events synchronization. Following previous work [17] which showed the `mwait` instruction relies on the physical address, we develop a primitive that allows a host to track guest memory even when the host cannot read it using an incorrect encryption key handle.

By combining these primitives into a hybrid monitoring approach, we create TDXRay, a practical host-side framework for recording accurate, cache-line-granular memory access traces of TDX CVMs. TDXRay thus provides a flexible toolkit for host-side leakage analysis, which operates entirely within the boundaries of a legitimate cloud host and requires no guest cooperation. We use TDXRay to conduct a detailed case study of LLM inference within TDX guests. LLM inference is an emerging confidential-computing workload, where cloud providers host proprietary models and process user data inside remotely attestable CVMs [7], [9], [21], [22]. We specifically analyze the leakage signals produced by tokenization, a phase that involves data-dependent memory accesses and is typically performed on the CPU,

not the GPU. Our experiments show that a host observing TDX management interfaces and cache behavior can infer substantial portions of user prompts, even though all guest memory remains encrypted, with high accuracy: for Llama 3.2, we achieve an average similarity score of 91.5%, and for Gemma 3, the average score further increases to 94.2%.

We further investigate potential mitigations. Although a comprehensive elimination of our cache side channel primitives necessitates design changes to the underlying hardware, software can take defensive measures to reduce their practical impact. For attacks on LLMs in particular, we construct a data-oblivious tokenization scheme with an acceptable performance overhead. Additionally, we outline potential system-level mitigations that hardware manufacturers can employ.

Contributions. In summary:

- We identify four novel side channels on Intel TDX, which cover page- to cache-line-level leakage and enable synchronized, low-noise measurements. We quantify them using an AES T-table attack benchmark.
- We design TDXRay, a host-side framework combining these primitives to record cache-granular access traces of unmodified CVM workloads without violating platform boundaries.
- We use TDXRay in an end-to-end attack on LLM inference inside TDX, leaking confidential information from user prompts.
- We investigate mitigations and employ data-oblivious tokenization to eliminate side channel leakage from LLM tokenizers with an acceptable performance.

2. Background

This section covers some preliminaries.

2.1. Confidential VMs

Trusted Execution Environments (TEEs) provide an isolated environment that even supervisor-mode software, such as kernels and hypervisors, cannot tamper with directly. TEE-enabled CPUs enforce this with various confidentiality and integrity protection measures, such as memory encryption and authentication. While early TEEs such as Intel SGX [23] and ARM TrustZone [24] were primarily designed for consumer-grade platforms, providing protection for cryptographic operations and Digital Rights Management (DRM), more recent TEE architectures are explicitly aimed at cloud and multi-tenant environments. TEEs like AMD SEV [25], AMD SEV-SNP [1], ARM CCA [3], and Intel TDX [26] can protect entire virtual machines, so-called Confidential Virtual Machines (CVM). This restricts rogue hosting providers or compromised hypervisors from exfiltrating tenant data, enabling the processing of confidential data on shared hardware.

Intel TDX. Recent server-grade Intel CPUs support CVMs with Trust Domain Extensions (TDX), which builds on

existing technologies like Multi-Key Total Memory Encryption (TME-MK) and Intel SGX. CVMs with TDX, called Trust Domains (TDs), run in so-called *private memory*, which is encrypted and, optionally, integrity-protected with cryptographic MACs. Each TD has an ephemeral encryption key, stored in the memory controller and identified by a unique Hardware Key Identifier (HKID) in the upper physical address bits. Additionally, to prevent ciphertext side channels [27] and chosen-plaintext attacks [28], the memory controller forwards a static data pattern when an untrusted component attempts to read from private memory. Writes are not prohibited, but trigger an integrity check violation when a TD accesses that memory area. Only explicit *shared memory* segments of a TD are unencrypted, enabling interaction with peripheral devices.

All interactions between TD and the untrusted host are governed by the *TDX Module*, a trusted component signed by Intel that is itself protected by TEE-like confidentiality and integrity measures. Only the TDX Module has direct access to a TD’s low-level control structures. To allocate TD memory, perform scheduling tasks, or handle exceptions, untrusted hypervisors rely on the TDX Module’s host-side API with the `seamcall` instruction. Hence, the hypervisor retains a high degree of control but is prohibited from tampering with a TD.

Another key element of TDX isolation is the Secure Extended Page Tables (SEPT). Like most hardware-backed VMs, virtual addresses in a TD are translated twice. The first step, translating a virtual address to a so-called guest-physical address (GPA), uses guest-controlled page tables, allowing guest kernels to perform independent memory management. The second step, translating GPAs to DRAM-backed host-physical addresses (HPAs), uses SEPT entries managed by the TDX Module. An untrusted hypervisor can request page mappings via the `seamcall` API, but it cannot arbitrarily modify guest mappings in a way that would compromise isolation.

Side Channels in Confidential VMs. Side channels arise when a program’s execution influences shared hardware resources in ways that allow an adversary to infer secret information through observing indirect effects. In the context of CVMs, a malicious host can exploit resource sharing to monitor guest activity, even though direct access to guest memory is blocked. The most relevant classes of such channels include controlled-channel attacks [20], cache attacks [10], and contention-based attacks [29]. Controlled-channel attacks manipulate page-table permissions to trigger and observe page faults, revealing guest memory access patterns. Originally shown for Intel SGX [20], they were also demonstrated on AMD SEV [30]. Cache attacks measure memory access latency to infer whether data resides in the cache, allowing fine-grained observation of victim code or data accesses. The generic Prime+Probe cache attack has been demonstrated on AMD SEV [31] and Intel TDX [26]. Additionally, Rauscher et al. [19] demonstrate a cache-line-granular Flush+Flush cache attack on Intel TDX. Despite this large variety of attacks, the TDX threat model explicitly

TABLE 1. MEMORY ACCESS ORACLES FOR TDX PRIVATE MEMORY

Primitive	Granularity	Cross-Core	Synchronizing	Rd/Wr
SEPTTrace	4 kB/2 MB	●	●	○
Load+Probe	64 B	●	○	●
TSX-Probe	64 B	○	○	○
MWAIT-Probe	64 B	●	●	○

excludes microarchitectural side channels and leaves their evaluation and containment to the developers.

2.2. Private Inference

Traditional cloud environments expose user data and model weights to potential threats. CVMs enable secure AI workloads in cloud settings, providing strong confidentiality for so-called *private inference* [9], [32] on shared hardware. Several industry players have implemented or are exploring confidential computing for AI applications: OpenAI proposes trusted hardware platforms for advanced AI [21], Google has integrated generative AI into TEEs [7], Intel provides open-source reference solutions for confidential AI using TDX-based CVMs [33], Meta leverages TEEs for private AI processing in WhatsApp [8], and Anthropic has developed trusted loaders for confidential AI inference within VMs [9]. Azure provides a comprehensive confidential computing ecosystem with CVMs and confidential GPUs (e.g., NVIDIA H100) integrated with Azure Machine Learning, enabling private inference at large scales [32]. These advancements demonstrate a rapidly growing demand for private inference, making its confidentiality a key requirement for many real-world applications.

3. Identifying CVM Side Channels

In this section, we conduct an in-depth analysis of side-channel attacks on TDX, discuss existing attacks, and identify new ones. While the fundamental categories of these side-channels—such as page-fault and cache-based monitoring—are well-known in the literature, Intel TDX’s architectural design actively mitigates traditional exploitation techniques. Therefore, realizing these attacks on TDX requires discovering and reverse-engineering undocumented microarchitectural behaviors to bypass these specific isolation barriers. Our contribution lies in adapting these known concepts into practical, TDX-specific primitives and demonstrating their viability on real hardware. We identify four separate side-channel primitives revealing a victim TD’s memory accesses to an untrusted privileged attacker. Table 1 lists these primitives along with their spatial granularity, cross-core observability, ability to serve as a synchronization primitive, and capability to distinguish between victim reads and writes (Rd/Wr).

Threat Model. We assume a malicious or compromised host with full control over the hypervisor and complete access to all architectural interfaces exposed by Intel TDX.

The host cannot directly access guest memory or registers due to hardware-enforced isolation. Direct interaction between the hypervisor and a guest TD is restricted to the interface exposed by the TDX module’s APIs. The hypervisor retains privileges to schedule the guest, manage its memory, and resolve guest-physical addresses (GPAs) to host-physical addresses (HPAs).

We assume the guest (victim) is uncompromised and does not cooperate with the host (attacker). The attacker aims to infer the victim’s memory activity. We explicitly exclude attacks that rely on hardware faults and physical access (e.g., power or electromagnetic emissions). We focus on assessing how much information remains observable through legitimate host interfaces and shared microarchitectural resources within TDX’s threat model.

Experimental Setup. Unless stated otherwise, all experiments are conducted on three Intel TDX-enabled CPUs: Xeon Silver 4510 (Sapphire Rapids), Xeon Gold 6526Y (Emerald Rapids), and Xeon 6736P (Granite Rapids). Each system is equipped with 128 GB of RAM and hyperthreading-enabled cores to support same-core and cross-core measurements. Turbo Boost is disabled, and CPUs are pinned to fixed frequencies to minimize timing noise. The host runs Linux v6.8.0 with KVM and QEMU supporting TDX `seamcalls`. Guests are unmodified Linux confidential VMs (TDs) configured with a single vCPU and 8 GB of memory, running Ubuntu 24.04 and Linux v6.8.0.

3.1. Page-Table Side Channels

An adversary controlling page tables (OS) or nested page tables (hypervisor) can force faults on victim accesses and thereby record page-granularity traces of a program’s secret-dependent memory behavior [20]. In Intel SGX and AMD SEV-SNP, the attacker can revoke permission bits so that each victim access to a targeted virtual or guest-physical page triggers a fault that the adversary can observe and timestamp; repeating this across pages reveals control-flow and data-access patterns [27], [34]. In contrast, the TDX architecture introduces a separate extended page-table entry (SEPT) accessible only to a trusted firmware (TDX Module) [2], posing an additional challenge to attackers.

