TEE-SHirT: Scalable Leakage-Free Cache Hierarchies for TEEs

Kerem Arıkan[*], Abraham Farrell[*], Williams Zhang Cen[*], Jack McMahon[*], Barry Williams[*], Yu David Liu[*], Nael Abu-Ghazaleh[†], and Dmitry Ponomarev[*]

[*]Binghamton University
[†]University of California, Riverside

Abstract—**Protection of cache hierarchies from side-channel attacks is critical for building secure systems, particularly the ones using Trusted Execution Environments (TEEs). In this paper, we consider the problem of efficient and secure fine-grained partitioning of cache hierarchies and propose a framework, called Secure Hierarchies for TEEs (TEE-SHirT). In the context of a three-level cache system, TEE-SHirT consists of partitioned shared last-level cache, partitioned private L2 caches, and non-partitioned L1 caches that are flushed on context switches and system calls. Efficient and correct partitioning requires careful design. Towards this goal, TEE-SHirT makes three contributions: 1) we demonstrate how the hardware structures used for holding cache partitioning metadata can be effectively virtualized to avoid flushing of cache partition content on context switches and system calls; 2) we show how to support multi-threaded enclaves in TEE-SHirT, addressing the issues of coherency and consistency that arise with both intra-core and inter-core data sharing; 3) we develop a formal security model for TEE-SHirT to rigorously reason about the security of our design.**

I. Introduction

Cache hierarchies are the target of many recent side-channel attacks that leak critical information from systems [4], [11], [19], [29]–[31], [36], [43], [45], [49]–[52], [59], [64], [65], [70], [72], [74]. These attacks enable a malicious process to infer secret information about a victim process by observing the performance of its memory accesses as they interact with a shared cache. Side-channel information leakage also enables dangerous transient execution attacks [42], [43], [64], [70], where the attacker forces speculatively accessed secrets to be exposed through cache side-channels. Cache-based side-channel attacks also compromise Trusted Execution Environments (TEEs) such as Intel SGX [17]. While TEEs provide logical isolation, they are still vulnerable due to the physical sharing of resources. Indeed, a number of recent cache-based attacks have been demonstrated against SGX [11], [19], [29], [50], [51], [64]. It is therefore critical to integrate leakage-free cache hierarchies into TEEs, augmenting the logical isolation with physical isolation to eliminate side-channel leakage.

In this paper, we investigate cache partitioning mechanisms for TEE systems with the goal of protecting the entire cache hierarchy, and not just a single cache level. Cache partitioning is a principled approach to security that physically isolates applications from each other eliminating leakage due to contention on shared resources. Since cache partitions belonging to different applications (or enclaves in TEE systems) are isolated, the behavior of the victim process does not impact any cache-related observations by attackers, making attacks impossible. Existing secure cache partitioning schemes consider only a single level of caches, either private upper-level caches [23] or shared LLC [41], [44], [56], [63]. These schemes partition caches by ways [23], [41], sets [21], [56], or both [63]. Without loss of generality, we study fine-grained approaches that partition caches by both ways and sets [63].

Various levels of the cache hierarchy require different approaches to achieve security. It has been established that cache partitioning is an effective approach for shared last-level caches [56], [63]. At the same time, prior research demonstrated that L1-caches can be flushed on context switches to prevent leakage with minimal loss in performance [28]. In this paper, we also make a case that private L2 caches have to be partitioned, and present TEE-SHirT, a security framework for multi-level cache hierarchies that combines a shared partitioned LLC, private partitioned L2 caches, and private L1 caches that are flushed on context switches and system calls. Partitioning private caches introduces performance, consistency, and coherence challenges, especially with multi-threaded workloads. To ensure high performance, correctness, and security of the entire memory hierarchy with TEE-SHirT, this paper makes the following key contributions.

First, to avoid scalability limitations due to the limited capacity of cache partitioning logic, we propose to *virtualize* the cache partitioning metadata to enable partitions to be tracked and maintained even when their respective enclave is not actively running. Specifically, we integrate cache partitioning metadata into an enclave context that can be saved and restored on context switches using mechanisms available in Intel SGX. We augment existing SGX data structures to include information about cache partitioning metadata, making partitioned caches an integral part of the SGX ecosystem. Once this support is established, the number of enclaves simultaneously sharing the cache space is limited only by

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.2024.24390
www.ndss-symposium.org

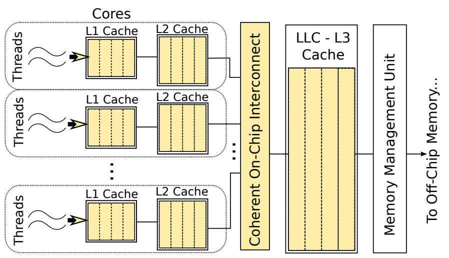


Fig. 1: A conventional 3-level cache hierarchy.

the total cache capacity, and not by the number of physical instances of hardware structures used to support partitioning.

Second, we address new consistency and coherence problems with multi-threaded enclaves when they use multiple levels of partitioned caches. When multiple threads of an enclave are scheduled on the same core, multiple partitions (one for each thread) for the same enclave can be formed within a single private L2 cache. Maintaining coherence of data within the same cache is not supported by traditional cache coherence protocols since they assume that at most a single copy of a cache line exists in each cache. Furthermore, it can also be possible that some private caches can maintain the most recent copy of shared data, but the partitioning metadata of the enclave that produced this data while executing on that core is currently context-switched out. In this case, the cache coherence protocol would be missing the metadata to perform the cache query correctly and will need to be augmented.We describe our solutions to both of these problems to provide a coherent and consistent memory system in the presence of shared and private partitioned caches.

