SENSE: Enhancing Microarchitectural Awareness for TEEs via Subscription-Based Notification

Fan Sang[1], Jaehyuk Lee[1], Xiaokuan Zhang[3], Meng Xu[4],
Scott Constable[2], Yuan Xiao[2], Michael Steiner[2], Mona Vij[2] and Taesoo Kim[1]
[1]*Georgia Institute of Technology,* [2]*Intel,* [3]*George Mason University,* [4]*University of Waterloo*

***Abstract*—Effectively mitigating side-channel attacks (SCAs) in Trusted Execution Environments (TEEs) remains challenging despite advances in existing defenses. Current detection-based defenses hinge on observing abnormal victim performance characteristics but struggle to detect attacks leaking smaller portions of the secret across multiple executions. Limitations of existing detection-based defenses stem from various factors, including the absence of a trusted microarchitectural data source in TEEs, low-quality available data, inflexibility of victim responses, and platform-specific constraints. We contend that the primary obstacles to effective detection techniques can be attributed to the lack of direct access to precise microarchitectural information within TEEs.**

We propose SENSE, a solution that actively exposes underlying microarchitectural information to userspace TEEs. SENSE enables userspace software in TEEs to subscribe to fine-grained microarchitectural events and utilize the events as a means to contextualize the ongoing microarchitectural states. We initially demonstrate SENSE's capability by applying it to defeat the state-of-the-art cache-based side-channel attacks. We conduct a comprehensive security analysis to ensure that SENSE does not leak more information than a system without it does. We prototype SENSE on a gem5-based emulator, and our evaluation shows that SENSE is secure, can effectively defeats cache SCAs, and incurs negligible performance overhead (1.2%) under benign situations.

I. INTRODUCTION

The ever-growing complexity and ubiquity of modern computing systems [1] have opened a Pandora's box of security challenges. One such challenge, microarchitectural side-channel attacks (SCAs) [2]–[12], has emerged as a threat to the confidentiality and integrity of sensitive information. These attacks exploit shared resources, such as cache memory, to covertly extract secret data from seemingly secure systems. Modern Trusted Execution Environments (TEEs) (e.g., Intel Software Guard Extensions (SGX) [13] and Trust Domain Extensions (TDX) [14], AMD Secure Encrypted Virtualization (SEV) [15], and ARM TrustZone [16]) aim to provide architectural protection against privileged attackers (e.g., OS and hypervisor) and consider some SCAs to be addressable by software, e.g., using constant-time programming techniques [6], [8], [17]–[26].

Although disabling resource sharing entirely at the hardware level (e.g., strict cache partitioning) mitigates many SCAs in TEEs, it is not practical and detriments utility. Constant-time programming is also challenging to deploy at scale. Sub-optimal hardware resource isolation techniques [27]–[31] try to find a balance between performance and security by allowing restricted resource sharing and dynamically partitioning a hardware structure (e.g., cache) among multiple security domains. However, hardware-based mitigations, designed with existing knowledge and fixed at hardware level once deployed, cannot be easily upgraded and adapted to defend future SCA strategies. Recent studies [32] even show that dynamic hardware resource partitioning still leaks information and accurately quantifying the leakage is hard.

To be more flexible as well as potentially catch the missed leakage, various detection mechanisms [33]–[41] for TEEs have been proposed. *Detection-based* mitigations rely on observing the impact of SCAs on the victim's performance (e.g., excessive cache misses) and detect ongoing attacks when anomalous performance characteristics are identified. Unfortunately, recent studies demonstrate that attackers can effectively bypass detection by leaking smaller portions of secrets (e.g., 1 bit) across multiple victim executions instead of inferring most secrets within a single execution [42]. Consequently, abnormal performance behaviors are amortized across multiple executions, preventing detection tools from differentiating between benign and potentially malicious executions. In fact, any detection tool relying on the victim's performance characteristics is likely to fail [42]. As long as information leakage persists, adversaries can compensate for the scarcity of information by adopting more computationally intensive strategies, as demonstrated by formal SCA models [43].

It is indeed an irony that in microarchitectural SCAs against TEEs, attackers have the freedom to collect various microarchitectural signals, including those from the kernel space, while victim programs running in TEEs are constrained in their ability to reliably gather SCA signals and lack runtime awareness of the microarchitectural context, since the OS cannot be trusted. More specifically, detection-based techniques under TEEs encounter a number of limitations: 1) *missing trusted data sources*, exacerbated in TEEs due to untrusted OS mediation [34]–[38]; 2) *low-quality data*, leading to imprecise attack detection and allowing stealthy attacks [34]–[40]; 3) *inflexibility*, as victims have limited contextual awareness and can only make coarse-grained decisions [34]–[41]; and 4) *platform constraints*, where techniques rely on platform-specific features, limiting extensibility and applicability to other architectures or future generations [33], [39]–[41].

Inspired by the practice of security through transparency (in contrast with security through obscurity), we believe that opening direct access to microarchitectural events with caution for TEEs can help to enable more complete forms of SCA defense

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24176
www.ndss-symposium.org

strategies as well as facilitate use cases beyond just preventing SCAs. As history has proven, providing more transparency to the public with careful mediation (e.g., open-source projects such as the Linux kernel and Kerckhoff's principle in cryptographic systems [44]) can contribute to the overall robustness of the system. To build a timely, accurate, flexible, and trustworthy technique that may fundamentally thwart microarchitectural SCAs, we argue that it is necessary to turn a *side channel* that can only be crafted by skilled attackers into a high-fidelity *direct channel* only accessible to victims in TEEs at runtime. This approach not only provides an immediate, effective, and future-proof response to potential attacks but also challenges the very foundation of existing SCA strategies, which relies on the information gap between the attacker and the victim.

SENSE. In this paper, we present SENSE, a paradigm-shifting hardware-software co-design solution that directly exposes events at microarchitectural level to userspace TEEs. SENSE is a general architectural extension that provides a reliable source of precise microarchitectural information and a flexible method for feeding microarchitectural events directly to TEEs. It achieves this by *directly notifying* userspace software in TEEs about microarchitectural events, allowing users to *proactively defend* themselves against SCAs using actions specified in the event handler, such as enforcing security invariants (e.g., pinning secret-dependent cache entries). As SENSE inevitably becomes part of the attacker's arsenal, we also conduct an in-depth security analysis of the SENSE architecture, SENSE handlers, and potential attack surfaces, and demonstrate that SENSE does not leak more information than a system without SENSE does.

While SENSE is designed to be a generic framework, in this paper, we enable SENSE specifically for thwarting cache-based SCAs for TEEs, as they are the most widely researched SCAs against TEEs. We prototype SENSE on a cycle-accurate gem5-based [45] emulator (§VI). Evaluation results show that SENSE can successfully defeat cache SCAs and incurs negligible overhead (1.2%) on reasonable TEE workloads under benign situations.

SENSE has more use cases than detecting SCAs. For example, secure software that typically incur high performance overhead (e.g., constant-time cryptographic algorithms) can be dynamically loaded (i.e., dynamic switching) in TEEs by SENSE handlers upon sensitive events, ensuring the performance penalty is only paid when necessary. SENSE can also be utilized to audit the faithfulness of an untrusted OS by verifying whether the contracts between the OS and userspace (e.g., cache coloring) are honored, owing to the rich microarchitectural information provided by SENSE.

SENSE serves as the first advocate for a transparent and trustworthy source of high-fidelity microarchitectural information dedicated to TEEs. SENSE is publicly available as an open-source project[1], allowing communities to test and contribute towards a more transparent microarchitectural paradigm for TEEs.

Contributions. This paper makes following contributions:

- **New approach.** We propose a novel approach to counteract microarchitectural SCAs in TEEs by actively exposing microarchitectural states to TEEs through notifications, turning a *side channel* that can only be crafted by skilled attackers into a high-fidelity and trustworthy *direct channel* accessible to victims within TEEs at runtime. We present our solution SENSE, an architectural extension that provides prompt, accurate, and trustworthy microarchitectural event notifications to userspace TEEs for flexible handling of potential SCAs.
- **SENSE for cache-based SCAs.** We provide a comprehensive, platform-agnostic description of SENSE architecture for addressing cache SCAs in TEEs, detailing the necessary processor architecture extensions and additional cache components to facilitate SENSE.
- **Security analysis of SENSE.** As SENSE inevitably falls into the attacker's arsenal as well, we conduct a comprehensive security analysis of the SENSE interface and possible attack scenarios to demonstrate its security.

[1] https://github.com/sslab-gatech/Sense

II. BACKGROUND

A. Microarchitectural Events

Microarchitecture comprises hardware components that are not directly accessible to software, as they are typically abstracted by the Instruction Set Architecture (ISA) and function (e.g., accelerate certain operations) transparently from the software layer. Microarchitectural events are generally related to instruction executions and memory operations. Instruction-related microarchitectural events include overall instruction fetch, issue, dispatch (execution) and retirement. Memory-related microarchitectural events include load and store operations on various memory components, such as CPU cache hit and miss, Translation Lookaside Buffer (TLB) hit and miss, and Page Table Walks.

However, while not *directly* accessible to the software layer, microarchitectural events can be inferred *indirectly* by carefully controlled software execution, hence, leaking information about software activities in unexpected ways. Furthermore, as all programs running on a machine share the same set of hardware components (i.e., microarchitectural states), attackers who can perform controlled execution of one program can leverage the observed microarchitectural events to infer the behavior of other programs. This forms the theoretical basis of side-channel attacks (SCAs).

B. Cache-based Side-Channel Attacks (SCAs)

Among a diverse set of SCAs, cache-based SCAs [2]–[5], [46]–[53] present a risk to secure computing across a variety of platforms and architectures, including TEEs such as Intel SGX [17], [24], [25], [54] and ARM TrustZone [52], [55]. These attacks can disclose both fine-grained and coarse-grained private data and operations, including bypassing address space layout randomization (ASLR) [48], [51], deducing keystroke patterns [49], [50], leaking sensitive information from human genome indexing computations [17], and exposing RSA [3], [53] and AES decryption keys [4], [56].

Cache-based SCAs exploit the time difference between cache hits and misses to infer secrets by learning whether specific cachelines have been accessed by the victim program. Commonly used techniques include Prime+Probe [2]–[4], [46]–[48] (to monitor cache set access patterns), and Flush+Reload [5], [49]–[51] (to evict shared target cachelines) while other variations have been proposed as well.

Detection-based defenses. To counter the aforementioned attacks, researchers have proposed various detection-based countermeasures [42]. A detection-based solution aims to identify ongoing attacks by monitoring program performance characteristics, such as cache miss rates and number of interrupts, to determine suspicious processes [33]–[38], [40], [41]. We cover detection-based techniques in depth in related work (§VIII).

Limitations. Detection-based strategies [33]–[38], [40], [41] are based on heuristics and face numerous challenges:

1) Missing trusted data sources in TEEs. Although the scarcity of data sources also applies to defending SCAs without TEEs, it is significantly exacerbated under TEEs. While direct information about microarchitectural events (e.g., cache events) is available through native interfaces (e.g., performance counters), the reliance on untrusted OS mediation (e.g., by registering a signal handler) renders these information sources inapplicable under TEEs [34]–[38], and no alternative sources for direct microarchitectural information exist.