GPA-Range Block/Unblock API. While a malicious host cannot directly access SEPT, Intel TDX exposes host-side `seamcall` leaf functions enabling a hypervisor to temporarily make private GPA ranges non-translatable [35]. These functions support dynamic page manipulation and TD migration, which require the hypervisor to guarantee “no active translations” for specific GPA ranges [2]. The two primary API leaves are:

- `TDH.MEM.RANGE.BLOCK` marks a GPA range as *blocked*, preventing new translations in that range. Internally, this clears the SEPT entry’s read, write, and execute bits, effectively marking the page as non-present.
- `TDH.MEM.RANGE.UNBLOCK` returns a previously blocked range to its prior state.

Additionally, the hypervisor can initiate a so-called “TLB tracking” sequence, ensuring that TLB entries for previous SEPT translations are flushed [2]. Depending on which page size the GPA ranges use, these API leaves support multiple granularity levels (e.g., 4 kB, 2 MB, and 1 GB) [2].

Side-Channel Attack via Block/Unblock APIs. A malicious hypervisor can construct a deterministic, page-granularity access trace by exploiting the blocking APIs to intercept individual page accesses. The hypervisor initiates the attack by blocking a target GPA range with `TDH.MEM.RANGE.BLOCK` and completing TLB tracking to ensure no stale translations exist. Next, it resumes the TD. Upon the TD’s first access to any page in the blocked region, the CPU raises a Secure-EPT violation, which is conceptually similar to a page fault. This triggers a TD-exit, which the TDX Module forwards to the untrusted hypervisor’s exit handler. This informs the hypervisor about the faulting GPA and exit qualification, thereby identifying the precise page the TD accessed. The hypervisor logs the GPA, unblocks only that page via `TDH.MEM.RANGE.UNBLOCK`, and resumes the TD while leaving all other pages in the range blocked. This process repeats for each subsequent access to another page within the still-blocked range. The resulting ordered sequence of GPAs constitutes a deterministic, page-granularity trace capturing the victim guest’s memory access pattern at page-granularity.

Practical Refinement. In a targeted attack, monitoring ranges at the lowest possible granularity level (i.e., 4 kB) naturally provides attackers with the highest spatial resolution. However, when tracing large memory regions, profiling at this granularity may be costly for an attacker. Since the TDX Module allows hypervisors to change the page size in a GPA range (i.e. *promoting* 4 kB pages to 2 MB and *demoting* pages from 2 MB to 4 kB), attackers can adopt a coarse-to-fine strategy: first block at 2 MB granularity to localize regions of interest, then demote this region to 4 kB pages to obtain a fine-grained access trace. This technique is especially useful for reconnaissance tasks, where the exact location of a target range is still to be identified.

Takeaway (1): Although TDX limits access to SEPT, the Block/Unblock API allows attackers to side-step that protection and obtain page-granularity traces. A malicious hypervisor can improve page-table side channels by promoting (merging) and demoting (splitting) pages.

3.2. Cache Side Channels

TDX prevents access to a TD’s private memory regions at the architectural level, but TDs still share microarchitectural resources, such as caches, with untrusted code. Consequently, cache side-channel attacks like Prime+Probe [19] are effective against TDX private memory regions. Google’s initial security report explored the impact of *physical alias addresses* to TDX private memory [35], i.e., an address with the physical address bits of a TDX private memory page, but a different HKID (indicating a different decryption key). In non-TDX contexts (e.g., within the hypervisor), accesses to

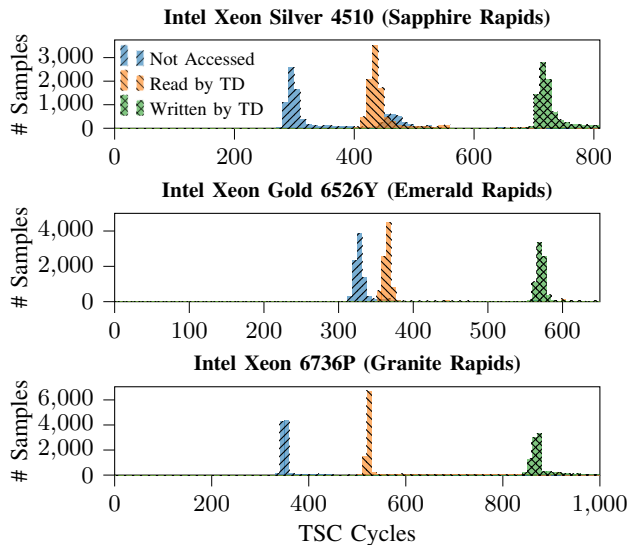


Figure 1. Distribution of read access times for the physical alias of TDX private memory in the hypervisor ($n = 10^5$). In the TD, the corresponding cache line was previously either read, written, or not accessed.

such addresses return a fixed all-zero pattern from the memory controller, preventing ciphertext leakage. While accessing physical aliases of private memory cannot read the ciphertext, such “invalid accesses” enables targeted eviction of a TD’s cache lines, boosting the effectiveness of attacks like *Prime+Probe* [35]. Similarly, Rauscher et al. [19] demonstrate that the `clflush` instruction ignores the HKID in physical alias addresses, allowing for *Flush+Flush* attacks on private memory. We extend these findings by introducing new low-noise primitives that amplify cache side channels: *Load+Probe*, *TSX-Probe* and *MWAIT-Probe*.

Load+Probe. We observe the *access times* of physical alias addresses in the hypervisor. In particular, we show that the time required for reading from a physical alias address depends on whether the TD previously accessed the corresponding cache line. Moreover, these access times reveal the TD’s access mode, i.e., read or write. To illustrate this, consider Figure 1, which shows the access time distribution for alias addresses on various TDX-enabled CPUs. When the TD does not access the cache line before our measurement, access time matches a DRAM fetch of about 350 TSC (Time Stamp Counter) cycles. However, if the TD previously read the cache line, latency increases by more than 150 TSC cycles on the Xeon 3736P. A prior write causes an even larger delay, up to twice as long. As an access oracle, this side-channel has numerous advantages. First, we only observe timing differences for exact physical aliases of the cache line we target. If the TD accesses a different cache line, even if it maps to the same L1D or L2 cache set, we observe no timing differences. In contrast to attacks like *Prime+Probe*, which suffer from interference by accesses to the same cache set, we can thus precisely target a single 64-byte cache line in the victim TD. Second, timing differences

TABLE 2. PMC DIFFERENCES FOR LOAD+PROBE WITH VARIOUS TYPES OF VICTIM TD ACTIVITY, MEASURED IN THE VICTIM’S SIBLING HYPERTHREAD.

Event	No Access	TD Read	TD Write
MEM_LOAD_RETIRED.L1_MISS	1	0	0
ASSISTS.HARDWARE [†]	0	1	1
MACHINE_CLEARS.MEMORY_ORDERING	0	1	1
CORE_SNOOP_RESPONSE.I_HIT_FSE*	0	1	0
CORE_SNOOP_RESPONSE.I_FWD_M*	0	0	1

* Event exists on Emerald Rapids only † Event exists on Granite Rapids only

similar to those shown in Figure 1 occur when accessing an alias address on any physical core, including cores that never run TD code. This access oracle thus works without hyperthreading and does not require the TD to be stopped or rescheduled, enabling cross-core cache attacks. Finally, we can distinguish whether a cache line was read or written. Thus, even without prior knowledge of the victim’s memory layout, we can infer whether a victim page holds constant data (e.g., code) or mutable data (e.g., stack variables).

Takeaway (2): The access time of a physical alias address reveals whether it was read, written or not accessed. An attacker can target a single 64-byte cache-line without interfering with the TD.

PMC Effects with Load+Probe. To identify the root cause of *Load+Probe*’s timing differences, we examine how it influences the host’s performance monitoring counters (PMCs) with different types of victim activity. We evaluate this on an Intel Xeon Gold 6526Y (Emerald Rapids) and Xeon 6736P (Granite Rapids), profiling 411 and 436 documented PMC events [36]. In each trial, the TD either reads, writes, or performs no access to a cache-line in private memory, while the untrusted host profiles a single load to its physical alias and records the PMC difference as a sample. For each PMC and victim activity type, we take 10 000 independent samples, and discard values outside the 1st and 99th percentile to eliminate outliers. To determine whether the PMC sample distributions differ with different victim activity, we perform a permutation test based on the absolute difference of means with 100 000 permutations to obtain a two-sided p -value. We consider two distributions for the same PMC event meaningfully different (i.e., revealing victim activity) if $p < 0.05$ and Cohen’s $d > 0.8$. Additionally, we perform this experiment in two scenarios: (1) while profiling in the TD’s sibling hyperthread and (2) while profiling on a separate physical core.

We use this method to find 64 PMC events on the Xeon Gold 6526Y that differ in the hyperthread, and 26 events show differences even across physical cores. On the Xeon 6736P, we find 70 events that differ in the hyperthread, and 33 that differ across physical cores. Most events count cycles in specific scenarios, thus providing comparable results to the TSC. For instance, the `CYCLE_ACTIVITY.CYCLES_MEM_ANY` event, which

```

1 is_cached_in_tdx:
2  xbegin abort_dest
3  mov rdi, [rdi]
4  xend
5
6  xor rax, rax
7  ret
8
9 abort_dest:
10 mov rax, 1
11 ret

```

Listing 1: Intel TSX as an access oracle. If the physical address of the pointer in `rdi` was cached in TDX, the TSX transaction aborts, and the function thus returns 1. If it is not cached, the transaction commits, and `rax` is set to 0.

counts cycles with outstanding memory loads, differs between conflicting victim reads, writes, and no activity, even on separate physical cores. When probing in the sibling hyperthread, we also identify events that increment only once, depending on the victim’s activity on the target cache line. See Table 2, which shows a subset of these events. For conflicting victim activity, we confirm that the untrusted hypervisor’s memory access hits the L1 cache despite the differing HKIDs. However, this triggers a microcode assist and a “memory ordering” machine-clear event [36], both of which likely factor into Load+Probe’s timing differences. Moreover, we confirm that Load+Probe invalidates the conflicting cache line. For TD writes, the `CORE_SNOOP_RESPONSE.I_HIT_M` event reveals that the conflicting cache-line is in the MESIF protocol’s *Modified* (M) state, meaning that invalidation causes a costly writeback to DRAM [37]. With TD reads, it is in the *Forward* (F), *Shared* (S), or *Exclusive* (E) state, for which cache lines can be safely dropped without writeback. We speculate that this causes the additional slowdown with victim writes.