Third, we prove security guarantees of TEE-SHirT through a formal security analysis that is based on cache-aware and enclave-aware operational semantics to account for the allocation across multiple levels of cache. Our analysis accounts for enclave behavior in a variety of settings, including enclave creation and destruction, and a full consideration of cache coherence and context switch.

We evaluated the performance of TEE-SHirT using various benchmarks, including MiBench [34], SPEC2017 [12], and WolfSSL [2] functions, using gem5 [7] cycle-accurate microarchitectural simulator. To estimate the area overhead of TEE-SHirT, we implemented parts of our design within the open-source cache subsystem of the open ESP project [46]. We integrated a prototype of TEE-SHirT with a 4-core CPU in an open-source System on Chip (SoC) platform, demonstrating an overhead of less than 2% relative to the baseline system.

II. Background and Threat Model

In this section, we provide background on modern cache hierarchies and the concept of cache partitioning for security, overview relevant components of Intel SGX's ecosystem, and describe our threat model.

A. Cache Hierarchies

Figure 1 shows a typical three-level cache hierarchy of a modern CPU. Each core has private L1 and L2 caches, and the L3 cache (also called LLC interchangeably) is shared among all cores/threads running in the system. This is representative of recent designs from major CPU manufacturers [13], [24], [35], [38], [61]. While L1 and L2 caches can be shared by threads simultaneously running on the same core in a simultaneously multithreaded (SMT) processor, we do not consider SMT in this paper — it is often disabled for security. However, even if SMT is used, private caches can be statically partitioned by ways across threads to provide isolation. Note that the number of simultaneous threads is typically smaller than the number of cache ways at the levels of private caches. If such provisions are implemented, then the rest of our design applies even to SMT systems (each thread works within its own partition). In some designs, L2 caches are shared by multiple neighboring cores. While we do not explicitly evaluate such systems (due to the limitation of our gem5-based simulation infrastructure), our partitioning principles apply to them as well.

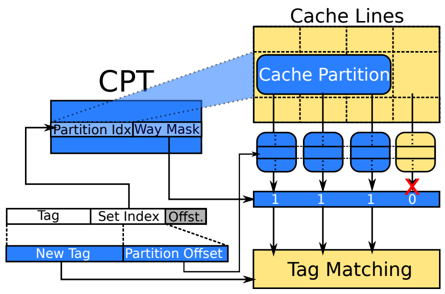


Fig. 2: Enclave address remapping in partitioned caches

B. Partitioned Caches for Security

We consider TEE-SHirT in the context of hardware-supported cache partitioning schemes, in particular in the context of Composable Cachelets [63]. The key idea is to create isolated cache partitions (CPs) for each enclave running on a TEE system. Each CP is composed of the number of consecutive sets and one or more ways. At a high level, a hardware component is added to the cache circuitry to control accesses to CPs. This component is called the Cache Partition Table (CPT) and it holds the metadata related to cache partitions. Specifically, this metadata includes the mappings between the original set indices to the indices within allocated CPs, and the way mask to indicate which cache ways belong to this partition. This information is maintained at the granularity of partition sizes in terms of the number of sets. By going through the CPT, the original memory address generated by an enclave can be remapped to a cache location within the allocated CP.

Figure 2 shows the address remapping mechanism for cache accesses from enclaves. Entries of the CPT are accessed with the higher bits of the set index. The accessed CPT entry's partition index points to a group of sets that belong to the enclave's CP. Depending on the desired partition size and shape, one or more of the CPT entries can be established by an enclave, thus appropriately deflecting the initial addresses to physical cache indexes within the boundary of the allocated CP. The set index's lower bits are used as *partition offset*, where they point to the specific set within the CP. The way mask determines which cache ways belong to this CP and therefore should be checked. For example, in Figure 2, the way mask is set as `1110`, which denotes that ways zero, one, and two belong to the CP being accessed, and way three does not (in a 4-way cache).

C. Relevant Intel SGX Data Structures

One of our contributions is to build support for saving cache partitioning metadata on context switches and system calls. We demonstrate it using Intel SGX as a target TEE. In this section, we provide relevant background on SGX and its data structures.

In the memory layout of SGX, enclaves' metadata is stored in a dedicated data structure called Enclave Page Cache (EPC) [17] within the Processor Reserved Memory (PRM). SGX also deploys an architectural layer of integrity checks by keeping a set of security records called the Enclave Page Cache Map (EPCM). Each entry of the EPCM contains a pointer to each EPC page, as well as the page type and permissions of the corresponding EPC pages.

The EPC includes several pages that contain enclave metadata. The SGX Enclave Control Structure (SECS) page maintains a data structure that has basic information about an enclave such as its size, Enclave ID (EID), and base address. To support multithreaded enclaves, SGX maintains an EPC page called the Thread Control Structure (TCS). The TCS mostly contains the offsets of various EPC pages of the corresponding thread (code pages, execution context, etc.). To support context switches, SGX saves the enclave's execution context to a set of pages called the Save State Area (SSA) upon a hardware exception. Each thread's SSAs can be composed of several pages called the SSA Frame Size. The SSA Frame Size is held in SECS, while the offset of the last SSA page is pointed to by the TCS. All of these structures are mostly controlled by the hardware, mainly the Memory Management Unit (MMU).

D. Threat Model

TEE-SHirT can be used to protect any program that uses it, but we use a threat model that assumes a more powerful adversary similar to the threat model of SGX [17]. Specifically, we assume that the system software, including OS and hypervisors, is untrusted and that the attackers can exploit any mechanisms available within these privileged software layers to amplify their attacks. We also assume that the attacker can mount any cache side-channel attack strategy such as *Prime+Probe* [39], [45]. Importantly, TEE-SHirT protects from situations where simultaneous attacks on multiple cache levels are possible and the attacker can leverage multiple threads (or enclaves) executing on multiple cores in the system, including the core on which the victim application executes.