2) Low quality of available data. The statistical and noisy nature of performance characteristics (e.g., number of page-faults or interrupts), along with existing microarchitectural information primarily intended for performance tuning and runtime profiling purposes (e.g., the number of cache misses), often leads to delayed and imprecise detection of attacks. This limitation allows for more "stealth" attacks [26], [47], [57] and leaves victims vulnerable to continued exploitation [34]–[40].

3) Inflexible. Due to the lack of detailed microarchitectural information in the userspace, victims have limited contextual awareness about underlying microarchitectural events and can only make coarse-grained decisions based on impressions. Existing techniques either terminate or retry the workload until success, restricting the flexibility of actions that victims can take [34]–[41].

4) Platform specific. In exchange for sacrificing the effectiveness of defending SCAs in TEEs, detection-based techniques gain practicality by avoiding hardware modifications and using existing platform-specific features (e.g., Intel Transactional Synchronization Extensions (TSX) [33], [39]–[41]) to collect abnormal performance characteristics and thwart SCAs for TEEs on the corresponding platforms (e.g., Intel SGX [33], [39]–[41]). However, relying on platform-specific hardware features makes these techniques non-extensible and inapplicable to other existing architectures or future generations, not to mention the potential risk of deprecation (e.g., TSX deprecation [58], [59]), limiting the generality of the solutions.

Reflection: It is ironic that TEEs, which are designed to shield a program from external inferences, are now blocking the program from using detection-based SCA countermeasures proactively. Even when the program inside a TEE has perfect knowledge on how it might be attacked (i.e., what microarchitectural signals to watch for), the victim lacks trusted medium to gather these signals, not to mention the performance penalties it has to pay for even coarse-grained data. In contract, attackers, not bounded by TEEs, have the freedom to gather all kinds of microarchitectural signals even from the kernel space.

III. Overview

While SENSE can be a generic mechanism for feeding microarchitectural events directly to userspace TEEs, in this paper, we initially demonstrate how to thwart cache-based SCAs—the most prominent type of SCAs in TEEs—with SENSE.

A. Targeted Platforms

SENSE targets platforms with a TEE component deployed and a modern cache architecture implemented.

Cache architecture. We assume a standard modern cache architecture, featuring multiple cache levels, including core-exclusive (L1 and L2) and shared (LLC) caches. Core-exclusive caches undergo flushing during context switches (which is aligned with most recent TEE architectures [31], [60], [61]). Additionally, we assume the cache controller is configured and only configured via dedicated registers, which is also consistent with common platforms.

Types of TEEs. SENSE is designed to accomodate both process-based TEEs (e.g., Intel SGX [13]) and guest-user processes within VM-based TEEs (e.g., AMD SEV [15] and Intel TDX [14]). From an architectural perspective, memory accesses differ between processes and VMs in that the latter must additionally traverse the extended page tables (EPTs). Leakage from EPT walks are out of scope for this work, and consequently the SENSE ISA extension (§IV-B) remains consistent across both processes and VMs. SENSE is not applicable to ARM TrustZone [16], which is a root TEE.

Trusted TEE component. TEEs provide established mechanisms to safeguard sensitive code within secure execution contexts known as enclaves. A trusted software component enforces access control mechanisms to maintain separation between secure enclaves and untrusted domains [13], [31], [60], [61]. Additionally, in the case of SENSE, the trusted software component handles other security-sensitive operations, such as determining the TEE status (i.e., whether the thread is under TEE or not) and setting up cache event monitoring and notification mechanisms, as detailed in Section §IV. We assume that security-critical metadata containing the TEE status is transmitted alongside memory requests. These assumptions align with current academic [31], [60]–[62] and industrial solutions [13], [63], [64].

Characteristics of TEE programs. We assume that the portion of isolated execution constitutes a minority of the application workload, as TEEs are most effective when the isolated execution is minimized to reduce the attack surface. This aligns with the intended usage of TEEs, where only small, sensitive code components are allocated to the TEE [27], [29], [30]. Although SENSE operates correctly even if the majority of the workload is isolated, the performance of isolated execution may be impacted [13], [27]. In addition, we assume that sensitive code uses writable shared memory solely for I/O, if at all, and access patterns to this shared memory do not leak information. Isolated code should focus on processing local data, limiting I/O needs to copying input and output data in and out of the component.

B. Threat Model

We adopt a robust adversary model aligned with hardware/firmware-defined TEE architectures [23], [39], [41],

where both the OS kernel and hypervisor are considered untrusted. The adversary can execute access-based [5], [49]–[51] and conflict-based [2]–[4], [46]–[48] cache SCAs to extract information from a sensitive execution domain (i.e., enclave). The adversary can initiate attacks from all privilege levels (excluding the highest level containing the trusted software component), access precise timing measurements and eviction instructions, and launch attacks from the same or a different CPU core.

Out of scope. We do not consider physical attacks on caches [65], fault injection attacks [66], or hardware flaw exploitation [67]–[69]; nor do we consider denial-of-service attacks from a security perspective. We assume that the adversary cannot compromise the trusted software component.

C. High-level Approach

The core of SENSE is an architecture-agnostic processor extension that features a new CPU SENSE *mode*, which notifies a running thread inside TEEs of the events it subscribes to. Only threads operating under TEEs run in SENSE mode and can subscribe to relevant microarchitectural events.

SENSE assumes that any TEE that is leveraged is in compliance with its original design intent, meaning that the majority of the execution workload is not security-critical and only a smaller portion is security-critical and isolated within TEEs (§III-A).

SENSE consists of three modules organized by functionalities. The *Subscription Module (SM)* (§IV-A) enables the subscription of cache-related microarchitectural events (e.g., cache evictions) for threads running in TEEs and monitors the occurrence of these events. The *Notification Module (NM)* (§IV-B) manages the CPU SENSE mode and provides the architectural interface of delivering microarchitectural events (e.g., cache events) to userspace TEEs for handling. The *Action Module (AM)* (§IV-C) allows the occurred events to be handled by a userspace event handler within TEEs.

The benefit of SENSE. SENSE is a novel interface for userspace applications in TEEs to listen to fine-grained microarchitectural events and take corresponding actions. Compared to detection-based defenses against SCAs (§II-B), SENSE has the following benefits:

1) Trusted native information source. By letting the hardware directly notifying the TEEs, SENSE bypasses the untrusted privileged software to provide microarchitectural information, fitting the threat model of TEEs. The control flow transition to the userspace handler is enforced by the CPU without changing privilege levels, ensuring that the handler operates within the same enclave of the program.

2) Prompt and precise notifications. SENSE notifies the userspace program about the events of the exact subscribed cache entries as soon as they occur, guaranteed by the existing communication channels of memory requests. By receiving instant notifications about cache events, victims can react promptly to potential cache SCAs, reducing the window of opportunity for attackers to leak secrets.

3) Versatility and extensibility. The subscribe-notify-act mechanism empowers victims with the ability to dynamically adapt their defenses based on real-time cache events, leveling the playing field with attackers who continuously evolve

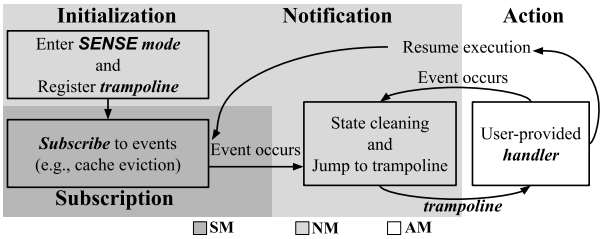

Fig. 1: SENSE high-level workflow.

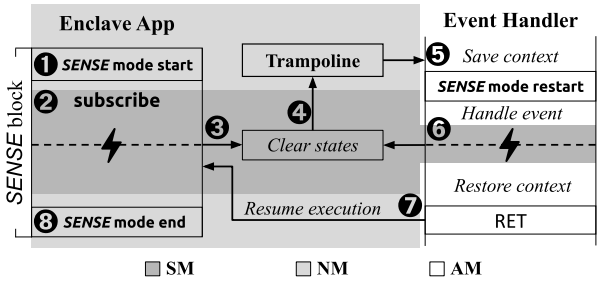

Fig. 2: SENSE in action.

their strategies. SENSE also supports custom event handlers. If necessary, developers can define their own handlers that suit their own applications best. Moreover, by exposing microarchitectural events, many applications beyond SCA defenses are made possible (§IV-E).

4) Compatibility and generality. SENSE is compatible with any existing partitioning and randomization-based cache SCA defenses without incurring significant extra hardware overhead. Besides, the NM is orthogonal to specific processor architectures, while the SM is cache organization agnostic. Moreover, SENSE is backward compatible and does not affect programs running without TEEs (§IV-D).

5) Security. As SENSE also falls into the attacker's arsenal, SENSE should not open new attack surfaces (§V).

D. SENSE *in Action*

A closer look at the SENSE notification flow and the collaboration among the three modules at runtime is shown in Fig. 1 and Fig. 2.

1) Initialization (NM): The enclave thread enters SENSE mode at the beginning of the enclosed security-critical block (❶). The address of the *event handler trampoline* inside the enclave is registered with the CPU for event notification. The trampoline is an address within the enclave where a *userspace event handler* will be invoked.

2) Subscription (SM): Cache microarchitectural events (e.g., cache evictions) for the memory resources within the enclave are subscribed under SENSE monitoring (❷).

3) Notification (from SM to NM): During the execution of an instruction at `RIP`, a subscribed event occurs (e.g., a monitored cache entry is evicted). The SENSE mode pauses and the microarchitectural states related to subscribed events are cleared (e.g., flush the cache) (❸). The control flow will be transferred by the CPU to the provisioned *event handler trampoline* inside the enclave (❹). If the event is an interrupt or exception, the

control flow will be transferred to the event handler trampoline after the interrupt or exception is served by the OS. Detailed microarchitectural information about the event that occurred (e.g., the thread identity that causes the event) is also supplied to the enclave. The trampoline pushes the RIP onto the user stack for later return and jumps to the SENSE handler within the enclave.

4) Action (from NM to AM): The SENSE handler saves the interrupted context, including the volatile registers and flag states to the enclave stack, and reenters SENSE mode. The handler then handles the event and restores the saved context before returning to RIP (❺). If another event occurs during the handling of the current event (❻), nested handling occurs by repeating steps ❸ – ❺.

Finally, the enclave thread resumes at RIP after the event is handled (❼). The enclave thread exits SENSE mode completely when it finishes, and then all subscriptions are canceled (❽).

IV. SENSE Design and Implementation

In this section, we provide design details of the three modules in SENSE, as summarized in the architectural overview in Fig. 3. Different implementations can be adopted for the SENSE design. To demonstrate the practicality of SENSE, we detail the prototype implementation of SENSE on a cycle-accurate gem5-based [45] x86 emulator with its out-of-order (O3) CPU model and the default Classical Memory System.

A. Subscription Module (SM)

Extensions to the existing cache architecture are required to precisely locate a specific cache event and promptly deliver the notification. In this section, we discuss two SENSE cache extensions: *precise subscription*, for locating the exact subscribed events, and *prompt delivery*, for initializing immediate notifications. The extended SENSE cache entry is shown in Fig. 3 and the cache controller logic for SM is depicted in Fig. 4.