When probing the events in Table 2 in a separate physical core, we see no change in their differences depending on victim activity. We thus conclude that both the machine clear event and the microcode assist occur only in the victim’s physical core, causing the attacker’s core to stall until the conflicting cache line is safely evicted.

From a privileged attacker’s perspective, these PMC events may serve as an alternative to the TSC when constructing a more reliable access oracle. In contrast to the TSC, checking whether the events are listed in Table 2 increment does not require prior calibration. In addition, they are unaffected, e.g., by dynamic frequency scaling, which can skew TSC-based measurements. We refer to this variant of Load+Probe as PMC-based Load+Probe.

Takeaway (3): Load+Probe causes both a machine clear and a microcode assist, leading to cache-line invalidation. Attackers can use PMCs as a noise-free alternative to the TSC when probing the victim’s cache state.

TSX-Probe. Intel TSX extends the x86_64 ISA with support for so-called *hardware transactions*, which ensure atom-

icity in a sequence of memory accesses in multithreaded applications. A TSX transaction, initialized by the `xbegin` instruction, defers all memory accesses until it successfully reaches the terminating `xend` instruction, and then *commits* the changes to memory. If a concurrent thread performs a conflicting memory access or the transaction encounters a fault, it *aborts*, reverting all program state changes and jumping to a fallback address. Disselkoe et al. [38] previously exploited this abort-on-conflict mechanism as a side channel to infer cache activity on Intel non-TDX CPUs.

On TDX-enabled CPUs, we observe that TSX transactions from the host treat reads to TDX private memory as regular memory accesses and successfully commit when there is no external interference. However, when loading the physical alias of an address that a TD previously cached in the core’s L1 cache, the transaction instead aborts. Notably, this occurs under the same conditions for which the PMCs indicate a microcode assist with Load+Probe. Hence, as with the PMCs in Table 2, untrusted code, such as the hypervisor, can use TSX transactions as an access oracle when executing on the same physical core as the victim. This is the case, for example, when running in the victim’s sibling hyperthread or in an interrupt handler. We refer to this primitive as *TSX-Probe*. An example of how to use TSX for probing the victim’s cache state is shown in Listing 1.

In practice, TSX-Probe can serve as a practical, low-noise alternative to Load+Probe. As with PMC-based Load+Probe, it is unaffected by error sources in TSC-based measurements. However, in contrast to the PMCs, it does not require knowledge of a CPU’s supported events, making it universally applicable to all TDX-enabled CPUs. Moreover, TSX does not require kernel privileges. Given an alias address mapped into a user-mode process in the untrusted host, this primitive thus enables complex attacks without requiring extensive kernel modifications.

Takeaway (4): TSX transactions involving physical aliases abort when a TD has L1-cached their addresses, thus providing a low-noise alternative to Load+Probe.

MWAIT-Probe. The privileged `mwait` instruction enables the OS to reduce power consumption by allowing the CPU core to enter an optimized power state, where execution is temporarily paused. This is conceptually similar to the `hlt` instruction, which pauses execution until an interrupt occurs. However, in contrast to `hlt`, `mwait` resumes execution when a different thread accesses a specific memory location. The OS can configure this location with the `monitor` instruction. However, although `monitor` takes a *virtual* address, previous work demonstrated that `mwait` monitors the *physical* memory location it corresponds to [17]. Specifically, any access to the same physical cache-line (i.e., bits 6 and above of the physical address match) allows `mwait` to resume execution, regardless of the virtual address. This allows it to observe accesses outside its virtual address space, such as in user mode.

We demonstrate that this also applies to memory accesses in TDX, thus confirming a theoretical concern origi-

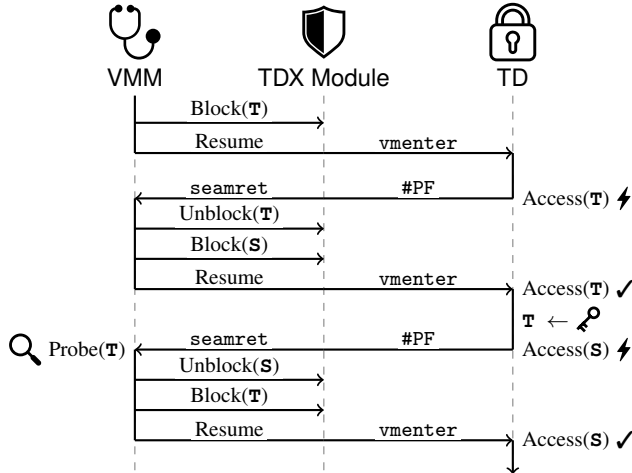


Figure 2. Monitoring a single target page T with TDXRay at cache-line granularity. After blocking T , we wait for the TD to access it and block the synchronization page S . The TD can then access T , but causes a page fault when accessing S . When handling this fault, the hypervisor probes the cache state to recover which cache lines in T were accessed.

nally raised by Aktas et al. without practical validation [35]. When configuring a physical alias address of a TDX private memory location with `monitor` and then pausing execution with `mwait` in the hypervisor, execution resumes when a TD accesses this private memory location. This is independent of the core the TD executes on, i.e., any core can execute `mwait` to observe the TD’s memory accesses.

Moreover, in addition to providing a memory access oracle, `mwait` can serve as a precise, cache-line granular synchronization primitive. For instance, if `mwait` is configured to monitor a specific code location in the TD, any code following the `mwait` instruction in the host is temporally aligned with the target code’s execution in the TD. This may allow attackers, e.g., to record perfectly aligned power usage [39] or execution port contention traces [29], [40], potentially revealing the TD’s execution path.

Takeaway (5): the `mwait` and `monitor` instructions allow any core to detect if a TD accesses a target physical memory location with cache-line granularity.

4. TDXRay Side-Channel Tracer

In this section, we present TDXRay, a generic tool for tracing the memory accesses of a TD in its guest-physical address space. TDXRay reliably monitors large memory regions, applicable to exploitation and reconnaissance, i.e., locating specific victim pages in a TD’s private memory.

4.1. Overview

TDXRay can trace a victim TD’s memory accesses at two spatial granularity levels: *page-level* granularity (i.e., 4 kB or 2 MB), and *cache-line* granularity (i.e., 64 B).

For page-level monitoring, it relies exclusively on the TDX module’s `block` and `unblock` APIs as a side channel. Given a set of target pages $T_0 \dots T_n$ to monitor, TDXRay initially blocks the entire set. Whenever the TD then accesses a target page T_i , TDXRay unblocks it, adds its GPA to the output trace, and resumes TD execution. When the TD accesses a different target page T_j , TDXRay blocks T_i again. As with T_i , it then adds the GPA T_j to the trace, unblocks it, and resumes TD execution in anticipation of the next page access. This cycle repeats for the duration of the monitoring period, resulting in a trace of page accesses. Optionally, the user may also configure TDXRay to leave a *trail* of pages unblocked, re-blocking a page only after a certain number of other pages have been accessed. This prevents stalls, e.g., when a single memory access crosses a page boundary between target pages.

For cache-line granularity, this process is augmented with a cache side channel from Section 3. TDXRay supports TSC and PMC-based Load+Probe and TSX-Probe, with TSX-Probe as the default. Before re-blocking a target page after the TD accessed it, we probe which of its 64 B cache lines are present in the cache, thereby revealing which of them the TD accessed. This happens in a fault handler, and thus on the same core that previously accessed the target page. We can thus probe the victim’s L1 cache state without requiring hyperthreading. However, while we demonstrate that this approach is accurate, it bears the risk of losing information, as the target page’s cache lines may be evicted if sufficient time passes until the next target page is accessed. Hence, each target page can optionally be assigned a *synchronization page*, which is not part of the target set, but triggers cache probing of the target page when accessed. Typically, this page is chosen such that the TD accesses it shortly after the target page, initiating the probing earlier than would be the case when waiting for a different target page. Since this reduces the time between the victim access and probing, the cache state is better preserved, improving accuracy. Moreover, it allows TDXRay to monitor accesses to a single target page at cache-line granularity, which is not possible without a synchronization page, as there would be no second target page to trigger probing. For a schematic overview of the steps involved with cache-line granularity monitoring, see Figure 2.

4.2. Implementation

We implement TDXRay as a Linux kernel module, eliminating the need for direct kernel patches. This module uses the Linux kernel’s `kprobe` mechanism to hook the existing TD exit handler function, allowing it to handle page faults when a TD accesses a blocked page. TDXRay does not interfere with other types of TD exit(s), maintaining correct guest execution. As TDXRay runs in kernel mode, it has direct access to the TDX module’s `seamcall` interface, allowing it to block and unblock arbitrary pages and resolve GPAs to their corresponding HPAs. Moreover, it can change the mapping level of a specific victim page. For instance, if a specific guest page is mapped as a 2 MB page in the

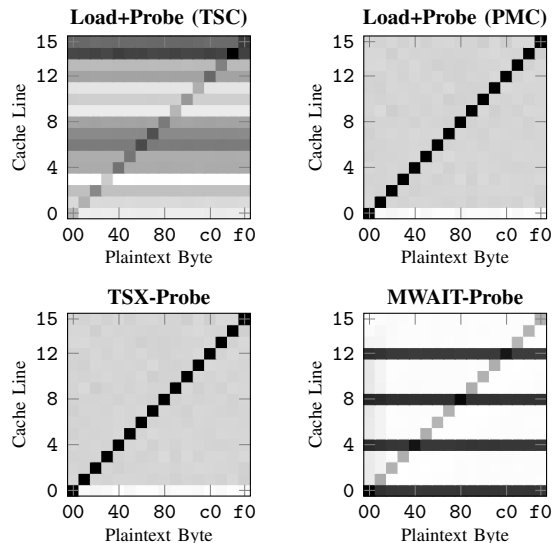


Figure 3. Heatmaps of the cache patterns recovered in the AES T-Table attack with various cache-granular access oracles. Note the characteristic diagonal line. The row maximum in each matrix is always clearly discernible, allowing the attacker to infer information about the first key byte.