While we do not directly address denial-of-service attacks, we limit the cache space that can be allocated to enclaves by reserving at least one or more ways for non-enclave programs. This reservation also has the important effect of allowing non-enclave programs to access the cache without going through the process of index remapping since at least one way is always available for non-enclave programs in every set.

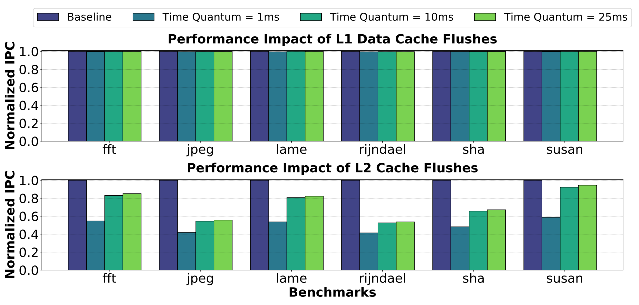


Fig. 3: Performance impact of private cache flushes in inclusive hierarchies

III. A Case for Partitioning L2 Caches in Inclusive Hierarchies

Previous efforts convincingly established that partitioning of the LLC and flushing of private L1 caches are attractive solutions for securing those cache levels [10], [28]. In this section, we explain in more detail why private L2 caches should also be partitioned. Flushing a non-partitioned L2 on context switches will have a more significant impact on performance due to the larger size and higher miss latencies. To demonstrate the performance impact of flushing L1 and L2 caches, we performed experiments with six encryption benchmarks from MiBench suite [34]. We account for invalidation and writeback operations by blocking the cache until they are completed. We also implement back-invalidations to ensure inclusivity. Figure 3 shows the normalized commit IPC (Instructions per Cycle) metric for scenarios where the L1 data cache and the L2 cache are flushed on context switches with three different time quantums. As observed, with L1 data cache flushes, IPC impact does not exceed 2% for all benchmarks, whereas the L2 cache flushes cause between 6% and 53% slowdown for 25ms time quantum. These results reassure the notion of L2 cache flushes being infeasible for performance.

Furthermore, if an L2 cache remains non-partitioned, another performance and scalability problem arises in a system with inclusive cache hierarchies. Inclusive cache hierarchies ensure that data that resides in an upper-level cache (say, the L2) also exists in a lower-level cache (say, the LLC). This organization simplifies cache coherence hardware because the absence of data in the LLC also indicates that the data is not in the private caches, thus avoiding unnecessary snoops and providing snoop filtering capabilities. Inclusive cache hierarchies are commonly used in modern CPUs [33]. To support inclusivity, the key additional mechanism used in such systems is *back-invalidation*, where the eviction of data from the LLC also caused the eviction of this data from all upper-level private caches (note that the term "upper-level" refers to caches that are closer to the CPU).

In traditional cache hierarchies, the L3 cache (LLC) has a larger size than the L2 cache. However, let us consider a cache hierarchy where the L3 cache is smaller than the L2 cache in an inclusive cache hierarchy. A simplified example

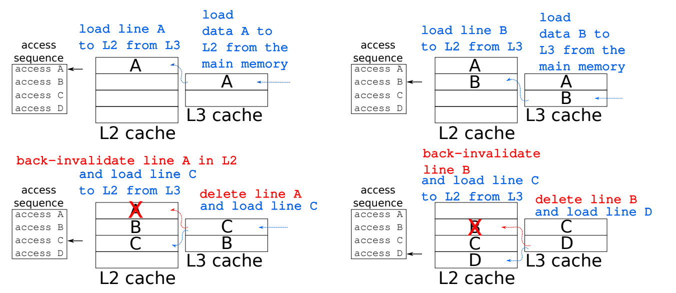

access sequence
access A
access B
access C
access D
load line A to L2 from L3
load data A to L2 from the main memory
A
L2 cache
L3 cache
load line B to L2 from L3
load data B to L3 from the main memory
B
back-invalidate line A in L2 and load line C to L2 from L3
delete line A and load line C
C
back-invalidate line B and load line C to L2 from L3
delete line B and load line D
D

Fig. 4: An example of back-invalidation in inclusive cache hierarchy

of this scenario is depicted in Figure 4, which represents a cache hierarchy featuring fully associative L2 and L3 caches with 4 and 2 lines respectively. The program accesses lines A and B, which are loaded into both L2 and L3 caches. Later, when line C is accessed, the least recently used line in the L3 cache (in this instance, line A) is evicted. As a result, line A must also be back-invalidated in the L2 cache to sustain the inclusivity with the L3 cache. The same sequence of events occurs for line B when line D is accessed. As a result, the effective capacity of L2 degenerates to that of L3 (2 lines).

Now let us project this example to a cache partitioning system. Exactly this situation would happen (on a larger scale) if the L3 cache is partitioned, the L2 cache is not, and the size of the allocated L3 partition is less than the size of the L2 cache. As a result, if only the LLC is partitioned, then either the performance suffers due to the effective reduction of the L2 capacity, or allocated partition sizes in the LLC are constrained to be larger than the L2 size. Neither of these scenarios is a desirable outcome in a high-performance and scalable TEE system.

IV. TEE-SHirT

Following the arguments presented above, TEE-SHirT employs partitioned LLC and L2 caches, and a non-partitioned L1 that is flushed on time-driven context switches and system calls. We now describe the architecture of TEE-SHirT, address its complexity, correctness, and performance challenges, and develop a formal model to demonstrate security.

A. *TEE-SHirT Design Overview*

TEE-SHirT partitions private L2 and shared L3 caches into smaller CPs. The partition size at every cache level is controlled independently depending on the size of the caches and the performance demands of the application. Enclaves can request the allocations of CPs through additional instructions. Though specific details are beyond the scope of the paper, some possible solutions are described by prior work [63].