Precise subscription of events. *Precise subscription* allows users to subscribe to the occurrence of events at the *exact* cache entry monitored by SENSE, instead of monitoring the entire cache component.

Requirements. Cache entries to be monitored should be precisely annotated when they instantiate in the cache. Depending on the structure of the cache, the SENSE annotation could appear as an extra bit indicating whether the cache entry is monitored under SENSE or as a hash map that records the information, for instance.

Design. An entry in cache will be annotated for SENSE status, preferably using an unused reserved bit to avoid extra hardware overhead. When a cache event of interest involves an SENSE-annotated entry, the cache component should raise a notification flag, indicating the need for the CPU under SENSE mode to deliver the notification for handling.

Prompt delivery of events. *Prompt delivery* ensures that the CPU under SENSE mode will immediately start delivering the notification to the enclave when a subscribed event occurs.

Requirements. An information channel needs to be established to promptly inform the NM about the raising of a notification flag in the SM. Fitting such a channel into the existing CPU microarchitecture is non-trivial. The channel should be close to the occurrence of the event, both temporally and spatially, to promptly relay the raised notification flag in a timely fashion. Meanwhile, the addition of the channel should not affect the correctness of existing microarchitectural components or impose too much performance overhead.

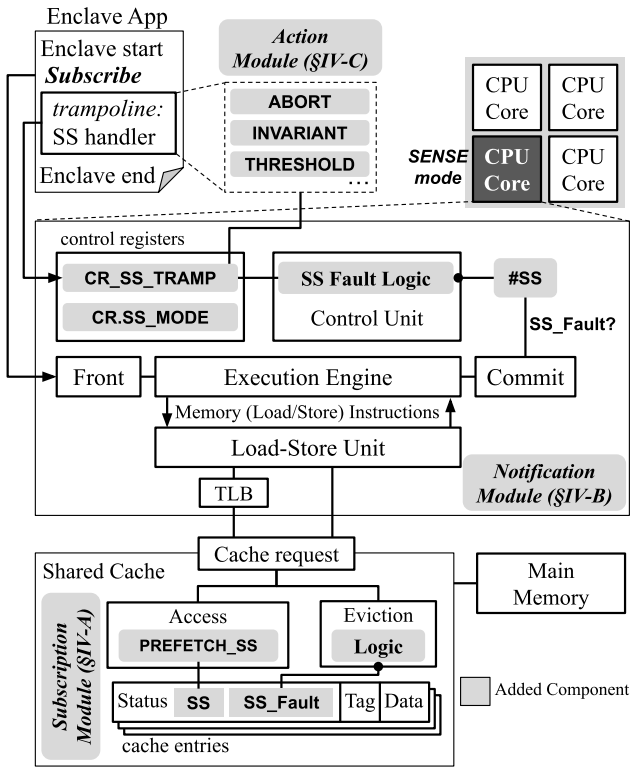

Fig. 3: SENSE architecture: Subscription Module (SM) (§IV-A), Notification Module (NM) (§IV-B), and Action Module (AM) (§IV-C).

Design. When executing a memory instruction (i.e., load or store), the CPU Load-Store Unit (LSU) first consults the TLB for address translation and then sends a load or store request to the cache. The status of the notification flag will be embedded in the existing responses sent back to the CPU LSU from the cache using reserved data space to avoid extra hardware overhead. When finalizing the memory responses, the CPU will start delivering the notification if the notification flag is set. Such a signal piggybacked onto the existing memory communication channel is aligned with the real-time workflow of the memory microarchitectural components and thus can immediately start delivering the notification as soon as a memory request triggers a monitored cache event.

Implementation. Achieving *precise subscription* and *prompt delivery of events* in the gem5-based x86 emulator incorporates five major tasks. Note that the tasks are not specific to gem5 but applicable to general modern CPU architecture.

1) Adding extra SENSE *status bits in each cache entry.* SENSE chooses to extend cache entries with extra bits. Specifically, each cache entry is extended with two SENSE status bits: SS bit and SS_FAULT bit (Fig. 3). The SS bit indicates whether the cache entry is monitored under SENSE mode, while the SS_FAULT bit represents whether the cache entry has been

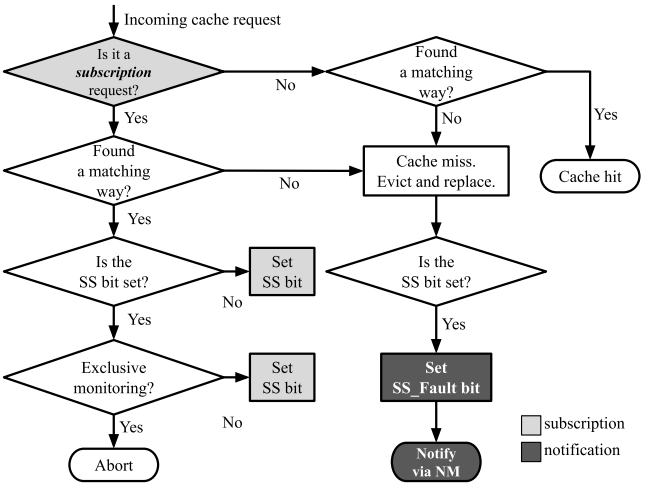

Fig. 4: SENSE cache controller logic.

evicted when monitored under SENSE. Both SENSE status bits are `off` (i.e., `0x0`) for all cache entries when booting up.

2) Initializing cache event subscriptions. In SENSE, subscribing to microarchitectural events occurs at the beginning of the enclave process. For cache eviction events, monitoring relevant cache entries under TEEs is realized by prefetching the entries into memory and turning on the `SS` bit. Therefore, similar to the `prefetch` macro-op, the subscription request boils down to a load (`ld`) CPU micro-op and is additionally extended with a `PREFETCH_SS` memory request flag to indicate the intention of event subscription under SENSE. At the beginning of the enclave process (e.g., enclave initialization), the cachelines corresponding to the sensitive data are prefetched into all levels of the cache hierarchy.

3) Marking the cache entries under SENSE monitoring. Load (`ld`) instructions enter the Load-Store Queue in the CPU execution engine when pipelined, and are executed by the CPU LSU. After the CPU LSU forwards the memory loading request to the corresponding microarchitectural component, i.e., cache, `PREFETCH_SS` flag is checked while performing the memory loading operation. If the `PREFETCH_SS` flag of the memory request is set (i.e., due to the memory prefetching under SENSE), the `SS` bits of the prefetched cachelines are turned *on* (i.e., `0x1`) (Fig. 4), meaning they are *watched* under SENSE mode. The set of cachelines under watch for the current TEE thread is called the *watch set*.

4) Signaling the occurrence of a subscribed eviction. Traditionally, during the eviction of a cache entry, either due to active flush (e.g., by using `clflush` instruction) or Least-Recently-Used (LRU) status caused by entry conflicts, the entry will be cleared and invalidated. If any member in an enclave thread's *watch set* (i.e., `SS` bit is set for the cache entry) is evicted while the enclave thread is executing under SENSE mode, the `SS_FAULT` bit will be set for the entry (Fig. 4), indicating an event of interest has occurred and the NM should be informed to immediately deliver a notification. When responding to the load/store request from the CPU LSU, the cache will check the `SS_FAULT` bit of the entry corresponding to the request.

5) Promptly informing the NM. If the `SS_FAULT` bit is set, the cache will flush all entries to eliminate cache microarchitectural traces and add an additional `SS_FAULT` flag to the memory request response. As soon as the CPU LSU receives the response, if the response has the `SS_FAULT` flag set, the CPU will enter the NM and trigger an SENSE notification (detailed in §IV-B) to the enclave thread using the information in the response. Eventually, the event handler trampoline starts execution, indicating a cache eviction event has occurred. When the enclave process finishes execution, the SENSE status bits of the entries in the *watch set* are turned `off`, and the *watch set* is reset.

Instruction	Description
`ssbegin [addr]`	Start SENSE mode with trampoline at `addr`
`ssend`	Terminate SENSE mode
`sstramp [addr]`	Push `rip` and transfer to `addr` after preparation
`sssub [addr][type]`	Subscribe to `[type]` event of resource at `addr`
`ssunsub [addr][type]`	Unsubscribe to `[type]` event of resource at `addr`

TABLE I: SENSE new ISA instructions.

B. Notification Module (NM)

CPU SENSE mode and new ISA instructions. A user-level enclave program under SENSE mode will get notified if a subscribed microarchitectural event has occurred. Upon notification, the CPU will supply detailed microarchitectural information to the enclave and jump to a registered *event handler trampoline*, where a *userspace event handler* will be invoked. By default, under SENSE mode, all interrupts and exceptions will be notified. An enclave process is also allowed to subscribed to cache microarchitectural events (e.g., cache evictions) related to specific memory resources in the enclave. The SENSE architecture provides several new ISA instructions, shown in Table I.

SENSE block and event notification. Any sequence of enclave instructions executed under SENSE mode is referred to as an SENSE *block*. If any subscribed event occurs during the execution of the SENSE *block*, the enclave thread will be notified. Specifically, an enclave thread is *notified* of subscribed SENSE events by noticing the control flow transfer to the *event handler trampoline*. This occurs when one of the following two conditions is met:

- A subscribed event is detected on the logical processor where the SENSE-protected enclave thread is executing;
- The enclave thread invokes an instruction that makes the SENSE architecture unable to track events for the thread (e.g., `SYSCALL`).

Upon a notification, the SENSE mode will pause (i.e., temporarily suspend delivery of SENSE notifications), and the microarchitectural states of the subscribed events will be cleared (e.g., flush the cache), which aligns with the strategies adopted by existing TEEs [31], [60], [61]. Pausing SENSE mode when an event occurs is necessary to safeguard the interrupted context information from being overwritten. This precaution prevents potential issues arising from an immediate subsequent event landing at the beginning of the handler, which is responsible for preserving and restoring the execution context (§IV-C). If an enclave thread running under SENSE mode is interrupted or triggers an exception, execution will resume at the *event handler trampoline* after the interrupt/exception has been serviced by the OS. This feature of SENSE is essential because microarchitectural events for the enclave thread cannot be handled by userspace handlers under untrusted kernel context.

Therefore, when a thread resumes execution, the event handler should reset the microarchitectural state for all events that were being tracked at the time the notification was delivered.

Modes of monitoring. As SENSE does not track the identity of the subscribing enclave in the cache entries, when enclaves on different cores monitor the same event, the event notification will be *broadcast* by default, i.e., delivered simultaneously to all subscribing enclave threads. This broadcasting strategy provides the convenience of sharing information about an event that might be of interest to multiple enclaves, while it may open up the attack surface due to improper synchronization and interaction among different event handlers (discussed in §V). To ease the effort of designing SENSE applications, SENSE offers the option to enforce *exclusive monitoring* of events in a *first-come, first-served* manner, which can be useful for applications that emphasize security. If an event is being exclusively monitored, only the enclave thread that subscribes to the event first will receive SENSE notifications; the subsequent subscription requests from other threads are aborted (e.g., exception), thus thwarting any potential interference among different event handlers.