TDX module’s SEPT structure, TDXRay can demote (split) it into a range of 4kB pages. Promoting a range of 4kB pages, i.e., merging them into a single 2MB page, is also possible, giving TDXRay complete control over the spatial granularity of SEPTTrace. Finally, TDXRay exposes all of its TDX-related functionalities to user-mode applications, including unrestricted `seamcall` access and full support for the raw side-channel primitives discussed in Section 3. This enables fast prototyping of complex attack chains.

4.3. Evaluation with AES Benchmark

We evaluate and compare TDXRay’s cache side channel primitives on OpenSSL’s T-Table-based AES implementation. This implementation is known to be affected by side channels and disabled by default, but it is used as a standard benchmark to evaluate cache side channels in prior work [14], [19], [38], [41]–[49]. We mount a first-round chosen plaintext attack as initially described by Osvik et al. [50], partially recovering the first key byte.

Setup. We run the attack with a single-core TD on an Intel Xeon 6736P (Granite Rapids) and TDX Module v2.0. The attacker runs in the untrusted host OS, whereas the victim runs OpenSSL v3.6.0 in the TD. While the original attack relies on shared memory between the victim and attacker, we adapt it to probe physical alias addresses of the Te_0 table instead. We assume that the physical address of Te_0 is already known to the attacker or has been previously identified using a method such as that described by Rauscher et al. [19]. The physical page containing Te_0 is our only target page with TDXRay, and we use the code page storing the call to `AES_encrypt` as its synchronization page. We configure TDXRay to use three different access oracles for

revealing the victim’s cache state: Load+Probe with the TSC for differentiating cache hits and misses, Load+Probe with the `ASSISTS.HARDWARE` PMC, and TSX-Probe. Additionally, since TDXRay does not support MWAIT-probe, we run a separate attack using MWAIT-probe to compare its performance with the other access oracles. Here, instead of probing whether an address is cached, we count the number of times MWAIT wakes up for a given cache-line while encrypting the chosen plaintext. In practice, we do this in parallel for all 16 cache lines monitored in the attack, using 16 physical CPU cores with hyperthreading disabled.

Results. We execute the T-Table attack 1000 times per configuration. Figure 3 shows the access patterns we observe, all exhibiting the characteristic diagonal line expected when the first key byte is 0. Although all access oracles can reliably produce this result, they differ in their susceptibility to noise. For instance, when using the TSC with Load+Probe, the success rate is 99.4% for monitoring 2000 encryptions per chosen plaintext byte, with an average time of 1.84s. In comparison, TSX-Probe and PMC-based Load+Probe with `ASSISTS.HARDWARE` are significantly more robust, with both achieving a success rate of 100% after only 100 encryptions per plaintext byte, and only 89ms and 170ms respectively. This is expected, as TSC measurements are often skewed by artifacts, e.g., from dynamic frequency scaling. In contrast, TSX-Probe and `ASSISTS.HARDWARE` only trigger with a Load+Probe-specific cache conflict, revealing whether a conflicting cache line is cached. With the same chosen plain texts, even the noise patterns in cells outside the typical diagonal in Figure 3 are virtually identical, indicating near-perfect leakage.

Finally, when using MWAIT-probe, we achieve a 97.7% success rate after observing 5000 encryptions per plaintext byte, taking 239ms on average. We also observe that `mwait` wakes up significantly more often for cache lines with indexes divisible by 4. These results indicate that MWAIT-probe is less robust than the other oracles, which is expected given that `mwait` is also influenced by unrelated interrupts. However, it has the advantage of being stealthier: unlike TDXRay, this attack does not incur page faults, causing a comparatively low runtime per encryption.

Overall, we demonstrate that all of our cache side-channel primitives work reliably in a practical scenario. In particular, TDXRay with PMC-based Load+Probe and TSX-probe provides exceptionally accurate results.

5. Attacking Private Inference with TDXRay

We demonstrate an end-to-end attack against a VM workload performing private LLM inference running in TDX. By exploiting side-channel information collected with TDXRay, a malicious hypervisor can steal user prompts.

5.1. Leakage during Tokenization

Large Language Models (LLMs) operate on discrete token IDs rather than raw text. Hence, when an LLM receives a prompt, the raw character stream is first converted into

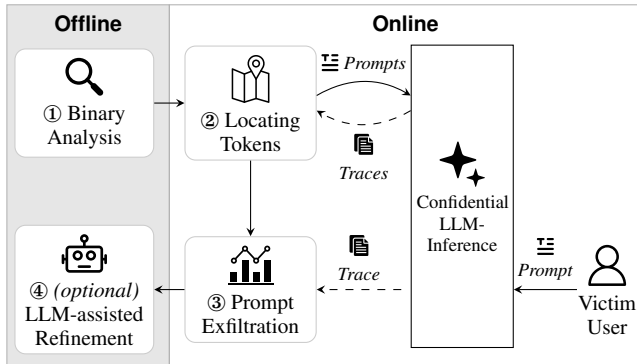


Figure 4. An overview of our end-to-end attack. The attacker ① analyzes the victim binary’s data layout and ② locates the token nodes in private memory by recording access traces for crafted prompts. They can then ③ exfiltrate user prompts and, optionally, ④ refine the leakage results with an external LLM.

a discrete sequence of token IDs that the model actually consumes. That conversion, called *tokenization*, segments the input text (prompt) into substrings (token strings) and maps each substring to a unique token ID from a fixed vocabulary. Common tokenizers implement this mapping with algorithms such as byte-pair encoding (BPE) [51], [52], unigram/wordpiece models [53], or longest-match greedy variants [54]–[56]; these algorithms differ in how they choose substring boundaries, but they all rely on a runtime lookup over an in-memory vocabulary to resolve candidate substrings (tokens) to a numeric token ID. This lookup usually involves a hash map, which results in a predictable, data-dependent memory access pattern whenever the tokenizer resolves a token, as illustrated in Figure 5. A hash function first decomposes a substring into a *bucket ID*, which serves as an index for an array of linked lists. The tokenizer then traverses the bucket’s list, either finding the target element or reaching the end of the list. Given the memory locations of bucket nodes and a view of the tokenizer’s memory accesses, attackers can recover a substring by determining which nodes the tokenizer accessed. Doing this repeatedly, leaking all substrings of a prompt in order, enables reconstructing the entire prompt.

Because this attack targets tokenization, which typically occurs exclusively in the CPU, it requires no shared CPU/GPU memory or access to GPU internals. The LLM’s inference could occur in a GPU, and our attack would remain successful. Moreover, an attacker only requires knowledge of the model’s vocabulary, rather than its trained weights or an identical copy of the model itself. Because custom confidential models are frequently fine-tuned from established public foundation models, their tokenization dictionaries typically remain standard and publicly known. In such cases, the attack remains fully effective even if the model weights are encrypted. Conversely, if a victim employs a completely private, custom-built vocabulary from scratch, the attack as presented would not directly apply; however, such scenarios are rare in practice and fall outside

our considered threat model.

To successfully launch this attack on private inference, there are two hurdles to overcome. *First*, an attacker must find the exact location of each bucket node in the victim’s confidential memory. In Section 5.2, we describe a generic and reliable method to achieve this: analyze the victim binary to obtain the hash map layout in virtual memory, then send a series of malicious prompts to the TD, tracing its memory accesses to identify this layout in its physical address space. This is a one-time effort, as the hash map is static and its layout typically remains the same until the victim process is restarted. The *second* hurdle comes from each victim user: the attacker must record noise-free memory access traces covering all list nodes in the target hash map whenever that victim sends a prompt. Section 5.3 details how we achieve this by configuring TDXRay to monitor the corresponding victim pages with a single, shared synchronization page that is accessed after every lookup. Since this process also captures artifacts from the tokenization algorithm, the attacker can optionally refine the output of this step with an external LLM. Refer to Figure 4 for a schematic of our approach.

We extensively evaluate our attack for `llama.cpp`, an LLM inference library for quantized versions of open-source models such as Meta’s Llama and Google’s Gemma family of LLMs. In Section 5.4, we show that after the initial setup, it can leak prompts by observing just a single tokenization for both Llama 3.2 [57] and Gemma 3 [58].

5.2. Locating Tokens in Victim’s Private Memory

As our attack aims to trace accesses to the nodes in a hash map’s linked lists (cf. Figure 5), the first step is to locate these nodes (token nodes) in a victim’s private memory. In `llama.cpp`’s virtual address space, token nodes are typically arranged contiguously in the program’s virtual address space with respect to their numeric IDs. However, after translation to guest physical addresses (GPAs), those contiguous virtual ranges are scattered across noncontiguous GPAs. In practice, the allocator and MMU assign pages from a large physical pool, so while the offset of a token within its 4kB page remains stable, the page-level placement of that token in GPA space is effectively randomized. Hence, our attack can make no assumptions on whether the target pages are contiguous in the GPA space.

An important observation is that, since the hash map’s layout in `llama.cpp`’s virtual memory is always the same for a given vocabulary, the offsets of tokens in a 4kB page are preserved. Hence, identifying which physical pages contain a specific token immediately reveals all other tokens co-resident on those same pages. While modern LLMs operate over dictionaries with hundreds of thousands of tokens, the attacker just needs to find the physical 4kB page of a single “representative” token in each 4kB page to obtain all physical addresses. This reduces the task of finding addresses for hundreds of thousands of tokens to just a few thousand ($\approx 128K$ to $\approx 2K$ for `llama.cpp`).

Our recovery pipeline proceeds in three phases.