The high-level architecture of a multi-core system with TEE-SHirT is illustrated in Figure 5. In this example, the system has four concurrent enclaves, with Enclave 0 and Enclave 1 currently executing on Core 0 and Core 1, respectively. L1 caches are flushed on context switches and system calls. Thus, they do not need additional support for security. However, in the L2 and L3 caches, all four enclaves have allocated

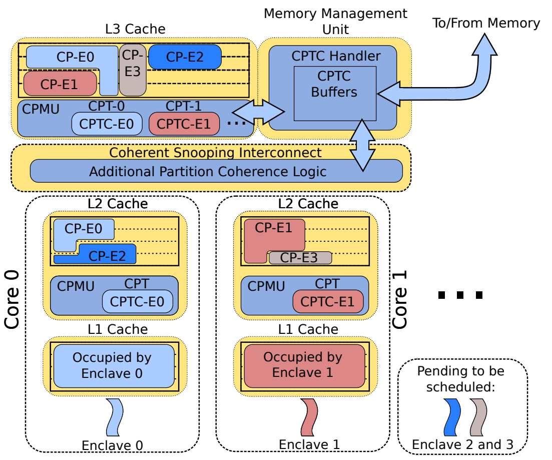


Fig. 5: An overview of the TEE-SHirT architecture

CPs that are isolated from other enclaves and also from non-enclave programs. This system creates separation for different processes in the cache. Notably, programs do not have to be actively running on a core to have partitions in L2 and L3 caches (i.e. Enclaves 2 and 3 in this example). We describe support to enable this feature in detail in Section IV-B1.

To support TEE-SHirT-related operations, every cache level except for L1s is equipped with a Cache Partitioning Management Unit (CPMU). We assume that allocation requests are specified in terms of the desired CP size without specifying a particular shape for the CP. In turn, TEE-SHirT maps allocation requests onto a specific way-set partition of the cache. This partition is represented in the metadata for the CP within the CPT. We refer to this metadata - the contents of the CPT - as Cache Partition Table Context (CPTC). The management of CPTC is done by the hardware and is not visible to the OS. This is implemented through a CPTC Handler that is integrated with the MMU.

When an enclave requests a CP allocation, a physical region within the cache is assigned to it. However, the allocated region is likely to contain data that was placed there by another program prior to the allocation request. If this data is not evicted from the cache, it will no longer be accessible by the original program once a CP is established. As a result, all the dirty data within the allocated region has to be written back to the lower-level structure. We now explain how the boundary is established and how the data is written back and invalidated.

After the CPTC is created, two other activities take place: 1) *CPTC update* when the CPMU sends the newly generated CPTC to main memory to be stored, and 2) *gang-invalidation* which is the invalidation and write-back of existing cache lines within the allocated region. In case an invalidated cache line is dirty, it is also written back. To support inclusivity in inclusive cache hierarchies, the invalidated cache lines are also back-invalidated in upper-level caches.

All cache accesses that are initiated after the CP allocation request must wait until all allocation operations have been completed. Consequently, TEE-SHirT-related requests should be globally serialized with regular cache accesses. During CP creation, the new CP mapping should not take effect until the gang-invalidation process is completed, even though the CPTC has already been created.

Another requirement is that before the gang-invalidation process, the CPMU has to ensure that all pending cache misses are handled. This is because if a cache line with a pending miss status is invalidated before being updated, then the newly received version of the cache line cannot be written to the cache since it is now reserved for an enclave. These operations ensure that consistency is maintained across all the levels of the cache system during CP allocation.

In the rest of this section, we address two key issues in the design and implementation of TEE-SHirT: 1) how to maintain partition data in the caches on context switches and system calls without significant additional hardware complexity; 2) how to support multi-threaded enclaves with partitioned caches.

B. Partition-Aware Context Switches and System Calls

To securely support context switches and system calls under the untrusted OS, leakage in caches has to be carefully considered during these operations. One way to handle this leakage is to flush the contents of enclave cache partitions on every context switch and system call. However, this can get expensive for larger lower-level caches. Ideally, we would like to preserve the CP contents of the enclave in the caches across context switches and especially system calls. However, to do this, the partition metadata (the CPTC) has to also be retained to appropriately link to the corresponding CP. If a large number of enclaves are sharing a system, maintaining multiple CPTCs at the same time in the cache requires significant complexity in the form of an additional number of physical CPTs.

1) CPTC Virtualization: We now address the challenge of supporting the scalable operation, where the number of enclaves that can share the cache subsystem concurrently is only limited by the cache capacity and not by the availability of CPTs or other similar remapping hardware. Partitioning metadata has to be maintained for every enclave. Maintaining this information only in the cache system entails significant hardware complexity or restricts the number of concurrent enclaves that can be running in the system. We now show how this partitioning metadata can be effectively saved into a reserved memory following a timing interrupt-driven context switch or a system call, to be restored when the enclave is rescheduled for execution. We call this scheme *CPTC Virtualization*.

The key idea behind *CPTC Virtualization* is to leverage existing TEE data structures (using Intel SGX as an example) and MMU hardware to maintain partitioning information rather than allocating a separate CPT for each enclave in the system. We accomplish this goal by storing CPTCs in the Processor Reserved Memory (PRM) of Intel SGX and reloading them back only when the corresponding enclave is scheduled to run. Figure 6 contrasts the hardware requirements between the CPT-per-enclave approach and CPTC virtualization. In the following example, there are `n` enclaves running in the system. Keeping CPTCs exclusively in the remapping logic may result

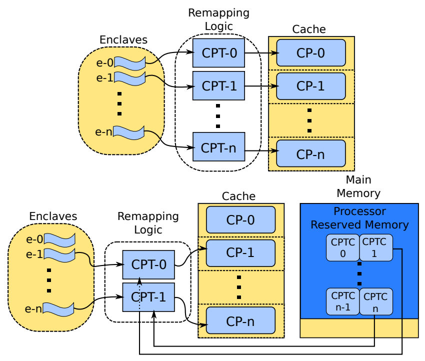


Fig. 6: CPT-per-enclave (top) vs. CPTC virtualization (bottom)

in two possible scenarios when an **`n+1`**-th enclave requests a new CP allocation:

Scenario 1: The lack of available CPTs will result in the enclave waiting until one of the existing enclaves terminates. This approach limits the maximum number of concurrent enclaves to the number of physical CPTs provided by the cache.