Implementation. We implement the NM in our prototype by integrating extended control registers and modifying the existing interrupt controller logic, enabling hardware interrupts without privilege level changes, providing a streamlined and efficient mechanism for event notifications.

SENSE *control registers (CRs).* We equip each CPU logical core with a `CR.SS_MODE` control register bit and a `CR_SS_TRAMP` control register (Fig. 3). Implementing this upgrade incurs zero hardware overhead when utilizing unused bits (e.g., `CR4` bit 26-63 in `x86`) and reserved control registers (e.g., `CR5-7` in `x86`), if available. The `CR.SS_MODE` bit holds the status about whether the CPU core is currently under SENSE mode. When an event occurs, the CPU will check the `CR.SS_MODE` bit and decide whether to deliver a notification. The `CR_SS_TRAMP` register hosts the address of the userspace *event handler trampoline* (§IV-B), the location where the control flow lands after the CPU decides to deliver the notification. We implement the new ISA instructions (Table I) as follows:

- `ssbegin`: This instruction is invoked by the trusted software component of the TEE at the beginning of the enclave program. The `CR.SS_MODE` bit will be set to indicate that the CPU enters SENSE mode if and only if the running thread is under TEEs (e.g., determined by the trusted software component), and the address of the reserved *event handler trampoline* will be loaded into the `CR_SS_TRAMP` register.
- `ssend`: This instruction is invoked by the trusted software component of the TEE at the termination of the enclave program. The `CR.SS_MODE` bit will be unset to stop the delivery of notifications if the finishing thread is under TEEs, and the `CR_SS_TRAMP` register will be cleared, terminating the SENSE mode completely.
- `sstramp`[2]: This instruction is invoked inside the trampoline. It pushes the current `rip` for later return and transfers control to the user handler.

[2] `entramp` is extendable for custom prepossessing operations before jumping to the handler, such as those inserted by the compiler but omitted by the hardware control flow transfer. For example, `entramp` can be extended to optionally decrementing `RSP` for `n` bytes to protect the stack red zone [70].

```c
void handler(int ss_info) {
  /* Implement policies.*/
  if (ss_info == 1) {
    /* Handle accordingly */
  }
}

/* _ssbegin(handler) by TEE
      at enclave entry */
void enclave_program() {

  /* watch resource at ptr
  with size and type */
  _sswatch(ptr, size, type);
  ...
  /* Event occurs */
  /* Resuming point */

  /* unwatch resource at ptr */
  _ssunwatch(ptr);
}
/* _ssend() by TEE
      at enclave termination */
```

Fig. 5: An example of SENSE in an enclave application.

```c
/* Pseudo code
      of _ssbegin() */
__inline void _ssbegin() {
  __asm {
    trampoline:
    // landing point
    sstramp sshandler

    sshandler:
    call save_cpu_states
    ssbegin trampoline
    mov $rdi, $rax
    call g_handler
    restore cpu states
    ret

    entry:
    // Update the ptr
    g_handler = handler
    // save trampoline
    ssbegin trampoline
  }
}
```

Fig. 6: Pseudo code of `_ssbegin()` function.

- `sssub`: This instruction emits a subscription request detailed previously in §IV-A. The cacheline corresponding to `addr` is prefetched into all levels of the cache hierarchy.
- `ssunsub`: This instruction clears the monitoring status set by `sssub`.

Event notifications in the form of SENSE *Faults.* The delivery of a notification takes in a form of a dedicated hardware exception triggered by the CPU using NM, minimizing the modification to the existing hardware architecture. We define such a hardware exception as SENSE *Fault (#SS)*, which occupies an unused interrupt/exception vector (e.g., 20 under `x86`) of the CPU. Supplemental information can be included in the fault and passed to the handler for handling, such as the thread identity that causes the event.

SENSE *Fault control logic.* When an SENSE Fault occurs, instead of vectoring into a preregistered hardware exception handler in the interrupt vector table, the SENSE thread is trapped into a dedicated piece of SENSE Fault handling logic in the CPU control unit (Fig. 3). The SENSE Fault control logic performs the following decisions and operations. Under SENSE mode, indicated by the `CR.SS_MODE` bit, the CPU will first pause SENSE mode by unsetting the `CR.SS_MODE` bit. Then, the CPU will check whether the the `CR_SS_TRAMP` register holds a valid address by testing the address. Finally, the CPU will extract the supplemental information corresponding to the event supplied in the SENSE Fault, push the data to the enclave stack, and transfer the control flow to the address specified in the `CR_SS_TRAMP` register. Note that the control flow transition to the userspace handler is enforced by the CPU without changing privilege levels, ensuring that the handler operates within the same enclave of the program.

Software support. We provide wrapper functions with function signature `_func()` for SENSE ISA instructions for ease of use. A code snippet for basic software usage is shown in Fig. 5. The SENSE block is defined by wrapping the enclave code section with a pair of `_ssbegin()` and `_ssend()` functions by the trusted software component of the TEE at the enclave entry and termination, respectively. `_ssbegin()` is a wrapper function for `ssbegin` and `sstramp` ISA instructions. The pseudo code of `_ssbegin()`

is shown in Fig. 6. `_ssbegin()` first saves the event handler's address and starts SENSE mode with an event handler trampoline using the `ssbegin` ISA instruction. The trampoline and the handler then follow the semantics defined by SENSE. `_sswatch()` is a wrapper function for the `sssub` ISA instruction to subscribe to data or instruction at address `ptr` for eviction event `cache_ev`. When an event occurs, the program control flow is transferred to the trampoline, followed by the preparation and the invocation of the handler. The CPU states are then restored before resuming the process. `_ssunwatch()` is a wrapper function for the `ssunsub` ISA instruction to unsubscribe the eviction event at address `ptr`. Finally, the `_ssend()` terminates the SENSE mode.

C. Action Module (AM)

The event handler first saves the interrupted context, including volatile registers and flag states, onto the enclave stack. Then, the handler reenters the paused SENSE mode and handles the event accordingly. Finally, the handler restores interrupted context and invokes `RET` to resume the interrupted program execution. An example is provided in Fig. 6. We assume the event handler is a part of the trusted software component of the TEE.

SENSE provides default handlers, i.e., `ABORT`, `INVARIANT`, and `THRESHOLD`, for common handling of the events (Fig. 3).

`ABORT`. The `ABORT` handler simply aborts the enclave process upon the occurrence of the event, a pessimistic strategy adopted by various existing detection techniques [34]–[41]. It provides the most constrained security policy for an enclave process that forbids any subscribed events. It targets events defined by the user that are absolutely forbidden from the normal execution of the enclave, such as writes to data from a malicious process during an atomic operation.

Implementation. Upon the eviction of members in the *watch set*, the `ABORT` handler will simply halt the enclave process by invoking `exit(0)`.

`INVARIANT`. The `INVARIANT` handler aims to preserve a safety property as an invariant for the enclave process. When the safety property is tampered with due to an SENSE event, the event handler could reestablish the property. Therefore, from the enclave perspective, the safety property is never broken. For example, a cache residency invariant for the enclave (i.e., retain the specified cachelines in the cache) can be preserved by refetching the evicted cacheline in the event handler.

Implementation. To maintain the cache residency of the *watch set* during enclave operations, the `INVARIANT` handler will *re-*subscribe to all of the cache entries in the *watch set* so that the entries are prefetched again and stay in cache after eviction. Upon eviction on monitored cache entries under SENSE mode, the *watch set* is *re-fetched* by subscribing to all *watch set* members again, thus preserving the cache residency of the enclave process. From the application's perspective, the cachelines in the *watch set* are effectively pinned into the cache. A detailed example of the `INVARIANT` handler is shown in Appendix §A.

`THRESHOLD`. The `THRESHOLD` handler keeps a thread-local counter that is incremented each time the handler is invoked. The user could define a termination policy in terms of this counter, such as *terminate after n events are detected*. As there is not a universal threshold, the developer can tailor a custom threshold for each application. For example, a security policy for the AES T-table will likely differ from one for a data-intensive function, e.g., a machine-learning algorithm.

Implementation. The threshold can be determined according to the *watch set* size and eviction times under normal circumstances. For instance, an AES-256 encryption key may be monitored under the *watch set* as four cachelines of 64 bytes. During encryption, if the key is evicted k times under normal conditions, a user might set the threshold at $2k$, allowing for a small amount of noise while terminating upon detecting suspicious actions by invoking `exit(0)`. Recent studies [32] have proposed more precise methods for automatically determining the optimal threshold value by systematically modeling and quantifying information leakage.

Custom event handlers. Besides the default event handlers, SENSE allows developers to design and register custom event handlers tailored for other needs. However, they should follow the assumptions of proper enclave workloads (§III-A) as handlers handle events within the enclave.

D. Compatibility

Backward compatibility. Similar to supplementary processor features like Page Attribute Tables (PATs) and Memory Type Range Register (MTRR), SENSE offers on-demand cache event notifications while maintaining backward compatibility. SENSE effectively provides cache event notifications during execution only when processes are running under TEEs, which are indicated by the SENSE status bit of each cache entry and communicated with each cache request. Otherwise, from a perspective of software outside TEEs, a CPU with SENSE functions identically to one without SENSE. If the trusted hardware is not initialized and deployed for processes, SENSE is designed to not turn on the SENSE status bit by default to incoming cache requests, treating all executions as not under TEEs with cache event notifications disabled. Only when trusted hardware or firmware support is provided, do CPUs with SENSE differentiate for TEEs and offer its microarchitectural notification capabilities.

Synergistic compatibility. SENSE is compatible with all existing partitioning and randomization-based defenses against cache-based SCAs in TEEs [27]–[31], which attempt to address such SCAs at their root cause (i.e., cache organization). This compatibility is due to the SENSE architecture-agnostic design, which does not rely on platform-specific hardware features and is independent of cache organizations (e.g., set-associativity). Furthermore, when combined with existing partitioning-based techniques, SENSE can reduce additional hardware overhead to near zero. Existing partitioning and randomization-based solutions already extend cache entries with multiple bits (e.g., 4 bits [27], [29]) for information tracking and verification, which can be shared with SENSE for tracking its status. Additionally, these solutions also extend cache controller logic [27]–[31] to enforce resource allocation and isolation, allowing SENSE's cache controller logic to be integrated with minimal additions.

Architectural compatibility. SENSE aims to offer ISA abstractions (Table I) that are sufficiently generic to be implementable on diverse architectures (e.g., `x86`, ARM, RISC-V, etc.) to enable microarchitectural event notifications. However, the microarchitectural implementation will likely vary across different chip designs. For example, the presence of a cacheline is generally easier to establish in an inclusive cache hierarchy than in a

non-inclusive one. SENSE is also designed to minimize side effects on an existing CPU execution pipeline, making it realistic on real-world hardware. We describe how we align SENSE modules with the existing CPU execution pipeline, including out-of-order execution and speculative execution, in Appendix §B.

E. SENSE *for General Hardening*

As SENSE provides notifications on finer-grained microarchitectural events, more advanced defense mechanisms are made possible with such capabilities.