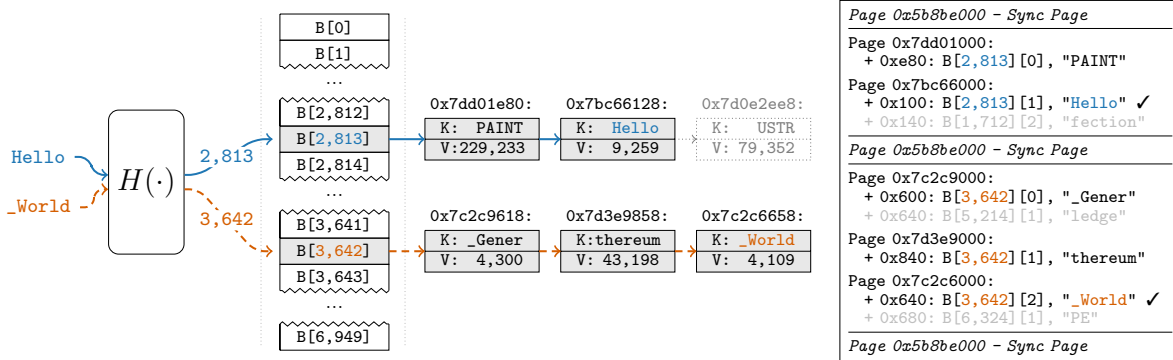


Figure 5. Mapping the numeric tokens for the strings `Hello` and `_World` in a hash map. The hash function first maps the string to a numeric bucket ID, which the program uses to index an array of linked lists. It then traverses the list’s nodes until it finds the correct key/value pair. By monitoring these nodes, we obtain the trace shown on the right, which can discern which bucket it traverses and where it stops.

- 1) **Binary analysis:** In an offline phase we analyze the binary to identify a representative token for each 4 kB page containing tokens.
- 2) **Coarse localization:** We trace page-fault activity at a 2 MB granularity to identify regions correlated with attacker-controlled prompts. This step filters the physical address space to a small subset of megabyte-scale regions that likely contain tokenizer tables. Using *demote* API, we split the remaining 2 MB to 4 kB pages.
- 3) **Pruning:** For each representative token t , we proceed as follows. We select a representative token $t' \neq t$, and define three prompts p_1, p_2, p_3 consisting of t repeated n times, t' repeated $2n$ times, and t' repeated n times, respectively, for a small $n \approx 10$. We then compare the observed page accesses (at 4 kB granularity) across these prompts: pages whose access counts approximately double from p_1 to p_2 , while remaining inactive during p_3 , are identified as candidates to contain the node of a target token t . Finding the correct page among these small set of candidates (typically 3 in our evaluation) can be easily done based on their relative order. We need to run 3 prompts for each of the 2K representative tokens, and one of them, namely p_3 , can be used twice, making the attack require 4k prompts.

Locating the Sync Page. We can reliably identify the shared synchronization page (the *Sync Page* in Figure 5) based on its distinctive occurrence pattern in the page fault traces of phase 3. Concretely, after determining the GPA corresponding to token t , we examine all occurrences of this token in the trace of prompt p_1 and collect the GPAs that fault in the subsequent few entries (typically the next ten faults) after each occurrence. By intersecting these sets and removing any GPAs corresponding to known token pages, we obtain a small set of candidate addresses (typically < 5) that consistently appear after token accesses. In practice, any of these candidates can serve as the *Sync Page*.

Localizing the token nodes is a one-time effort. For `llama.cpp`, we confirm that after initialization, the layout of token nodes both in virtual memory and the GPA space remains static until the victim process terminates. After this

initial step, an attacker can thus record traces for arbitrarily many tokenization attempts.

5.3. Leaking the User Prompt

Once we have located the token nodes and a suitable synchronization page in the TD’s private memory, we configure TDXRay to monitor them at cache-line granularity. When the victim tokenizes a prompt, this results in a memory access trace as shown in Figure 5. The synchronization page is accessed after every lookup, indicating where a bucket list traversal starts and ends. We call the memory accesses between two synchronization page accesses, i.e., comprising a single traversal, a *frame*. Due to TDXRay’s recording method (see Section 4), each frame consists of a series of page accesses, and for each page access, a list of cache lines that the victim touched. In practice, the nodes of a single bucket list are typically scattered across different pages, meaning that each page access in a frame corresponds to a single accessed node.

Since traversal ends when the victim finds the correct target node, we expect the last page access in a frame to cache the node of a token from the original prompt. However, this alone is insufficient to uniquely identify the token, as the accessed cache lines may cover two or more candidate nodes. In such cases, we rely on a series of heuristics to leak a single token rather than multiple candidates. For instance, since the hash function is public, an attacker can determine each node’s bucket ID. With frames of more than two pages, the bucket being traversed can thus be determined. By checking which candidate belongs to this bucket, we can confidently identify the correct token. For single-page frames, we choose the candidate node with the lowest page offset. While this provides a lower confidence in the predicted token, we find that this heuristic holds in the majority of cases.

Finally, there is a risk of leaking incorrect tokens if the lookup does not find the requested token string. In this case, the bucket list corresponding to the string’s hash is still traversed, but does not yield a valid result. As the

Please reverse 'dlrow olleH'

```
us|se|er|ser|user|user|Pl|le|ea|as|se|e|r|re|ev|ve|er|
rs|se|e|'|d|dl|r|ro|ow|W|o|ol|ll|le|e|H'|ver|ers|re
|rev|W|o|ol|Pl|le|le|le|H|le|as|ase|lro|row|e|re|rev
erse|se|rever|verse|se|'|d|le|ase|ase|re|Please|le|ase
re|reverse|verse|'|W|ol|olle|Please|re|Please|reverse|
reverse|'|d|dlrow|Please|reverse|'|dl|row|ol|le|H'|
|mo|od|de|el|del|mod|odel|model|model
```

Figure 6. All substrings that `llama.cpp` looks up when tokenizing a test prompt for Gemma 3, separated by vertical lines. Strings yielding a valid token are underlined. Note the highlighted passage, which contains the prompt verbatim.

traversal ends with the last node, we incorrectly leak its key. However, in practice, this is rarely an issue, as the attacker can still leak a superset of the prompt’s tokens, thus recovering all information the user provided. Moreover, most BPE implementations [59]–[61] decompose large chunks (e.g., a sentence or a line) of the prompt into a sequence of valid substrings before resolving the numeric IDs of these substrings in order. Hence, there are large contiguous sequences of lookups in which the victim resolves only valid tokens while preserving the original prompt’s order. Figure 6 depicts an example: the substrings looked up by `llama.cpp` when tokenizing a test prompt for Gemma 3 [58]. This tokenizer processes the prompt line-by-line, with a new chunk starting after every line break. Hence, with the 1-line prompt in the example, there is a sequence of valid lookups that make up the entire prompt when combined. Even with multi-line prompts, the output of our attack contains the entire prompt in contiguous sections of human-readable text, trivializing to infer confidential information from its output. Moreover, we show that general-purpose LLMs can reliably isolate these sections from the attack’s output when instructed to do so, thus making it possible to leak the prompt verbatim.

5.4. Evaluation

We evaluate the practicality of our attack using the `llama.cpp` library with a dedicated victim program that accepts prompts via a network connection. Unless otherwise stated, all experiments are performed on an Intel Xeon 6736P (Granite Rapids) with TDX Module v2.0. The victim runs in a single-core TD with 8 GB of RAM and Ubuntu 24.04. Both inference and tokenization execute entirely in private memory with a CPU-based inference backend.

Locating Tokens. In our (unoptimized) proof of concept for locating the GPA of every token node, we focused on `llama.cpp` with Llama 3.2. Here, 128 256 tokens are spread over 2175 4kB pages, meaning that we have to identify 2175 representative tokens. Our results show that we can achieve this with near-perfect (100%) accuracy, taking approximately 10s on average per 4kB page. This is an overhead of $\approx 10\times$ over inference without monitoring. Overall, it takes roughly six hours to identify the GPAs

for all tokens. In Appendix B.2, we present more efficient techniques that require fewer prompts.

General Accuracy. To evaluate the accuracy of the recovered prompts, we use the prompt exfiltration stage of our attack to recover 500 single-sentence questions randomly sampled from the Stanford Question Answering dataset [62], [63]. Our goal is to leak the entire prompt in a single shot, i.e., our victim processes each prompt only once. After recovering the accessed nodes as described in Section 5.3, we merge the substrings of the leaked nodes into a single string and instruct GPT5-nano [64] to isolate only the original prompt. Hence, we recover a character string rather than just a sequence of accessed nodes, enabling a direct comparison with the original prompt. We attack two separate models: Llama 3.2 (1B), with a dictionary of 128 256 tokens, and Gemma 3 (1B) with a dictionary of 262 144 tokens. Furthermore, we assume that the token GPAs are already known, i.e., the localization stage of the attack was successful.

We assign each sample a *similarity score* based on the length-normalized edit distance:

$$1 - \frac{d_{edit}(P, P')}{\max(|P|, |P'|)}$$

where P is the sequence of characters making up the original prompt, P' contains characters we leak, and d_{edit} denotes the Levenshtein edit distance. This measures how closely the recovered prompt matches the original, with 100% indicating a perfect match.

For Llama 3.2, we achieve an average similarity score of 91.5% in this experiment, with 287 of 500 prompts being recovered perfectly. The scores for Gemma 3 are even higher, with an average similarity score of 94.2% and 316 perfect recoveries. Similarity scores of less than 50%, indicating that more than half of the recovered text is missing or corrupted, occur in only 26 samples for Llama 3.2 and 14 for Gemma 3. This means that even when some tokens are corrupted, it usually affects only short passages or single words, potentially still allowing an attacker to infer the prompt’s meaning. Additionally, even when profiled with TDXRay, tokenizing the prompts in our test set takes only 62.1 ms on average with Llama 3.2, and 346 ms with Gemma 3. To a human user, the attack is thus barely perceptible.