Scenario 2: The CPTC and CP of an existing enclave are removed to make space for the **`n+1`**-th enclave, leading to loss of the enclave's cache contents, thus limiting performance.

We address the issue in Scenario 2 as follows. CPTC virtualization requires that a CPTC should be present in a CPT only if the corresponding enclave is executing in the core. If an enclave is context-switched out of the core, its CPTC is removed from the CPT since it is already stored in the PRM memory. And if the program running on the core is a non-enclave program, then the corresponding CPT is unused.

Importantly, note that the contents of an enclave's CP are not invalidated upon a context switch even when its CPTC is removed from the CPT. Therefore, the enclave's data is preserved in the cache, addressing the problem in scenario 2 and resulting in substantial performance improvement in system call-intensive workloads. The CP remains isolated as TEE-SHirT diverts cache accesses from non-enclave processes and other enclaves from this enclave's CPs. Furthermore, this design allows the CPT to load the CPTC when it is needed, which eliminates scenario 1 and reduces the CPTs in the remapping logic by **`n-1`**. We evaluate the impact of the number of CPTs on hardware complexity in Section VI, as well as the performance improvement CPTC virtualization provides.

2) Memory Layout Extensions for TEE-SHirT: TEE-SHirT-related operations must be performed without relying on an untrusted system software. With TEE-SHirT, the Memory Management Unit (MMU) conducts CPTC virtualization with additional hardware. TEE-SHirT does this by extending the SGX-reserved memory areas — the TCS, SSA, and SECS (introduced in Section II-C)— along with the corresponding EPCM entries.

Every SGX enclave maintains the TCS and SSA pages for every thread. The TCS includes a set of offsets that point to the head of particular data structures. For example, the Offset of the SSAs (OSSA) contains the head of the SSA pages. We

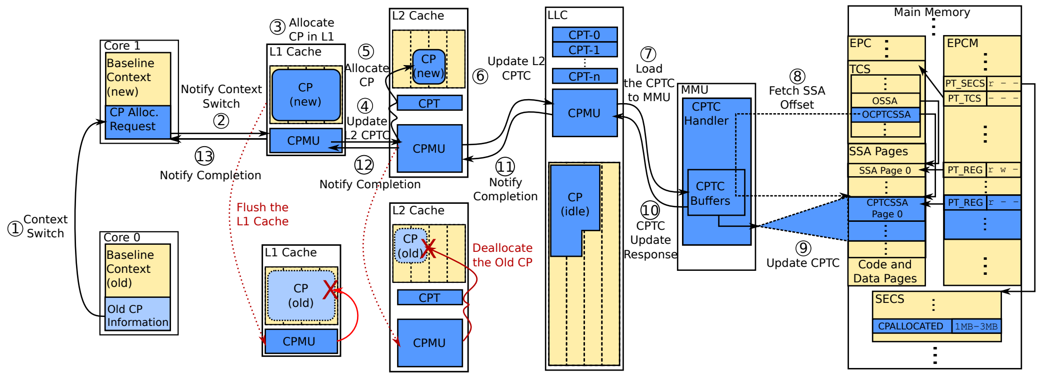


Fig. 7: Context switching example for enclaves with CPs. White and yellow colored structures represent the standard logic. Light blue represents structures associated with the CPTC of the enclave being switched out; while dark blue represents additional logic and structures. Red arrows are related to the operations that are performed in the background and not a part of the main workflow.

extend the MMU and the aforementioned areas to store the CPTC alongside the traditional thread context.

We extend the TCS with the Offset of the CPTC State Save Area (OCPTCSSA), which contains the relative address to the head of the CPTCSSA pages. Considering that there is a separate TCB for each thread of an enclave process, TEE-SHirT can provide a non-uniform allocation to threads as an additional feature, meaning that threads of the same enclave can allocate differently sized CPs. In terms of composition, there is no difference between regular SSA and CPTCSSA pages. The only divergence between them stems from the contents and corresponding permission bits in the EPCM, which has `rw-` permissions for the regular SSA pages. Due to security implications, we cannot let any software explicitly modify the CPTCSSA; that task is handled only by TEE-SHirT. Hence, we reduce the permissions on CPTCSSA pages to `r--` for "read-only", making them non-cacheable and accessible only by the hardware.

To store the combined CP size allocated by an enclave, we can also add an auxiliary field called `CPALLOCATED` to SECS to store the total CP sizes for the enclave on each cache level. In the figure, the `CPALLOCATED` indicates 1MB and 2MB CP allocation for L2 and L3 caches, respectively.

3) Partition-Aware Context Switches: Figure 7 illustrates the cross-core context switch process of a single-threaded enclave with pre-allocated CPs along with the memory layout extensions to SGX. In ①, the baseline context switch is initiated, which is followed by an enclave allocation instruction in ②. After the allocation request is processed by the L1 cache, the L1 cache is flushed to avoid leakage in ③. The request received by the L1 cache contains the allocation data for both of the lower-level caches, so it can be forwarded to the L2 cache in ④. The allocation process in ⑤ involves the CPMU in the L2 cache checking for available cache space and filling the CPT with the new CPTC. When caches receive a context switch-related allocation request, they send a CP deallocation message to other same-level caches in the system.