Case 1: On-demand loading of secure libraries under attack. Software-based SCA mitigation techniques generally incur heavy performance overhead in order to hide secret-dependent microarchitectural traces with extra operations. For example, constant-time cryptographic algorithms can effectively defeat timing-based SCAs but incur performance overhead that largely hinders their adoption in applications. During normal executions, the performance overhead paid for constant-time cryptographic operations is unnecessary; on the other hand, with the concern of SCAs, it is insecure to rely on optimized but vulnerable libraries all the time. Ideally, applications want to benefit from the optimized performance in the less-secure libraries and pay the overhead of the secure libraries only when under attack, which is difficult to achieve.

SENSE can be utilized to reach this goal. The application can first execute using an optimized but potentially vulnerable library (e.g., vulnerable OpenSSL). When malicious microarchitectural states are discovered under SENSE (e.g., an excessive number of cache evictions), the more secure version of the library (e.g., constant-time OpenSSL) can be loaded on demand by the event handler, replacing the potentially vulnerable one. We present the evaluation of such an application in §VI-E.

Case 2: Verifying contracts with the OS. The OS is in charge of managing system resources with a higher privilege than applications. To this end, there is a hidden contract between applications and the OS: the OS will fulfill the requests from applications with due diligence and perform privileged tasks on behalf of applications faithfully. However, an untrusted OS does not always honor such contracts. To verify the contracts with the OS, applications can implement a verification logic in a trusted way using SENSE.

To illustrate, a concrete use case is whether the OS enforces cache coloring [71], [72] as it claims to be, i.e., physical addresses that belong to a particular color are only accessible (i.e., read, write, evict) by the threads assigned the same color. When using cache coloring to isolate cache accesses in SCA mitigations, one cache color can be assigned to at most one thread, thus preventing cache sharing between the victim thread and malicious threads. When a security-critical cacheline in the enclave application's *watch set* is evicted, the handler should therefore verify whether the cacheline is evicted by the enclave thread itself. If not, the OS is not following the contract and might be unfaithful. Further actions such as terminating the process can be enforced to protect the process from the malicious OS. We present the evaluation of such an application in §VI-F.

V. SECURITY ANALYSIS

Each existing side-channel defense aims to mitigate a well-defined vulnerability and thus does not worry about increasing the attack surface. In contrast, SENSE, being a non-traditional interface that explicitly exposes microarchitectural information to TEEs, faces various unpredictable situations. We first introduce the security guarantees enforced by the SENSE architecture. Then, we analyze possible attack surfaces and show that SENSE does not empower attackers more than the system without it.

A. Security Guarantees

1) Delivery of SENSE *notifications is guaranteed.* The delivery of the notification is enforced by the trusted hardware (CPU). The SENSE *Notification Module* ensures that the control flow will be directed to the event handler trampoline when an event occurs under CPU SENSE mode. The trampoline in the enclave then guarantees the execution of the event handler.

2) SENSE *handlers are inaccessible from other threads.* As SENSE is core specific, SENSE handlers are only accessible by the enclave thread that sets the trampoline. When an event occurs or the SENSE thread is suspended, the SENSE mode pauses and the control flow is transferred to the SENSE trampoline (§IV-B). The SENSE trampoline directs to the event handler, and the address of the trampoline is registered in the SENSE control registers at the start of the enclave process; thus it is specific to the thread. Therefore, the event handler is only invokable by the enclave thread that registers the trampoline and no one else.

3) Microarchitectural states are cleared when SENSE *mode pauses/exits.* When an event occurs, the monitored microarchitectural states (e.g., the watch set) are cleared (e.g., flushed) before jumping to the trampoline to avoid leaving microarchitectural traces when events are not monitored, which is aligned with the strategy of existing TEEs.

4) Security of event handlers. SENSE guarantees only the execution of the userspace handler when the event occurs, while SENSE does not make any assumption about the security of the handler. Default handlers provided by SENSE (i.e., `ABORT`, `INVARIANT` and `THRESHOLD`) can be used directly if they already meet the functionality requirement; however, SENSE does not verify whether custom handlers are secret-independent, audit the custom handlers for potential vulnerabilities, or prevent the custom handlers from intentionally leaking the secret, which should all be prevented when designing a new custom handler.

B. Attacker's Exploitation of SENSE

We examine whether the attacker can utilize SENSE to acquire extra information about the victim that is not available otherwise. We assume that there are two threads running, i.e., one victim thread and one attacker thread. Within the TEE threat model, the victim thread typically operates inside a TEE, whereas the attacker can be any other thread outside of that TEE. Additionally, there's the potential risk of an attacker deploying a rogue TEE to subscribe to SENSE events, thereby monitoring the victim's behavior.

Same logical core. When running on the same core with the victim, the attacker cannot monitor events in the victim thread, as they cannot run at the same time. However, the attacker is able to learn about the absence of cache microarchitectural states, which is cleared after the SENSE process is context-switched out (e.g., a timer interrupt). This does not leak information about the program activity since

the cache microarchitectural states appear all as *cleared* (e.g., all cache misses in cache) to the attacker.

SMT and different physical cores. When the attacker and the victim are located on different cores, they can interfere with each other through shared microarchitectural resources such as L1/L2 cache under SMT or the Last Level Cache (LLC) shared by different physical cores. There are three attack scenarios when SENSE is available to both the attacker and the victim.

1) Only the victim is using SENSE. That is, the attacker is not a TEE thread. The attacker can maliciously trigger an SENSE event (e.g., evict the *watch set*) and invoke the corresponding handler registered by the victim. The attacker can also choose to invoke a specific handler that is less desired by the victim thread (e.g., triggering abort instead of preserving an invariant) by targeting the corresponding enclave thread. The attacker can learn about the victim's *watch set*, which is the set of the memory objects in the enclave. It does not reveal useful information since the enclave memory is already managed by untrusted privileged software. The attacker cannot infer cache activities, as any cache probing will be notified to and handled by the enclave (e.g., pin the cache).

2) Only the attacker is using SENSE. That is, the attacker is a malicious TEE thread targeting the OS and userspace. By requesting notifications on specific cacheline evictions, the attacker can infer the victim's cache access behavior with lower noise and delay, hence boosting the efficiency of an attack. More specifically, with SENSE, a malicious TEE thread can probe victim cache accesses without a traditional coarse-grained *timing measurement* phase, as SENSE promptly delivers notifications to the attacker. This reduces the overhead to launch an attack (as evaluated in §VI-B). However, SENSE does not expose *new* information to a malicious TEE thread through notification delivery. The cacheline access information an attacker can subscribe to using SENSE can already be obtained without SENSE [3]–[5], [73].

Furthermore, an OS possesses alternate defense mechanisms to shield itself and its user processes from a potentially malicious TEE. For example, the OS can mitigate SMT attacks (e.g., L1, L2 cache attacks) by leveraging Linux's core scheduling to ensure that threads belonging to a potentially malicious TEE are not co-scheduled on the same physical core with another process's threads [39], [74]. Cache partitioning strategies [27]–[31] (or coloring techniques [75]–[77]) can be employed to combat cross-core attacks (e.g., LLC). By designating a specific partition, an OS can isolate a potentially malicious TEE, and this approach can seamlessly integrate with SENSE as discussed in §IV-D.

3) The attacker and the victim are both using SENSE. That is, both the attacker and the victim are TEE processes. In this case, the attacker has extra capability due to simultaneous handling of the event. We give an example security caveat under such a circumstance. Suppose that both the attacker and the victim monitor a shared data region (e.g., AES T-table), each with an SENSE handler registered. Two of the cachelines monitored by the victim, A and B, map to the same cacheline in the LLC. The victim's `THRESHOLD` handler checks the cache eviction events and aborts the process if it exceeds a preset threshold, while the attacker's `INVARIANT` handler waits and refetches the evicted entry. Upon cache eviction, the attacker and the victim will thus both get notified simultaneously, followed by the invocation of corresponding handlers.

Suppose that the victim program touches A and B in a sequence, and originally B is in the LLC. The initial threshold counter (tc) of the victim is 0. When the victim visits A, B is evicted from the LLC and replaced by A, which triggers both handlers. The victim handler increases the counter ($tc = 1$); the attacker handler refetches B into the LLC. When the victim visits B, since B is already in the LLC, no eviction is needed. However, if there is no attacker, B will not be present in the LLC; thus, one more eviction is needed to replace A with B ($tc = 2$). In this case, the attacker earns one eviction quota, which can be used to perform malicious evictions (e.g., *prime* the eviction set) while avoiding detection by the victim handler.

The root cause of such a problem lies in a lack of enclave identity when monitoring cache entries. As a result, the notification cannot be sent to only the monitoring thread but to all enclave threads under SENSE. This causes the unpredictable interaction and synchronization among handlers from different sources. Such a hidden interaction of cache microarchitectural events among different handlers is difficult to prevent, as both the method of interaction and the number of interacting parties are unknown.

This unpredictable interference among handlers can be mitigated by integrating the enclave identity into the monitored cache entries so that the event notifications can be demultiplexed according to the enclave identity. However, this approach increases the hardware overhead, while using such an interference to collect meaningful microarchitectural traces is only theoretically possible. Nonetheless, the hardware overhead of tracking enclave identities of the monitored cachelines can be eliminated if SENSE is deployed together with other partitioning-based mitigation techniques for TEEs that are already tracking the enclave identities at the cacheline level ([27], [30]). When SENSE is deployed alone on vanilla processors instead, to minimize the likelihood of such a threat, it should be configured with the *exclusive monitoring* feature (§IV-B) offered by the SENSE architecture while sacrificing the functionality of *event broadcasting*.

VI. EVALUATION

Experiment setup. We implement and perform our evaluation on SENSE using a cycle-accurate gem5-based [45] `x86` emulator with its out-of-order (O3) CPU model at 4 GHz and the default Classic Memory System. The server for our gem5 emulation is running Linux kernel version 4.8.1 and is equipped with a 4-core Intel i7-6700K CPU (Skylake) operating at 4 GHz, and 64 GiB of RAM. Each evaluation experiment conducted is run 10 times and we calculate the average values as the final results.

A. Security Evaluation

Harden AES T-Tables. To evaluate SENSE on closing timing-based cache SCAs, we showcase the Prime+Probe attack against the the vulnerable AES T-table implementation with and without the protection of SENSE. Although the T-table is disabled by default in OpenSSL, it is still widely used to evaluate new and existing SCAs. The attack implementation is based on strategies from previous works [78]. During encryption, the T-table entries are used together with the secret key k to compute the ciphertext from the plaintext p. Specifically, the

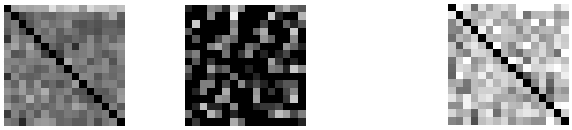

Fig. 7: Color matrices showing cache hit patterns of the first AES T-Table ($k_0 = 0x00$) under Prime+Probe attack. Left: attack without SENSE; Right: attack with SENSE cache `INVARIANT` handler.