Case Study: Stealing Identities. In addition to assessing the attack’s overall accuracy, we evaluate its effectiveness for stealing individual secrets within a larger prompt, such as user credentials and identities. For this, we use the *Faker* Python library [65] to generate 500 fictitious user profiles. These profiles contain a user’s full name, home address, email address, US Social Security number, and credit card details with expiry dates and CVCs. See Appendix A for an example of such a profile. We then provide these profiles as an input prompt to our victim program while using the second stage of our attack to recover them. As in the previous experiment, we assume the localization stage has already succeeded and leak the prompt in a single shot. However,

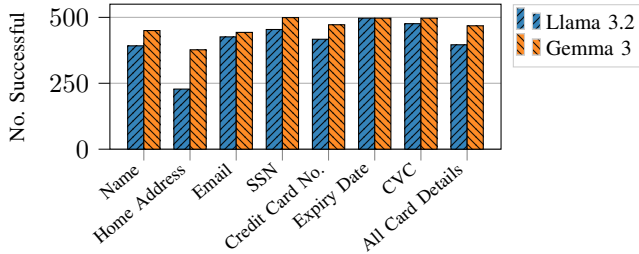


Figure 7. The number of samples for which our attack successfully recovers a specific secret verbatim ($n = 500$). Additionally, we count instances for the *All Card Details* bar if their card number, expiry date, and CVC are all disclosed correctly in the same sample.

since our goal is only to leak individual data points, we forego prompt isolation from the attack’s output with an LLM. We consider the attack a success if a specific data point, such as the user’s credit card number, is contained verbatim in the attack’s raw output.

The results for Llama 3.2 and Gemma 3 are shown in Figure 7. While the success rates for Gemma 3 generally exceed those for Llama 3.2, we observe high success rates for both. For instance, a user’s full credit card details, including card number, expiry date, and CVC, were successfully disclosed in 79.2% of samples with Llama 3.2 and 93.6% with Gemma 3. We recover the US Social Security number in 90.8% for Llama 3.2, and 99.8% for Gemma 3. However, the success rate for longer sequences, such as the home address, is lower than for shorter sequences, such as the credit card expiry date. This is expected, as longer sequences are more likely to have single tokens leak incorrectly, e.g., due to noise. However, as in the previous experiment, we observe that such defects are usually minor. For example, in the home address field for Llama 3, we observe that periods in abbreviations are frequently misinterpreted as slashes (e.g., *Rosewood Ave.* would become *ROSEWOOD Ave/*). Although the metric does not account for this, such defects would not impair real-world attackers. As such, we interpret the success rates in Figure 7 as lower bounds.

While this attack focuses on stealing user prompts, the ideas could be easily extended/adapted to stealing LLM responses. This variant is briefly discussed in Appendix C.

6. Mitigations

We begin by proposing and empirically validating a software-based defense for tokenization. Following this, we discuss theoretical system-level and architectural mitigations. While an empirical evaluation of these hardware and OS-level changes is outside the scope of this paper—largely because they require modifying existing Intel TDX silicon or incur prohibitive system-wide overheads—we outline them as critical directions for future secure processor design.

TABLE 3. FRACTION OF WIKITEXT TOKENS MISSING IN THE DATA-OBVIOUS MAP’S VOCABULARY AFTER REDUCING THE MAXIMUM KEY SIZE

Size (bytes)	64	32	16	8
Missed Tokens	0.0002 %	0.0018 %	0.045 %	10.06 %

6.1. Tokenization-specific Mitigations

Prompt leakage during tokenization is caused by secret-dependent hash map lookups. The attack we identify relies on a deterministic assignment of indexes. If the hash function were instead randomized and secret, reproducing the attack reduces to recovering that random secret. This is not a complete solution, however, as other attacks that do not rely on knowing the complete list of assignments may exist.

A more robust mitigation is to use a *data-oblivious* map, which guarantees that different lookups produce indistinguishable access patterns. Such data structures are used in privacy-relevant apps such as the *Signal* contact discovery [66], and performant solutions exist [67]–[70].

To evaluate the impact of this approach for LLM tokenization, we store our entire token-to-ID lookup table—over 128K tokens for the Llama 3.2 vocabulary—in a data-oblivious map.¹ This data structure requires fixed-size keys: Smaller keys reduce memory overhead and lead to a more performant map, but they truncate large tokens, limiting the vocabulary. For Llama 3.2, the largest token is 128 bytes, and limiting the key size to 64 or 32 bytes would exclude 346 and 3,976 tokens, respectively. To assess practical impact, we counted affected tokens in the complete WikiText-103 dataset (>100M tokens). Results are shown in Table 3.

To empirically evaluate its runtime overhead, we compare the oblivious data structure with `std::unordered_map` from the C++ standard library on a randomly sampled subset of 100,000 WikiText-103 tokens, varying key size from 8 to 128 bytes. We evaluate on an isolated performance core of an Intel i9-12900K CPU and disable frequency scaling and ASLR to reduce variance. Our benchmark is repeated 100 times to ensure stable results (max stdev. non-oblivious $\pm 0.000460\mu s$, oblivious $\pm 0.076438\mu s$). Figure 8 reports the results:

We observe that the overhead of obliviousness ranges from 52.52x to 369.15x. However, the overall lookup time per token is still on the order of microseconds, and hence the LLM inference time dominates the overall running time for small queries [71]. For large queries, we believe more efficient oblivious tokenization can be achieved based on Private Set Intersection [72]. We leave a detailed exploration of oblivious tokenizers to future work.

Interestingly, we observe a higher relative overhead with smaller key sizes. This is because with smaller key sizes, more oblivious map lookups and merges are required. Future work could revisit the benefit of smaller token sizes for different oblivious map implementations.

1. We use the implementation from Oblivious Labs [69].

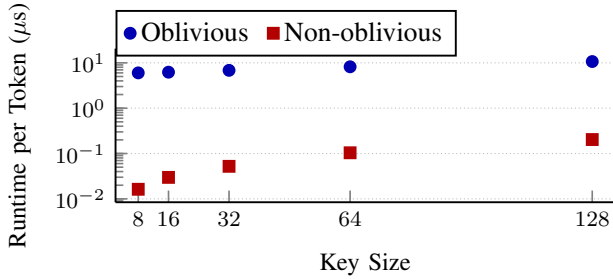


Figure 8. Benchmark results for oblivious and non-oblivious token-to-ID mapping with different key sizes. Speed is in tokens per us (lower is better).

6.2. System-level Mitigation

Next, we propose mitigations at the system level:

Detecting Microarchitectural Footprints. Although TDXRay uses less invasive methods than single-stepping, it still leaves observable microarchitectural footprints, such as inflated page faults and cache misses.

For some workloads, an effective countermeasure could be anomaly detection. One could track the confidential workload’s performance metrics and flag when they deviate significantly from a benign baseline, as done in prior work [73]. Modern CVMs make this viable by supporting virtualization of performance counters: a monitor within the guest can collect accurate microarchitectural statistics, enabling the detection system to operate in the TCB.

Limiting Page Table Modification. Intel TDX introduces host-callable page table modification APIs to facilitate complex guest life-cycle management features like dynamic page swapping and live migration. Intel TDX currently lacks a mechanism to disable or restrict access to these host-side APIs, even though many static CVM workloads do not require these capabilities; such a mechanism would negate our SEPTrace primitive.

Future architectures can also introduce a special set of memory pages whose state (mapping status, permissions) remains immutable throughout the CVM’s lifetime. A corresponding alternative API or guest configuration option would be necessary to enforce this immutability, effectively removing the block/unblock capability for critical memory regions.

Cache-Line Isolation. TDX’s current cache coherence protocol does not allow cache lines with matching physical addresses but different HKIDs to be cached at the same time, which enables Load+Probe and TSX-Probe: if two such lines are in the cache hierarchy, one is evicted, creating observable timing differences.

The obvious mitigation is to incorporate the HKID (or a derivative) into cache tag lookups and management, allowing cache lines with the same physical address but different HKIDs to be cached concurrently.

As an alternative, access attempts by an untrusted entity to a TDX private physical address could be forced to reliably

behave as cache misses or trigger a hardware fault, thereby preventing direct measurement of the cache-line’s state.

7. Related Work

Controlled-channel page fault attacks. Previous work [20], [34], [74] has shown that an untrusted OS can extract secrets from Intel SGX enclaves by inducing page faults and tracking memory page activities, forming a deterministic rather than a noisy timing signal.

In VM-based TEEs, the hypervisor plays the role of the untrusted OS. With AMD SEV/SEV-ES, guest memory is encrypted, but the hypervisor still controls nested-page table (NPT) permissions and can log guest-physical page accesses as faults, enabling controlled-channel tracing [30], [31]. SEV-SNP adds integrity (RMP) to block page-remapping attacks, but AMD explicitly excludes hypervisor monitoring of page accesses and page-fault rates, so access-pattern channels remain. In concurrent work, SNPeek [75] and T-Time [76] explore these vulnerabilities on AMD SEV-SNP and Intel TDX, respectively. Intel TDX similarly encrypts guest memory and state while retaining hypervisor control over nested translations. However, TDX imposes unique isolation barriers and architectural constraints that render existing SGX and SEV-ES attack methodologies ineffective. Unlike prior systems, TDXRay is specifically designed to address these TDX-specific defenses, unifying the necessary, novel primitives into a coherent exploitation framework. We also note that ARM CCA, another emerging VM-based TEE, is currently in an early phase without publicly available hardware, precluding concrete empirical evaluation of similar vectors at this time. Thus, while TDX/SEV-SNP close remapping and state-exfiltration classes, page-granularity access-pattern leakage through controlled faults (or page-bit variants) persists as an open issue in many threat models.

Side-Channel Attacks on Intel TDX.

Google’s initial security review of Intel TDX [35] identified several potential side channels. They noted that Block/Unblock APIs could serve as an alternative to traditional page-fault-based attacks, flush-reload variants exploiting KeyID aliasing, and the use of `mwait` to monitor the dirty state of a TD’s cache lines.

However, because TDX hardware was unavailable at the time of their report, [35] broadly posits the existence of these side-channels but lacks concrete evaluation, working primitives, or target-specific adaptations. In contrast, our work demonstrates practical exploitability on real hardware by reverse-engineering the specific, undocumented microarchitectural behaviors required for reliable channels.