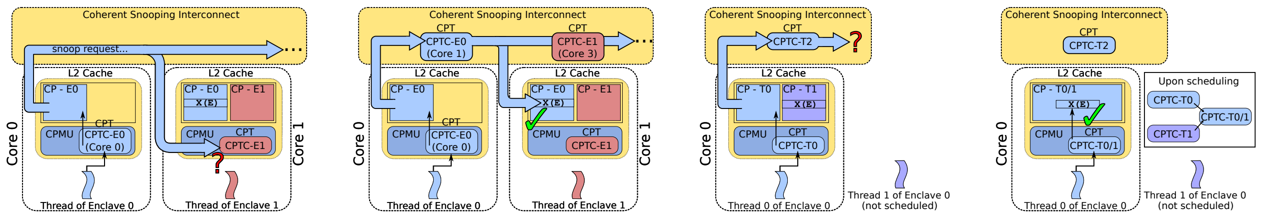
When caches create a new CPTC, they forward it to the lower-level structure in the hierarchy. So, in ⑥, the L2 cache sends its newly created CPTC to the LLC. Since all of the cores share the LLC, we do not need to update the CPTC at that level. The LLC propagates the request to the MMU in ⑦. Before updating the CPTCSSA pages in ⑨, we have to load the OCPTCSSA in ⑧. The CPTC Handler is equipped with a state machine and dedicated CPTC buffers that carry out the CPTC Update operation which reads the OCPTCSSA and updates the CPTCSSA pages. Once the MMU is done with the updating process, it sends a "completion" response in ⑩. ⑪, ⑫, and ⑬ forward events to notify completion of the CPTC-update operation to the core.

Still, the core has to wait until existing non-enclave data within the boundary that is now allocated to the program are gang-invalidated so that the data integrity of non-enclave processes is retained.

C. Multithreaded Enclaves with Partitioned Caches

A distinct problem with cache partitioning is how to correctly and efficiently support enclave data sharing with multi-threaded enclaves. Modern TEEs such as Intel SGX offer support for multi-threaded enclaves, where threads within an enclave are mutually trusted. These threads can naturally access shared memory locations. While this programming model does not pose challenges for LLC-only partitioning schemes, *consistency and coherence* problems can occur when multiple levels of the cache hierarchy are partitioned, including private caches. We explore this challenge under two scenarios: 1) Inter-core sharing, when threads of an enclave are being executed in separate cores, and 2) Intra-core sharing when threads belonging to the same enclave are being context-switched in and out within the same core. In this section, we initially focus on snooping-based [25] MOESI coherence protocols [62], and then we discuss the implications of these scenarios for directory-based coherence [16].

1) Challenge 1 - Inter-Core Data Sharing: Traditional cache coherence protocols are built on the premise that cache accesses can be performed solely based on the cache line's address. This assumption does not apply to the design principle of TEE-SHirT, as we rely on the CPTCs to deflect the access



(a) Snoop request for **X** fails due to the lack of required CPTC in Core 1

(b) The snooping CPT embedded in the interconnect is used to access CPs in other core's L2 cache

(c) Snooping broadcast cannot cannot retrieve **X** from other cores when caches are partitioned

(d) Merged partitions effectively resolve the intra-core coherence issue.

Fig. 8: The illustration of TEE-SHirT-related snooping challenges and respective solutions

to the CPs held by an enclave. Therefore, when a cache is snooped in TEE-SHirT, only lines within the enclave's CPTC should be accessed.

This challenge is illustrated in Figure 8(a), where threads of different enclaves run on Core 0 and Core 1. The thread of Enclave 0 tries to access the line **X**, which is in an Exclusive state only in Core 1. Therefore, the read request in Core 0 results in a snooping broadcast to all other cores and the L3 cache. When the snoop request reaches Core 1, where a thread of Enclave 1 is currently running, the L2 cache cannot access **X** in the partition boundary. At this point, Core 1 has two options: it can either wait for the corresponding thread of Enclave 0 to be scheduled, or it can manually load the CPTC of Enclave 0 each time it gets snooped; both of these alternatives would incur significant performance overhead.

To address this issue, we propose including additional CPTs for each core, embedded in the cross-core interconnect fabric, which we call *snooping-CPT*s. Snooping-CPTs serve as a cache-like optimization for coherence interconnects, allowing them to store frequently accessed CPTCs and access them quickly. In cases where a CPTC is not found in the snooping-CPT during a coherence query, it needs to be loaded from CPTCSSA pages. As they contain the CPTCs of L2 caches of other cores, their size is the same as regular CPTs in L2 caches. This mechanism improves performance by reducing the latency associated with fetching CPTCs from memory.

Figure 8(b) depicts this strategy, where following the L2 cache miss, Core 0 accesses its snooping-CPT entry associated with **X**, and subsequently, the interconnect broadcasts the snoop with the retrieved entry to all other cores. As a result of this broadcast, Core 1 successfully retrieves **X** by utilizing the snooping-CPT entry served from the interconnect. However, if Core 0 tries to access a cache line that exists in another core than Core 1, its snooping-CPT has to replace its CPTC, which causes some performance overhead. We evaluate snooping-CPT hit rates and compare the average snooping delay inflicted by snooping-CPT and load-CPTC-on-snoop approaches in Section VI-C.

2) Challenge 2 - Intra-Core Data Sharing: Threads within an enclave execute separate workflows once they are initiated, resulting in disjointed execution paths. This implies that enclave threads, in conjunction with TEE-SHirT, can invoke allocation instructions with individual parameters, enabling them to independently allocate cache partitions of varying sizes while concurrently sharing data. This allows enclaves to create threads with diverse workloads to meet their varying resource requirements. However, when multiple threads of a single enclave are scheduled into a single core, consistency issues emerge when they access the same address through separate partitions that reside in a single cache.