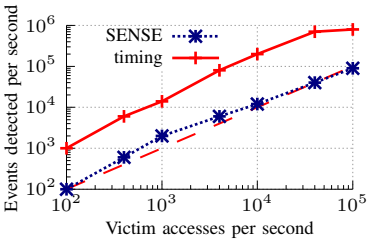

Fig. 9: Cache access detection rates using the timing channel and using SENSE notifications. The $y = x$ line indicates perfect performance, i.e., all events are detected without false positives. Being more congruent to the $y = x$ line means lower false positive rate.

key bytes and plaintext bytes are used as indexes to access the T-table entry $T_j[p_i \oplus k_i]$, where $i \equiv j \mod 4$. An attacker is able to use Prime+Probe to leak the access patterns of the T-table entries, which reveals the values of $p_i \oplus k_i$. As a result, k_i can be computed when p_i is chosen by the attacker.

Each cacheline (i.e., 64B) can hold 16 T-table entries. We launch the attacks against the first line of the first T-table `Te0` (i.e. $k_0 = 0x00$) to leak its access patterns for demonstration [49]. First, we run the attacks without SENSE protection. Then, we monitor cache eviction events of the encryption operation using SENSE, assuming the encryption operation is typically the critical section executing under TEEs. The `INVARIANT` handler refetches the evicted cacheline, as described in §IV-C, to effectively pin it in the cache throughout the attack so that the timing side channel is concealed from the attacker. We show the sampled cache access patterns of the first line in T-table `Te0` in Fig. 7. The T-table without SENSE protection reveals a clear cache hit pattern under the Prime+Probe attack, while the T-table protected by SENSE does not leave useful microarchitectural traces for the attacker to infer secret key bits. The same experiment for Flush+Reload is shown in Appendix §C.

B. Attacker's Exploitation of SENSE

We evaluate the efficiency benefits brought by SENSE to a malicious TEE (§V-B). On average, identifying *a single cache access* via SENSE notifications is ∼8× faster than probing using a traditional timing channel (i.e., 576000 vs. 4649000 emulated CPU cycles, respectively), allowing the attacker to carry out more probing attempts within the same timeframe. Fig. 9 represents the success rates of detecting victim memory accesses through both techniques. For this experiment, the victim consistently accessed a single memory location while the attacker aimed to monitor these accesses. The results highlight that while both methods can identify victim accesses, SENSE notifications have a reduced false positive rate compared to the conventional timing channel. This improved accuracy stems from SENSE's immediate hardware callback upon cache evictions, obviating any requirement for a potentially error-prone *timing measurement* phase as in traditional cache attacks.

Fig. 8: Cache hit patterns using SENSE notifications for probing. The signal is stronger compared with Fig. 7 Left.

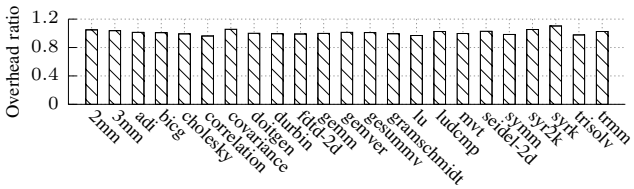

Fig. 10: Performance overhead of the Subscription Module (SM). Average overhead is 1.2% of the overall execution.

We further show the efficiency benefits during an actual attack using the same Prime+Probe attack outlined in §VI-A. The primary segment of Prime+Probe utilizing SENSE mirrors the *prime* phase of a regular Prime+Probe attack, except that it primes by subscribing the eviction set to cache eviction events first. Once the *prime* phase is completed, the attacker simply waits for a SENSE notification, from which the attacker can conclude that another program has accessed an address in the target cache set. This is similar to the information leaked by a regular Prime+Probe attack. As shown in Fig. 8, assisted by both the faster execution speed and the lower false positive rate of SENSE notifications, the signal is stronger when using SENSE for Prime+Probe.

C. Performance Evaluation

We first evaluate the performance of each SENSE module, i.e., *Subscription Module (SM)*, *Notification Module (NM)*, and *Action Module (AM)*, using benchmark PolyBenchC ([79]). Next, we evaluate the overall performance of SENSE when protecting AES encryption from attack.

Benchmark performance of SENSE without attacks present. The overhead of SM comes from prefetching the secret data into cache and marking cache entries as SENSE monitored. The overhead from NM lies in the extra CPU control logic that relays the cache event information, microarchitectural state cleaning upon notification, and control transfer to the trampoline. The overhead of AM solely depends on the handling logic.

We use PolyBenchC [79] to conduct the performance evaluation. Each benchmark in PolyBenchC has a small matrix function, which we treat as the minimal critical section (§III-A) executing inside a TEE and thus monitored by SENSE, in our experiment. The functions behave similarly to how look-up tables are used in AES encryption. We then launch the programs and measure the execution time of each module by the CPU cycles emulated in gem5.

1) Performance of Subscription Module (SM). The SM prefetches all security-critical memory objects by iterating through the objects as cacheline-size data blocks (i.e., 64B) to be put under SENSE monitoring. The results are shown in Fig. 10. On average, the SM incurs a 1.2% overhead compared to the execution time of the original program, which is negligible.

2) Performance of Notification Module (NM). The result is shown in Fig. 11. Under normal cases, as the monitored data size is smaller than the cache size, cache evictions on the

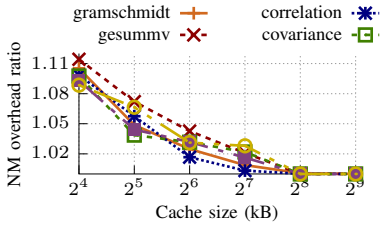

Fig. 11: Performance overhead of the Notification Module (NM).

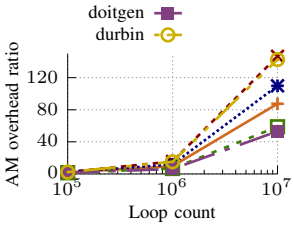

Fig. 12: Performance overhead of the Action Module (AM). The cache size used is 32kB to incur more evictions.

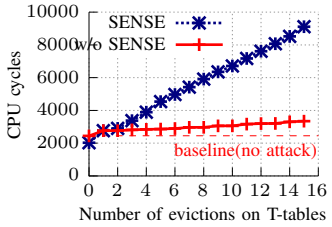

Fig. 13: Performance of AES encryption w/ and w/o SENSE under attack. The red dotted line indicates the baseline performance of AES encryption without experiencing attacks.

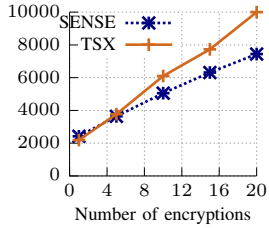

Fig. 14: Performance of AES encryption under SENSE and TSX *without attack*.

monitored data do not occur often, and thus the performance impact is negligible. Up to a cache size of 256kB, the overhead incurred by NM on PolyBench kernel functions is imperceptible. As a showcase of extreme scenarios, we also radically reduce the size of cache of the gem5 emulation configuration to as low as 16kB to intentionally increase cache contention so that more events can be triggered. As shown, the smaller the size of cache, the more performance overhead NM incurs, as the cache evictions due to cache conflicts occur more often under a smaller cache, which matches our initial speculation. In conclusion, with a realistic cache size and a minimal enclave program, the NM incurs negligible performance overhead, while under extreme cases where the cache size is impractically small to serve the enclave program, extra overhead will be caused by increased notifications.

3) Performance of Action Module (AM). To analyze the performance overhead of AM, we use dummy handlers that perform busy waiting inside a loop. By increasing the size of the loop variable, we can simulate the increase of handler complexity and analyze the correlation between performance overhead and complexity of the handler. To make the behaviors of the handler more observable, we use a cache size of 32 kB, which is shown to trigger the event quite often in the previous experiment. The result is shown in Fig. 12. Not surprisingly, as the handler becomes more complex, the AM incurs more performance overhead. Overall, the performance of the handler dominates the overhead of SENSE. Note that this is only to showcase the overhead of the AM under extreme cases with particularly small cache size and over-complicated handler logic. With practical cache size and minimal enclave programs, events are rarely observed, as shown in the previous experiment, and the complexity of default handlers is much simpler than the ones in the experiment. Therefore, the AM also incurs negligible performance overhead under practical use cases without attacks in place.

Performance of SENSE under attack. We compare the performance impact of SENSE when protecting AES encryption under attack by using the `INVARIANT` handler. The results are shown in Fig. 13, where the red dotted line indicates the baseline performance of AES encryption without attack. The AES encryption is typically fast without inducing self-evictions. When under attack, T-table entries are intentionally evicted from the cache (e.g., using *prime* phase) for launching cache SCAs such as Prime+Probe. We observe a minor and negligible decrease of performance when experiencing (potentially malicious) cache evictions on T-table entries during AES encryption without SENSE. When protected by SENSE, the performance of AES encryption quickly decreases as the number of experienced evictions increases. The performance degradation is mainly caused by the SENSE notifications and the event handling by the `INVARIANT` handler to fetch the evicted T-table entries back into the cache. As a reference, the Intel TSX-based mitigation technique [33] against cache SCAs is reported to reduce the performance of AES encryption with T-table to 23.7% (∼4.22×) of the baseline performance under a Prime+Probe attack. The lower performance while experiencing evictions is justified given that the timing channel is eliminated. Besides, the attacker has to heavily load the whole CPU to launch the attack, which occupies a significant amount of hardware resources and largely slows down the performance already [33].

D. Comparing with TSX

SENSE is influenced in part by Intel TSX [80], which was deprecated recently [58], [59]. TSX has been used to address SCAs [33], [40], [41] with noteworthy caveats. The developer will need to partition the program into sections that are small enough to fit within a TSX transaction, which is inflexible and difficult to achieve. Additionally, each TSX transaction must be restarted from the beginning after a transactional abort, while executions can be resumed at the point where the event was detected under SENSE. Furthermore, TSX is only available on Intel platforms and was not originally motivated by SCAs, thus not applicable for monitoring microarchitectural events other than cache evictions, while SENSE, being agnostic to processor architecture and microarchitecture, can be easily extended to other cache events or events of other microarchitectural components.

Despite the limitations of TSX, we consider TSX as an existing technique to compare with SENSE on mitigating cache SCAs. We conduct rounds of AES encryption both under SENSE protection and within TSX transactions and compare the performance overhead. The results are shown in Fig. 14. Due to the higher chance of evictions induced by more encryptions, more TSX aborts and SENSE notifications are triggered, thus the decreased performance. However, the performance of SENSE decreases more slowly than that of TSX. One reason is that instead of retrying from the beginning after a TSX abort, SENSE handles the event *in place* and restores the operation at the interrupted location. This makes SENSE more scalable than TSX to mitigate cache-based SCAs.

E. On-demand Loading of Cryptographic Functions

We demonstrate that upon a notification, the event handler is able to replace the functions that are optimized but vulnerable

AES T-table	Function switch	AES no T-table
2443	76	5195

TABLE II: Performance impact of replacing cryptographic functions. Values are in CPU cycles. The switching overhead is around 1.46% compared to encryption operations, and is only paid once.