They also discussed single-step techniques, prevalent in attacks against Intel SGX, noting that Intel’s hardware mitigation allows a random number of TD’s instructions to progress per TD-EXIT. Wilke et al. [77] bypassed this mitigation by lowering the CPU frequency to defeat single-step detection. In response, Intel replaced the timing-based check with the Instruction-Count Single-Step Defense (IC-SSD), which uses performance counters to verify how many

instructions the TD executed. Rauscher et al. [19] subsequently showed that the random number generator (RNG) that controls this instruction progress shares a per-core state. An attacker can profile this shared state from other attacker-controlled TDs to schedule a single-step attack against a victim TD on the same core, although each step incurs roughly 3.7ms latency. The cumulative latency creates a pronounced timing signature, making the analysis easily detectable with execution timeouts (e.g., the timeout limit for WASM sandbox in Google CVMs workloads [78]). Given these limitations, and that Intel has announced plans to fix the issue [19], we omit single-step from our analysis.

LLM Side Channels. Prior work has studied side channels on machine-learning models. DeepCache [79] showed how a neural-net’s architecture in the cloud can be reverse-engineered by exploiting cache-aware optimizations. The recent work by Adiletta & Sunar aims at extracting LLM tokens [80]. Their work does not touch on TEEs or use paging as an attack vector, instead relying on co-located processes. Gao et al. [81] also focus on cache attacks and using, in their words, a co-located “spy application”. Their approach differs by exploiting token embeddings and the auto-regressive nature of language models, whereas we target tokenization. Carlini & Nasr also explore how to extract content of LLM conversations, but they opt for measuring the time that the model takes to respond to a query [82]; they write that, “under practical assumptions, [...] a passive network adversary can learn limited information (e.g., the topic of conversation).”

8. Conclusion

In this paper, we systematically analyze the side-channel attack surface of Intel TDX, identifying four new primitives that reveal victim memory accesses. SEPTTrace exploits the TDX Module’s Block/Unblock APIs to obtain a page-level access trace, while Load+Probe, TSX-Probe, and MWAIT-Probe reveal the victim’s cache state. By combining these primitives, we present TDXRay, a novel framework to generate highly accurate cache-line-granular memory access traces of unmodified confidential VMs. We demonstrate the practical impact of this leakage through a powerful attack on LLM inference, in which we reliably recover sensitive user prompts by observing memory access patterns during tokenization. Our findings underscore that, despite advanced architectural isolation, critical microarchitectural vulnerabilities persist in state-of-the-art confidential computing systems, highlighting the need for stronger defenses.

Ethics Considerations

Although all experiments rely only on publicly documented hardware interfaces and fully comply with the TDX security model, we followed responsible disclosure and notified Intel as well as major vendors that publicly rely on confidential computing for private inference—including Apple, Anthropic, Google, Microsoft, Meta, and OpenAI—in November 2025.

Our experiments were performed in controlled lab settings, without involving production systems, user data, or human subjects. The attacks are evaluated solely to inform the community and vendors about security and privacy risks and improve defense against this class of attacks.

We have released the source code for TDXRay and our evaluation artifacts to support reproducibility and on-going assessment of confidential computing platforms. The repository is publicly available at: <https://github.com/hoseinyavarzadeh/tdxray>.

LLM Usage Considerations

Our paper extensively uses Large Language Models (LLMs) as a research subject, proposing a method to extract user prompts from side-channel leakage in Confidential Virtual Machines and utilizing LLMs in the attack pipeline. LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

References

- [1] AMD, “Strengthening vm isolation with integrity protection and more,” *White Paper*, 2020.
- [2] Intel Corporation, “Intel® tdx module: Base architecture specification (v1.5),” Technical Report, 2023. [Online]. Available: <https://cdrdv2-public.intel.com/733575/intel-tdx-module-1-5-base-spec-348549002.pdf>
- [3] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell, “Design and verification of the arm confidential compute architecture,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.
- [4] google.com, “Bringing transparency to confidential computing with slsa,” <https://security.googleblog.com/2023/06/bringing-transparency-to-confidential.html>.
- [5] Y. Yan, C. Wei, X. Guo, X. Lu, X. Zheng, Q. Liu, C. Zhou, X. Song, B. Zhao, H. Zhang *et al.*, “Confidentiality support over financial grade consortium blockchain,” in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 2227–2240.
- [6] R. Li, Q. Wang, Q. Wang, D. Galindo, and M. Ryan, “Sok: Tee-assisted confidential smart contract,” *Proceedings on Privacy Enhancing Technologies*, vol. 2022, no. 3, pp. 711–731, 2022. [Online]. Available: <https://doi.org/10.56553/popets-2022-0093>
- [7] G. Developers, “Enabling more private generative ai,” 2025. [Online]. Available: <https://developers.googleblog.com/en/enabling-more-private-gen-ai/>
- [8] M. Engineering, “Whatsapp private processing for ai tools,” 2025. [Online]. Available: <https://engineering.fb.com/2025/04/29/security/whatsapp-private-processing-ai-tools/>
- [9] Anthropic, “Confidential inference via trusted virtual machines,” 2025. [Online]. Available: <https://www.anthropic.com/research/confidential-inference-trusted-vm>
- [10] 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, pp. 605–622.
- [11] G. Irazoqui, T. Eisenbarth, and B. Sunar, “Cross processor cache attacks,” in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, 2016, pp. 353–364.

- [12] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Tbleed: When protecting your cpu caches is not enough," *Black Hat*, 2018.
- [13] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>
- [14] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, p. 279–299. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_14
- [15] D. Evtushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *SIGPLAN Not.*, vol. 53, no. 2, p. 693–707, Mar. 2018. [Online]. Available: <https://doi.org/10.1145/3296957.3173204>
- [16] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 955–972. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/gras>
- [17] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels," in *USENIX Security*, 2023.
- [18] Intel, "Guidelines for mitigating timing side channels against cryptographic implementations," <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html>, 2022.
- [19] F. Rauscher, L. Wilke, H. Weissteiner, T. Eisenbarth, and D. Gruss, "Tdxploit: Novel techniques for single-stepping and cache attacks on intel tdx," in *USENIX Security 2025*, 2025.
- [20] 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, pp. 640–656.
- [21] OpenAI, "Reimagining secure infrastructure for advanced ai," 2025. [Online]. Available: <https://openai.com/index/reimagining-secure-infrastructure-for-advanced-ai/>
- [22] Apple, "Private Cloud Compute: A new frontier for AI privacy in the cloud," <https://security.apple.com/blog/private-cloud-compute/>.
- [23] Intel Corporation, "Intel software guard extensions (intel sgx)," <https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html>, 2025.
- [24] ARM Limited, "Building a secure system using trustzone technology," 2009, document Number PRD29-GENC-009492C.
- [25] Advanced Micro Devices, Inc., "AMD64 architecture programmer's manual volume 2: System programming," 2025, revision 3.43.
- [26] Intel Corporation, "White paper - intel trust domain extensions," 2022, document Number 343961-003US.
- [27] 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, pp. 337–351.
- [28] B. Schlüter, C. Wech, and S. Shinde, "Heracles: Chosen plaintext attack on amd sev-snp," in *ACM CCS*, 2025.
- [29] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Taveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 870–887.
- [30] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, "Severed: Subverting amd's virtual machine encryption," in *EuroSec*, 2018.
- [31] R. Zhang, A. Cheu, A. Gascon, D. Moghimi, P. Schoppmann, M. Schwarz, and O. Suciuc, "Farfetch'd: A side-channel analysis framework for privacy applications on confidential virtual machines," in *NDSS*, 2025.
- [32] M. Azure, "Azure confidential computing for ai," 2025. [Online]. Available: <https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-ai>
- [33] Intel, "Confidential ai – intel confidential computing zoo," 2025. [Online]. Available: https://github.com/intel/confidential-computing-zoo/tree/main/cczoo/confidential_ai
- [34] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschadler, 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, pp. 2421–2434.
- [35] E. Aktas, C. Cohen, J. Eads, J. Forshaw, and F. Wilhelm, "Intel trust domain extensions (tdx) security review," *Google security review*, 2023.
- [36] Intel Corporation, "Intel performance monitoring events," <https://perfmon-events.intel.com/>, accessed 2025-11-13.
- [37] H. H. J. Hum and J. R. Goodman, "Forward state for use in cache coherency in a multiprocessor system," U.S. Patent US6 922 756B2, 2002.
- [38] C. Disselkoe, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A Timer-FreeHigh-Precision l3 cache attack using intel TSX," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 51–67.
- [39] 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 *S&P*, 2021.
- [40] T. Hornetz and M. Schwarz, "Portprint: Identifying inaccessible code with port contention," *μASC*, 2025.
- [41] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious management unit: Why stopping cache attacks in software is harder than you think," in *USENIX Security*, 2018.
- [42] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks," in *USENIX Security*, 2020.
- [43] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of amd's cache way predictors," in *Asia CCS*, 2020.
- [44] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+ scope: Overcoming the observer effect for high-precision cache contention attacks," in *ACM CCS*, 2021.
- [45] L. Gerlach, D. Weber, R. Zhang, and M. Schwarz, "A security RISC: microarchitectural attacks on hardware RISC-V CPUs," in *IEEE S&P*, 2023.
- [46] A. Purnal, M. Bogner, F. Piessens, and I. Verbauwhede, "Showtime: Amplifying arbitrary cpu timing side channels," in *Asia CCS*, 2023.
- [47] L. Gerlach, S. Schwarz, N. Faröß, and M. Schwarz, "Efficient and generic microarchitectural hash-function recovery," in *IEEE S&P*, 2024.
- [48] D. Weber, L. Niemann, L. Gerlach, J. Reineke, and M. Schwarz, "No leakage without state change: Repurposing configurable cpu exceptions to prevent microarchitectural attacks," in *ACSAC*, 2024.
- [49] F. Thomas, D. Moghimi, M. Torres, and M. Schwarz, "Exfilstate: Automated discovery of timer-free cache side channels on arm cpus," *ACM CCS*, 2025.
- [50] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.
- [51] P. Gage, "A new algorithm for data compression," *The C Users Journal*, vol. 12, no. 2, pp. 23–38, 1994.