An illustrative example of such a scenario is depicted in Figure 8(c), which showcases the execution of Thread 0 and Thread 1 of an enclave, both scheduled on Core 0. Both threads have CPs in the L2 cache and are running concurrently. Thread 1, again, has an instance of **X** in its CP. After Thread 0 gets context switched in the core, it attempts to load **X**. The load request leads to a cache miss, triggering a snooping broadcast. However, the queries to all cores result in a miss, indicating that the system is unable to retrieve the most recent version of **X**.

Hence, if cache partitions are maintained separately, the system needs to employ a mechanism that iterates through each partition individually, resulting in a significant cost. Instead, we propose *partition merging* where threads belonging to the same enclave all access the same CPs. To do this, when two threads of the same enclave are scheduled in the same core, the CPMU has to combine the CPTCs of the threads into one and update the changes in the CPTCSSA pages of both threads. This ensures that there is at most a single instance of a data line in each cache. This approach not only addresses the cost issue but also aligns with our threat model since threads are permitted to access the same enclave pages. The CPMUs in caches are responsible for handling the merging process during cross-core context switches and updating the CPTCs accordingly.

We demonstrate the efficacy of partition merging in Figure 8(d). When Threads 0 and 1 are scheduled in Core 0, CPMU combines CPTC-T0 and CPTC-T1 in a merged CPTC (denoted as CPTC-T0/1). This way, **X** can be serviced from the common partition. By having a shared partition for all threads within an enclave, TEE-SHirT ensures that a cache line does not have multiple instances in a single cache, preventing version inconsistencies.

3) Considerations for directory-based coherence: Similar challenges also arise for directory-based coherence mechanisms for the two scenarios. For example, for intra-core sharing, a cache still has to have the CPTC of the cache line after a directory query. However, in systems with a directory, TEE-SHirT can utilize the directory entries to include supplementary CPT entries as an additional field. By adding the corresponding CPT entry of the cache line to the directory entries, probes to cache lines can be served upon request. Directory entries need to exclusively store the CPT entry for cache lines in the Modified, Owned, and Exclusive states, as they indicate that the cache line has an updated version present in only one cache. Notably, this approach does not introduce additional cycles during coherence-related probes but incurs additional hardware overhead. Intra-core sharing with directories can also be efficiently supported with a partition merging approach as described above

While these solutions handle coherence challenges, it has been previously shown that directories are also prone to side-channel attacks [72]. As we do not alter the remapping of the directory, proposed directory defense mechanisms such as SecDir [73] can be cleanly integrated with TEE-SHirT.

V. A Formal Security Analysis

We now rigorously establish the security guarantees provided by TEE-SHirT. We present an outline of the formal model here, with the full model included in the Appendix. Our approach is to define the essential TEE-SHirT-aware program behavior through small-step operational semantics, whose metatheory confirms the desirable properties of TEE-SHirT. This is a non-trivial task: our semantic system captures a rich yet essential set of features — such as cache hierarchy access and replacement, cache coherence, enclave lifecycle, and hardware/OS context switch — and it reasons about the properties of cache isolation and side-channel immunity.

A. Definitions

a) Common Notations: Notation $\overline{X}^m$ represents the sequence of $[X_1, \ldots, X_m]$ for some $m \geq 0$. When the length of a sequence does not matter, we also shorthand $\overline{X}^m$ as $\overline{X}$. We use $\emptyset$ to represent an empty sequence and comma (,) as the binary operator for sequence concatenation. We also call a special form of sequences, $\overline{X \mapsto Y}$, a *mapping* when the elements in $\overline{X}$ are distinct. For any mapping, we use notations $M[X \mapsto X']$, M_X, $dom(M)$, $ran(M)$ to refer to the update, restriction, domain, and range of mapping M with standard definitions.

b) Physical Cache Hierarchy and CPU Cores: For convenience, we associate identifiers to both physical cache units ($\lambda \in \mathbb{PCU}$) and CPU cores ($q \in \mathbb{CORE}$). The cache hierarchy is captured by a static structure $H : \mathbb{PCU} \cup \mathbb{CORE} \rightarrow \mathbb{PCU} \cup \{\top\}$, which maps a "child" cache unit in the cache hierarchy to its "parent" cache unit, where a "child" cache unit is closer to the CPU core than its "parent". For completeness, we use $\top$ to represent the "imaginary" parent of the physical cache unit at the root of the cache hierarchy.

Top-Level Structures		
Σ	$::= \langle \kappa; \mu; \rho; \pi \rangle$	*runtime state*
μ	$::= \overline{b \mapsto D}$	*memory*
ρ	$::= \overline{r \mapsto v}$	*registers*
π	$::= \overline{p \mapsto \langle \epsilon; l; q \rangle}$	*program store*
Multi-Level Cache		
κ	$::= \overline{\lambda \mapsto \psi}$	*multi-level cache*
ψ	$::= \langle F; V; C; R \rangle$	*single-level cache unit*
PCU-Related Structures		
F	$::= \overline{c}$	*free list*
V	$::= \overline{e \mapsto L}$	*CPT*
C	$::= \overline{c \mapsto \langle vb; t; D \rangle}$	*way-set cache*
L	$::= \overline{s \rightarrow W}$	*remapping list*
D	$::= \overline{\delta \mapsto v}$	*data block*
c	$::= \langle w; s \rangle$	*CP index*
vb	$::= \mathsf{M} \mid \mathsf{O} \mid \mathsf{E} \mid \mathsf{S} \mid \mathsf{I}$	*coherence bit*
W	$::= \overline{w}$	*way mask*
Memory/Register-Related Structures		
v	$::= \iota \mid n$	*memory value*
ι	$\in \mathbb{INST}$	*instruction*
n	$\in \mathbb{DATA}$	*data*
Enclave-Related Structures		
ϵ	$::= \langle e; E \rangle$	*enclave state*
E	$::= \overline{\mathsf{e} \mapsto \langle l; n \rangle}$	*enclave memory range*
e	$::= \mathsf{e} \mid \bot$	*enclave ID*
Access and Observations		
a	$::= \mathsf{R} \mid \mathsf{W}$	*access type*
ot	$::= a \mid \mathsf{LM} \mid \mathsf{GM}$	*observation type*
o	$::= \langle ot; c; \lambda \rangle$	*observation*
O	$::= \overline{o}$	*observation trace*
τ	$::= \langle l; a; \epsilon \rangle$	*access descriptor*
Identifiers and Atomic Values		
q	$\in \mathbb{CORE}$	*core ID*
λ	$\in \mathbb{PCU}$	*physical cache unit ID*
w	$\in \mathbb{WAY}$	*way ID*
s	$\in \mathbb{SET}$	*set ID*
b	$\in \mathbb{BLOCK}$	*block ID*
r	$\in \mathbb{REG}$	*register ID*
e	$\in \mathbb{ENCLAVE}$	*raw enclave ID*
δ	$\in \mathbb{Z}^*$	*data offset*
t	$\in \mathbb{TAG}$	*cache tag value*
l	$\in \mathbb{ADDR}$	*memory address*