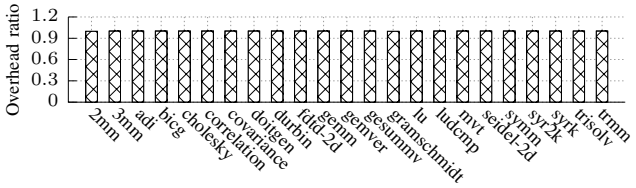

Fig. 15: Performance overhead of the Action Module (AM), when verifying cache coloring. Cache size is 32kB to incur more events.

to SCAs with the secure versions for subsequent operations while only paying a small overhead to perform the switching. We first collect the execution time of two OpenSSL AES encryption implementations (i.e., with and without T-table), as well as the overhead of replacing the vulnerable one with the one without the T-table, which is considered secure. As shown in Table II, compared to the execution time of the library functions, the overhead of function switching is negligible.

The performance of AES encryption with the T-table protected by SENSE becomes comparable to that of AES encryption without the T-table (i.e., 5195 cycles) when experiencing about seven malicious evictions on T-table entries, extrapolated in Fig. 13. Therefore, to maintain good security and performance levels, the event handler can perform the function switch when observing more than seven evictions on the protected entries, thus only paying the performance overhead of the secure version when handling events becomes too expensive.

F. Verification of cache coloring

We enable the event handler that verifies cache coloring on PolyBenchC programs as well to analyze the performance overhead of such a verification. The identity of the thread evicting the monitored cacheline is supplied to the enclave stack by the CPU. Then, the handler reads the corresponding data from the stack and checks whether the thread that caused the cache eviction matches its own thread identity. Similar to previous experiments, we use a cache size of 32kB to make the behaviors of the handler more observable. The performance overhead of cache color verification is negligible, as shown in Fig. 15.

G. Hardware and Memory Overhead

We implemented the RTL [81] for SENSE and evaluated its hardware overhead for cache events, following previous works [27], [30]. The hardware logic overhead mainly stems from the SENSE cache controller logic in SM and the SENSE Fault interrupt controller logic in NM. Compared to an 8-core Xeon Nehalem [82] of 2,300,000,000 transistors, the logic overhead of SENSE is minor, estimated at 0.1%. The logic overhead can be further reduced if the target platform support updating the interrupt controller by firmware or microcode, which do not incur hardware modifications. The memory overhead includes the additional register components per logical core and two bits per cache entry for tracking SENSE status. When targeted platforms possess unused reserved registers and register bits, which is a common case in modern processors, the SENSE register hardware overhead can be eliminated. When combined with existing isolation-based techniques, SENSE can significantly reduce additional hardware overhead even further, as discussed in §IV-D.

VII. DISCUSSION

Limitations. First, the size of the monitored data is bounded by the size of the underlying microarchitecture. For example, if the size of data monitored for cache eviction events exceeds the size of the cache, prefetching the cachelines under SENSE will always evict another cacheline, causing a deadlock. Therefore, knowledge of the target platform is needed when deploying SENSE applications. Second, interaction between SENSE blocks and code outside SENSE blocks may introduce side effects. For example, SENSE will not track the memory event when the monitored memory is copied into another memory region. That is, SENSE applies only to the original instance of the memory in the microarchitecture. Taint-based approaches might be adopted to propagate the monitoring status outside SENSE blocks.

Defending other SCAs. SENSE is most practical for side channels that are stateful, such as TLBs and caches (in this paper), which are the most widely researched side channels. SENSE could also be adapted to detect controlled-channel (i.e., architectural) SCAs such as A/D-bit assists [57] and page faults [23]. There is currently no way for an SGX enclave, for example, to detect when an A-bit assist has occurred. We leave those events for future work. We admit that port contention [83], [84], on the other hand, is challenging for SENSE to mitigate as it occurs at very high frequency, and thus the event notifications might happen so frequently that software cannot make forward progress.

Updating SENSE. To update SENSE, developers only need to extend the Subscription Module (SM) for new microarchitectural events and handler designs, as the Notification Module (NM) is not specific to microarchitectural components and all notifications are reprsented as SENSE Faults. Developers should first identify event paths in microarchitectural components and incorporate them into the SM. For instance, adding a speculative execution event requires including branch predictors and caches. Developers must then insert a SENSE status marker in the component to indicate SENSE monitoring and ensure correct notifications. This may involve extending branch predictors and caches with an extra bit for monitoring status. Developers should also follow event paths to implement monitoring status propagation and define detailed event information to create effective handlers in userspace TEEs. For example, providing handlers with branch prediction history and prediction results can help users detect signs of malicious branch predictor training.

VIII. RELATED WORK

We categorize cache side-channel defenses into two broad classes: 1) isolation-based and 2) detection-based. In this section, we focus only on the most relevant works to SENSE.

A. Isolation-based Defenses

Partition-based defenses. The partitioning-based defenses propose new cache architectures that allocate cache resources (cachelines or ways) exclusively to protected domains (e.g.,

TEEs) [28]–[31], [61], [85], [86]. These defenses can result in cache underutilization when assigned cache ways are not evenly used by a protected domain, as the unused cachelines are blocked for all other domains on the system. Other approaches [27], [87] are more flexible, as they partition the cache on a cacheline basis. However, they still do not provide strong security guarantees against occupancy-based attacks, as they do not enforce strict partitioning. In memory page-coloring schemes [55], [60], [75], [88], the mapping from physical memory addresses to cachelines ensures that cachelines used by sensitive applications do not overlap. One issue with page-coloring is its reliance on OS or hypervisor, which are untrusted under TEEs. Furthermore, the assignment of cachelines is static. SENSE, on the other hand, offers flexible monitoring on exact cache entries that is satisfied upon TEE initialization at runtime. SENSE bypasses the untrusted privileged software as a native interface and does not affect the memory layout.

Randomization-based defenses. Randomization-based defenses employs randomized mapping tables to randomize the mapping of memory lines [87], [89], [90]. Cryptographic randomization techniques [91]–[95] aim to circumvent the storage overhead of large randomized mapping tables by depending on cryptographic primitives to consistently generate randomized mappings. These methods can only diminish the bandwidth of cache attacks rather than completely eradicate them. Attackers can still execute eviction operations when they access a sufficient number of lines across a vast array of cache sets. Furthermore, some strategies may experience significant performance decline when implemented on the considerably larger last-level cache. SENSE, on the other hand, fundamentally eliminates the aforementioned unreliability and inflexibility by providing timely, accurate, and flexible notifications directly to userspace TEEs.

B. Detection-based Defenses

Detection-based defenses can be primarily divided into two categories: signature-based [35], [50], [96]–[99] and anomaly-based [100]–[102]. Some methods [34], [38], [103] utilize a combination of both signature- and anomaly-based detection techniques.

Signature-based. Demme et al. [96] employ L1 hits events in Hardware Performance Counters (HPCs) to detect malware and cache SCAs. Allaf et al. [97] suggest another signature-based detection technique to identify Prime+Probe and Flush+Reload attacks using machine learning (ML) models and HPCs while running an AES cryptosystem. The hardware events utilized include core cycles, reference cycles, and core instructions. NIGHTs-WATCH [98] can detect access-driven cache SCAs using various ML models that leverage LLC misses and CPU cycles in HPCs. This method trains the model under specific system loads, but it is unclear whether it performs well under unknown system loads. Mushtaq et al. [99] apply linear and non-linear ML classifiers to detect Prime+Probe attack variants running under the AES cryptosystem. HexPADS [35] uses cache miss rates and page fault values to detect Flush+Reload and cache template attacks [50].

Anomaly-based. CacheShield [100] is an anomaly-based detection mechanism for legacy software (victim application) that monitors LLC cache misses using HPCs. Bazm et al. [101] detect cross-VM cache SCAs by utilizing hardware fine-grained information provided by Intel Cache Monitoring Technology (CMT) and HPCs, following the Gaussian anomaly detection method. SpyDetector [102] can identify Flush+Reload, Flush+Flush, and Prime+Probe attacks running on RSA, AES, and ECDSA cryptosystems by monitoring L3 cache and L1 data cache through HPCs.

Signature and anomaly-based. Chiappetta et al. [34] propose a machine learning-based detection mechanism for Flush+Reload attacks on AES and ECDSA cryptosystems, monitoring L3 access to detect attacks. CloudRadar [38] is a signature and anomaly-based detection system that detects attacks in two steps. The first step involves identifying cryptographic applications using branch instructions and dynamic time warping. The second step defines a criterion for distinguishing between benign and malicious programs. CloudRadar considers an attack to have occurred when the detected value exceeds this criterion. Alam et al. [103] present a multi-layer detection approach based on machine learning, which collects microarchitecture events (e.g., branch misses, LLC accesses, and LLC misses) during attacks. They train machine learning models based on these events to detect attacks.

The aforementioned works rely on microarchitectural events as feature vectors for detection. However, due to the limited capacity of microarchitecture components, they are highly susceptible to interference from system loads. These heuristic approaches lack robustness and tend to experience high false positives and false negatives. In contrast, SENSE detects potentially malicious behaviors at their exact locations and promptly notifies userspace TEEs, preventing the delayed awareness of potential attacks experienced in signature and anomaly-based strategies. Furthermore, SENSE enables flexible responses to suspicious behaviors, thanks to the rich microarchitectural information it provides.

IX. CONCLUSION

Current side-channel attack (SCA) detection techniques within Trusted Execution Environments (TEEs) exhibit various limitations due to the absence of direct access to precise microarchitectural information and the lack of flexible methods to respond to potential SCAs inside TEEs. In this paper, we introduce SENSE, an innovative interface that empowers TEE programs to directly subscribe to microarchitectural event notifications and actively manage these events using userspace handlers. We present a comprehensive design of SENSE for cache and conduct an in-depth security analysis to demonstrate its robustness. Our evaluation reveals that SENSE effectively mitigates side-channel attacks while maintaining minimal performance overhead.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers and our shepherd for their helpful feedback. This research was funded by a research gift from Intel.

REFERENCES

[1] B. W. Lampson, "Lazy and speculative execution in computer systems," in *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, 2008, pp. 1–2.

[2] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' Track at the RSA Conference*, 2006.

[3] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[4] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[5] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[6] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[7] W. He, W. Zhang, S. Das, and Y. Liu, "Sgxlinger: A new side-channel attack vector based on interrupt latency against enclave execution," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 108–114.

[8] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," in *Proceedings of the 23st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Williamsburg, VA, Mar. 2018.

[9] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazalch, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, Taiwan, Oct. 2016.

[10] ——, "Covert channels through branch predictors: A feasibility study," in *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2015.

[11] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2019.

[12] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[13] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 2016/086, 2016, http://eprint.iacr.org/2016/086.pdf.

[14] Intel, "Intel Trust Domain Extensions (TDX)," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[15] AMD, "AMD Secure Encrypted Virtualization (SEV)," https://developer.amd.com/sev/.

[16] ARM, "Building a Secure System using TrustZone Technology," 2009.

[17] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, BC, Canada, Aug. 2017.

[18] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Stealing intel secrets from sgx enclaves via speculative execution," in *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[19] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT)*, Baltimore, MD, Aug. 2018.

[20] D. Moghimi, J. V. Bulck, N. Heninger, F. Piessens, and B. Sunar, "CopyCat: Controlled Instruction-Level attacks on enclaves," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual, Aug. 2020.

[21] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[22] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[23] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[24] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*, 2017.

[25] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[26] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct.–Nov. 2016.