- [52] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. [Online]. Available: <https://doi.org/10.18653/v1/p16-1162>
- [53] T. Kudo, "Subword regularization: Improving neural network translation models with multiple subword candidates," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, I. Gurevych and Y. Miyao, Eds. Association for Computational Linguistics, 2018, pp. 66–75. [Online]. Available: <https://aclanthology.org/P18-1007/>
- [54] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. R. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. S. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *ArXiv*, vol. abs/1609.08144, 2016. [Online]. Available: <https://api.semanticscholar.org/CorpusID:3603249>
- [55] C. L. Jacobs and Y. Pinter, "Lost in space marking," *CoRR*, vol. abs/2208.01561, 2022. [Online]. Available: <https://doi.org/10.48550/arXiv.2208.01561>
- [56] V. Hofmann, H. Schuetze, and J. Pierrehumbert, "An embarrassingly simple method to mitigate undesirable properties of pretrained language model tokenizers," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 385–393. [Online]. Available: <https://aclanthology.org/2022.acl-short.43/>
- [57] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv e-prints*, pp. arXiv-2407, 2024.
- [58] A. Kamath, J. Ferret, S. Pathak, N. Vieillard, R. Merhej, S. Perrin, T. Matejovicova, A. Ramé, M. Rivière *et al.*, "Gemma 3 technical report," *arXiv preprint arXiv:2503.19786*, 2025.
- [59] Google, "sentencepiece," <https://github.com/google/sentencepiece>, 2025.
- [60] ggml, "llama.cpp," <https://github.com/ggml-org/llama.cpp>, 2025.
- [61] OpenAI, "Tiktoken," <https://github.com/openai/tiktoken>, 2025.
- [62] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016.
- [63] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for squad," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2018.
- [64] OpenAI, "GPT-5," <https://openai.com/gpt-5/>, accessed 2025-11-13.
- [65] "Faker v.37.11.0," <https://github.com/joke2k/faker>, 2025.
- [66] G. Connell, "Technology deep dive: Building a faster oram layer for enclaves," <https://signal.org/blog/building-faster-oram/>, 2022.
- [67] A. Tinoco, S. Gao, and E. Shi, "EnigMap: External-memory oblivious map for secure enclaves," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4033–4050.
- [68] L. Reichert, G. R. Chandran, P. Schoppmann, T. Schneider, and B. Scheuermann, "Menhir: An oblivious database with protection against access and volume pattern leakage," in *AsiaCCS*. ACM, 2024.
- [69] Oblivious Labs, "Oblivious map," <https://github.com/obliviouslabs/oram/tree/oblmap>, 2024.
- [70] S. Peters and K. Lewi, "oram," <https://github.com/facebook/oram>, 2024.
- [71] Phoronix Test Suite, "Llama.cpp benchmark," <https://openbenchmarking.org/test/pts/llama-cpp>, OpenBenchmarking.org, 2025, pts/llama-cpp benchmark results.
- [72] Y. Huang, D. Evans, and J. Katz, "Private set intersection: Are garbled circuits better than custom protocols?" in *NDSS*, 2012.
- [73] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs." in *NDSS*, vol. 6, 2017, pp. 15–43.
- [74] 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 *USENIX Security Symposium*, 2017.
- [75] R. Zhang, A. Cheu, A. Gascon, D. Moghimi, P. Schoppmann, M. Schwarz, and O. Suciuc, "Snpeek: Side-channel analysis for privacy applications on confidential vms," *arXiv preprint arXiv:2506.15924*, 2025.
- [76] W. Lee, T. Kim, S. Shin, J. Hur, and Y. Shin, "T-time: A fine-grained timing-based controlled-channel attack against intel tdx," in *Computer Security – ESORICS 2025: 30th European Symposium on Research in Computer Security, Toulouse, France, September 22–24, 2025, Proceedings, Part III*. Berlin, Heidelberg: Springer-Verlag, 2025, p. 323–341. [Online]. Available: https://doi.org/10.1007/978-3-032-07894-0_17
- [77] L. Wilke, F. Sieck, and T. Eisenbarth, "Tdxdown: Single-stepping and instruction counting attacks against intel tdx," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 79–93.
- [78] Google, "Protected auction key/value service," <https://github.com/privacysandbox/protected-auction-key-value-service>, 2025.
- [79] Z. Liu, Y. Yuan, Y. Chen, S. Hu, T. Li, and S. Wang, "Deepcache: Revisiting cache side-channel attacks in deep neural networks executables," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4495–4508.
- [80] A. Adiletta and B. Sunar, "Spill the beans: Exploiting cpu cache side-channels to leak tokens from large language models," *arXiv preprint arXiv:2505.00817*, 2025.
- [81] Z. Gao, J. Hu, F. Guo, Y. Zhang, Y. Han, S. Liu, H. Li, and Z. Lv, "I know what you said: Unveiling hardware cache side-channels in local large language model inference," *arXiv preprint arXiv:2505.06738*, 2025.
- [82] N. Carlini and M. Nasr, "Remote timing attacks on efficient language model inference," *arXiv preprint arXiv:2410.17175*, 2024.

Appendix A. Customer Profile Example

```
1 Customer Profile:
2 Name: James Gonzales
3 Home Address: 8717 Morris Green Suite 199, West
  ↪ Brendan, MN 84363
4 Email: amandaknapp@example.org
5 SSN: 808-87-7687
6 Credit Card Number: 4581225909566848
7 Credit Card Exp: 12/32
8 Credit Card CVC: 237
```

Listing 2: An example of a fictitious customer profile used in Section 5.4

One experiment described in Section 5.4 aims to leak information from a victim tokenizing a synthetically generated customer profile. An example of such a customer profile is shown in Listing 2. All samples contain a name, a home address with a United States ZIP code, an email address, a US Social Security number, and credit card details in the same format as shown in this example. To generate such profiles, we use Faker v.37.11.0 [65].

Appendix B. Optimizations for Locating Tokens’ GPAs

Recall that, as described in Section 5.2, to obtain the physical page of a representative token we need to run 3 queries and collect traces for them. One of such traces can be used for a subsequent token, so in total we need to prompt the LLM ≈ 4000 times as part of the attack, to recover all representative tokens (recall that our task is to recover ≈ 2000 representative tokens that are guaranteed to be in different 4 kB pages).

B.1. Optimization 1: Less Prompting

We can reduce the number of necessary prompts by batching several tokens in a single execution of the trick from Section 5.2. We could batch b tokens t_1, \dots, t_b in a single execution by defining $p_1 := (t_1)^r (t_2)^{2r} \dots (t_b)^{br}$, $p_2 := (t_1)^{2r} (t_2)^{4r} \dots (t_b)^{2br}$ and $p_3 = (t')^r$. We obtain traces for these three prompts and proceed as above, namely we then collect pages that (a) are common to t_1 and t_2 , (b) do not appear in t_3 , and appear more times in t_1 than in t_2 . This approach requires N/b queries, where $N \approx 2000$ is the total number of representative tokens for which we want to identify a page. Therefore, by setting, for example, $r = 10$ and $b = 50$, we just need to trace roughly $3 \cdot 2000 / 50 = 120$ prompt tokenizations, over a $10X$ improvement of lengths roughly $2r(b(b+1)/2) = 25500$.

B.2. Optimization 2: Shorter Prompts

The above approaches do not exploit the order in the traces. We can use the batching trick with shorter prompts

by using location to group traces, and get away with prompts of the form $p_1 := (t_1)^r (t_2)^r \dots (t_b)^r$. Then define $p_2 := (t_1)^{2r} (t_2)^{2r} \dots (t_b)^{2r}$. In this way, prompts are much shorter. For $r = 10$ and $b = 50$, we get lengths roughly $2br = 1000$.

Appendix C. Attacking De-Tokenization: Stealing LLM Responses

In addition to the user’s input, attackers in specific scenarios may also be interested in the inference result, i.e., the LLM’s output. This could be the case, for instance, with confidential, fine-tuned versions of public models that use the same dictionary as their public counterparts. As with the input, the output is a sequence of numeric tokens, which the victim translates back into human-readable character strings. In contrast to tokenization, this de-tokenization procedure does not require a hash map. Instead, pointers to the individual token strings typically reside in a linear array, which the de-tokenizer directly indexes with numeric tokens. Hence, as with tokenization, there are secret-dependent memory accesses that tools like TDXRay can observe. Using a cache-line-granularity access oracle, attackers can infer a set of candidate tokens with every lookup into this array, depending on how many token pointers fit into a single cache line. With `llama.cpp`, in particular, it is possible to identify the indexed token uniquely. The array it uses for de-tokenization contains additional information, amounting to 40 B per element with 8 B alignment. While this is smaller than a 64 B cache line, observing which cache line boundary the access crosses (if any) reveals the exact element under ideal conditions. As such, while we do not verify this in a practical attack, we expect de-tokenization to be similarly susceptible to side channel leakage as tokenization.

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

This paper introduces TDXRay, a host framework to read cache-granular access traces of TDX-based CVM workload as side-channel attacks. Using the four side channels identified from the existing literature, TDXRay can achieve page- and cacheline-level tracing capabilities.

D.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.

D.3. Reasons for Acceptance

- 1) This paper explored multiple side-channel vulnerabilities within TDX.
- 2) The paper provides a valuable step forward in confidential computing, while TDX has been widely deployed by major cloud vendors.
- 3) The evaluation using LLM inference is timely and useful.

D.4. Noteworthy Concerns

System-level mitigations beyond data-oblivious tokenization are not empirically validated.

Appendix E. Response to the Meta-Review

We agree that our system-level mitigations are theoretical. Data-oblivious tokenization (ORAM) serves as our primary, empirically validated, and targeted fix for critical components, where its cost is negligible compared to overall inference time. In contrast, empirical validation of broader system-level mitigations falls outside the scope of this paper. Naive alternatives, such as disabling the cache entirely, incur prohibitive performance penalties and fail to prevent page-table side channels (SEPTrace). Furthermore, our proposed architectural mitigations require hardware modifications that cannot be implemented on current Intel TDX silicon.