[27] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid Side-Channel-Resilient caches for trusted execution environments," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual, Aug. 2020.

[28] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021.

[29] D. Townley, K. Arıkan, Y. D. Liu, D. Ponomarev, and O. Ergin, "Composable cachelets: Protecting enclaves from cache Side-Channel attacks," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[30] G. Dessouky, A. Gruler, P. Mahmoody, A.-R. Sadeghi, and E. Stapf, "Chunked-cache: On-demand and scalable cache isolation for security architectures," in *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Apr. 2022.

[31] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "CURE: A security architecture with CUstomizable and resilient enclaves," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual, Aug. 2021.

[32] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Untangle: A principled framework to design low-leakage, high-performance dynamic partitioning schemes," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vancouver, Canada, Mar. 2023.

[33] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[34] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494616304732

[35] M. Payer, "Hexpads: A platform to detect "stealth" attacks," in *Engineering Secure Software and Systems*, 2016.

[36] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Cambridge, UK, Dec. 2014.

[37] M. Yan, Y. Shalabi, and J. Torrellas, "Replayconfusion: Detecting cache-based covert channel attacks using record and replay," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, Taiwan, Oct. 2016.

[38] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 118–140.

[39] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jul. 2018.

[40] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with Déjá Vu," in *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Abu Dhabi, UAE, Apr. 2017.

[41] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[42] J. Jiang, C. Soriente, and G. O. Karame, "On the challenges of detecting side-channel attacks in sgx," Oct. 2022.

[43] F.-X. Standaert, T. Malkin, and M. Yung, "A formal practice-oriented model for the analysis of side-channel attacks," 2006.

[44] M. S. Taha, M. S. M. Rahim, S. A.-S. Lafta, M. M. Hashim, and H. M. Alzuabidi, "Combination of steganography and cryptography: A short survey," *IOP Conference Series: Materials Science and Engineering*, vol. 518, 2019.

[45] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[46] O. Acıiçmez and W. Schindler, "A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl," in *Cryptographers' Track at the RSA Conference*. Springer, 2008, pp. 256–273.

[47] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+abort: A timer-free high-precision l3 cache attack using intel tsx," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[48] B. Gras and K. Razavi, "Aslr on the line: Practical cache attacks on the mmu," Feb.–Mar. 2017.

[49] 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, pp. 279–299.

[50] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[51] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2016.

[52] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[53] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[54] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, 2017.

[55] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Truspy: Cache side-channel information leakage from the secure world on arm devices," 2016.

[56] J. Bonneau and I. Mironov, "Cache-collision timing attacks against aes," in *Cryptographic Hardware and Embedded Systems (CHES)*. Berlin, Heidelberg: Springer, 2006, pp. 201–215.

[57] 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 *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[58] Intel, "Deprecated technologies — 12th generation intel® core™ processors," [Online; accessed 2-Oct-2022]. [Online]. Available: https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/009/deprecated-technologies/

[59] ——, "Performance monitoring impact of intel transactional synchronization extension memory ordering issue," 2021. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/604224

[60] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[61] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys'20, 2020.

[62] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[63] D. Kaplan, "{AMD} x86 memory encryption technologies," 2016.

[64] S. Pinto and N. Santos, "Demystifying arm trustzone," *ACM Computing Surveys (CSUR)*, vol. 51, pp. 1 – 36, 2019.

[65] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*, 1996.

[66] I. Biehl, B. Meyer, and V. Müller, "Differential fault attacks on elliptic curve cryptosystems," in *Annual International Cryptology Conference*, 2000.

[67] A. Tang, S. Sethumadhavan, and S. Stolfo, "Clkscrew: Exposing the perils of security-oblivious energy management," in *USENIX Security Symposium*, 2017.

[68] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0ltpwn: Attacking x86 processor integrity from software," in *USENIX Security Symposium*, 2019.

[69] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies," *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[70] "Stack frame layout on x86-64," 2011, https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64.

[71] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: A dynamic cache partitioning system using page coloring," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014.

[72] S. Perarnau, M. Tchiboukdjian, and G. Huard, "Controlling cache utilization of hpc applications," in *International Conference on Supercomputing (ICS)*, 2011.

[73] Y. Yarom, "Mastik: A micro-architectural side-channel toolkit," https://github.com/0xADE1A1DE/Mastik.

[74] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[75] T. Kim, M. Peinado, and G. Mainar-Ruiz, "StealthMem: System-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[76] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2011, pp. 194–199.

[77] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," ser. CCSW '09. New York, NY, USA: Association for Computing Machinery, 2009.

[78] D. Gruss, "flush_flush," 2019, https://github.com/IAIK/flush_flush.

[79] "PolyBench," 2015, http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[80] W. Kim, "Fun with Intel Transactional Synchronization Extensions," 2013, https://software.intel.com/en-us/blogs/2013/07/25/fun-with-intel-transactional-synchronization- extensions.

[81] Pulp-Platform, "Cva6 risc-v cpu," https://github.com/openhwgroup/cva6.

[82] Intel, "Intel xeon processors," 2009, https://www.intel.com/content/www/us/en/products/details/processors/xeon.html.

[83] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port contention for fun and profit," May 2019.

[84] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," Jun. 2019.

[85] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: Secure dynamic cache partitioning for efficient timing channel protection," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[86] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 974–987.

[87] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, San Diego, CA, Jun. 2007.

[88] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proceedings of the 22nd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Barcelona, Spain, Feb. 2016.

[89] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[90] F. Liu and R. B. Lee, "Random fill cache architecture," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 203–215.

[91] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[92] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 775–787.

[93] ——, "New attacks and defense for encrypted-address cache," in *Proceedings of the 46th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Phoenix, AZ, Jun. 2019.

[94] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[95] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "Phantomcache: Obfuscating cache conflicts with localized randomization," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[96] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[97] Z. Allaf, M. Adda, and A. E. Gegov, "A comparison study on flush+reload and prime+probe attacks on aes using machine learning approaches," in *UK Workshop on Computational Intelligence*, 2017.

[98] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapôtre, and G. Gogniat, "Nights-watch: a cache-based side-channel intrusion detector using hardware performance counters," *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2018.

[99] M. Mushtaq, A. Akram, M. K. Bhatti, R. N. B. Rais, V. Lapôtre, and G. Gogniat, "Run-time detection of prime + probe side-channel attack on aes encryption algorithm," *2018 Global Information Infrastructure and Networking Symposium (GIIS)*, pp. 1–5, 2018.

[100] S. Briongos, G. I. Apecechea, P. Malagón, and T. Eisenbarth, "Cacheshield: Detecting cache attacks through self-observation," *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018.

[101] M.-M. Bazm, T. Sautereau, M. Lacoste, M. Südholt, and J.-M. Menaud, "Cache-based side-channel attacks detection through intel cache monitoring technology and hardware performance counters," *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 7–12, 2018.

[102] Y. Kulah, B. Dinçer, C. Yilmaz, and E. Savaş, "Spydetector: An approach for detecting side-channel attacks at runtime," *International Journal of Information Security*, pp. 1–30, 2019.

[103] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 564, 2017.

APPENDIX A
IMPLEMENTATION DETAILS OF THE INVARIANT HANDLER

An example of the INVARIANT handler is shown in Fig. 16. By calling the _sswatch() function with the security-critical data mem and an event type cache_ev, mem is put into the *watch set* for cache eviction events. Upon eviction on mem under SENSE mode, the *watch set* is *refetched* by calling the _sswatch() on all *watch set* members, thus preserving the cache residency of mem.

APPENDIX B
MINIMIZING IMPACT OF SENSE ON CPU EXECUTION PIPELINE

Out-of-order execution. An SENSE Fault takes place in the form of a hardware fault (§IV-B). Whenever a hardware fault (e.g., x86) is detected, the *commit* stage in the CPU execution pipeline squashes the instructions fetched under the mispredicted speculative execution or when executed out of order. Related hardware resources such as the instruction queue and load/store buffers are cleaned up afterward. Therefore, no additional hardware logic is required to clean up microarchitectural execution states while handling SENSE Faults. The adoption of SENSE to modern processors thus does not incur major hardware changes on the out-of-order execution pipeline.

Speculative execution. The security guarantee of SENSE is provided by the fact that the enclave process is notified as soon as the subscribed cache entries are evicted during execution of the instruction under CPU SENSE mode. However, when SENSE-monitored cache entries are evicted under incorrect speculations, the incurred SENSE Fault (i.e., a CPU hardware fault) will be squashed at the *commit* stage and the event handler will not be invoked, leaving the microarchitectural trace in the cache. This allows an attacker to speculatively evict SENSE-monitored entries without causing SENSE notifications.

To protect SENSE from the above-mentioned attacks, we perform an instruction serialization when the CPU enters SENSE mode, which is a common feature in modern CPU architectures. Instruction serialization forces the processor to complete all prior instructions and drain all buffered writes to memory before the next instruction is fetched and executed. Therefore, instructions issued before the SENSE block cannot speculatively evict the cache entries in the SENSE block. Gem5 allows one to define a serializing instruction by setting the serializingBefore flag using the .serialize_after micro operation. To enable the serializingBefore flag for ssbegin, we insert the .serialize_after micro operation at the beginning of the ssbegin instruction. When the serializingBefore flag is encountered, the rename stage of the CPU execution pipeline stalls until the *reorder buffer* becomes empty, i.e., all prior instructions are committed. After ssbegin is finished, the CPU executes under SENSE mode without further serialization constraints, allowing the enclave process to be executed with performance optimizations. As a result, instruction serialization effectively prevents speculative execution SCAs within SENSE blocks, while maintaining good performance.

```
void _sswatch(void *ptr, size_t size, ss_t type) {
  /* Add [ptr, size, type] tuple
       to the watch set if not found. */
  watch_set.add([ptr, size, type]);
  sssub(ptr, size, type);
}

void _ssunwatch(void *ptr) {
  /* Remove ptr in the watch set if found. */
  watch_set.remove(ptr);
  ssunsub(ptr, size, type);
}

void handler(int ss_info) {
  _ssbegin(handler);
  for (t in watch_set)
    _sswatch(t->ptr, t->size, t->type);
}

/* _ssbegin(handler) by TEE
       at enclave entry */
void enclave_program() {

  char mem[80];
  /* Subscribe to target memory. */
  _sswatch(mem, 80, cache_ev /* event type */);
  ...
  _ssunwatch(mem);

}
/* _ssend() by TEE
       at enclave termination */
```

Fig. 16: An example of state-pinning model.

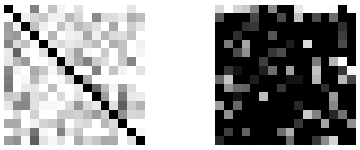

Fig. 17: Color matrices showing cache hit patterns of the first AES T-Table ($k_0 = 0x00$) under Flush+Reload attack. Left: attack without SENSE; Right: attack with SENSE cache INVARIANT handler.

APPENDIX C
HARDEN AES T-TABLES UNDER FLUSH+RELOAD

Fig. 17 shows that the T-table without SENSE protection reveals a clear cache hit pattern under Flush+Reload attacks, while the T-table protected by SENSE does not leave useful microarchitectural traces for the attacker to infer secret key bits.