Fig. 9: Runtime Definitions

Example V.1 (Cache Topology)**.** *For a two-core CPU (q_1 and q_2) with 2 L1 private caches (λ_1 and λ_2) and 1 L1 shared cache (λ_3), H is defined as*

$$q_1 \mapsto \lambda_1, q_2 \mapsto \lambda_2, \lambda_1 \mapsto \lambda_3, \lambda_2 \mapsto \lambda_3, \lambda_3 \mapsto \top$$

In line with the notion of the cache *hierarchy*, we further require that H be a total and surjective function, and the relation it defines forms a poset. Since H is static for a concrete machine, the definitions for the rest of this paper are implicitly parameterized by this structure.

c) Runtime State: As shown in Fig. 9, the runtime state (Σ) consists of the TEE-SHirT (κ), the memory (μ), the register file (ρ), and the program store (π). For cache lines, note that a cache coherence bit (vb) is associated; our operational semantics is fully compliant of the MOESI cache protocol [62]. Each entry in our program store π is aligned with our intuitive notion of a thread. Our formal system considers the general model where multiple applications may co-run on the same system, and each application may be multi-threaded. Our operational semantics makes the common assumption that different applications (such as the victim and attacker) do not share memory locations, but the threads within an application may share memory, where cache coherence is at work. Given a memory address l, we define a bijective function $\alpha : \mathbb{ADDR} \rightleftharpoons \mathbb{BLOCK} \times \mathbb{Z}^*$ to compute its block index b and offset δ in the block. We use $\mu\{l\}$ to refer to $\mu(b)(\delta)$ where $\alpha(l) = \langle b; \delta\rangle$. We use $\mu\{l \mapsto v\}$ to refer to $\mu[b \mapsto D']$ and $D' = D[\delta \mapsto v]$ where $\alpha(l) = \langle b; \delta\rangle$.

B. Observation and Observation Traces

To reason about side channels, what can be observed is important and must be clearly defined.

Definition V.1 (Observation and Observation Trace)**.** *We define an observation* o *as a tuple* $\langle ot; c; \lambda\rangle$*. It says CP* $c \in \mathbb{WAY} \times \mathbb{SET}$ *residing in physical cache unit* $\lambda \in \mathbb{PCU}$ *is accessed with observation type* $ot \in \{\mathsf{R}, \mathsf{W}, \mathsf{LM}, \mathsf{GM}\}$*. The identifiers correspond to a hit-read, a hit-write, a level-scoped miss (the content is only available in a peer cache on the same cache level), a global miss (the content is not available in any cache of the same cache level), respectively. An observation trace* O *is defined as a sequence of observations.*

Example V.2 (Observation Trace)**.** *Observation trace* $[\langle \mathsf{R}; \langle w_1; s_1\rangle; \lambda_1\rangle, \langle \mathsf{LM}; \langle w_1; s_2\rangle; \lambda_1\rangle, \langle \mathsf{W}; \langle w_3; s_3\rangle; \lambda_2\rangle]$ *says in physical cache unit* λ_1*, CP* $\langle w_1; s_1\rangle$ *is first hit-read and its CP* $\langle w_1; s_2\rangle$ *access leads to a cache miss but found in another cache unit on the same cache level; afterward, in physical unit* λ_2*, CP* $\langle w_3; s_3\rangle$ *is hit-written.*

Our definition of observations is *fine-grained*: an attacker can observe not only the timing/power/magnetic-field difference of hit-reads/hit-writes/local-misses/global-misses but also where they happen: a specific CP in a specific physical cache unit (level). As we shall see, our theorems state that despite strong assumptions about the attacker's observation capability, our system can guarantee that an attacker cannot use such observations to infer program values.

With parallelism inherent in our scope, the observation trace generated by individual threads may be interleaved when taking a global view. To capture this, we introduce

Definition V.2 (Global Observation and Observation Trace)**.** *A global observation* ω *is a tuple* $\langle p; \mathrm{o}\rangle$*, where* p *is the thread where the observation* o *is observed. A global observation trace* Ω *is a sequence of global observations.*

C. Operational Semantics

a) Multi-Level Access: For convenience, we define access through an *access descriptor* τ, defined as a triple $\langle l; a; \epsilon\rangle$, including the address to be accessed l, the access mode itself a, and the enclave where the access is initiated.

The behavior of intra-level and inter-level cache access is defined in Fig. 10. Relation $\kappa, \mu \xRightarrow[O]{\lambda, v, \tau} \kappa', \mu'$ says that a multi-level cache κ and main memory μ transitions to κ' and μ' respectively, when cache unit λ subjects to access defined by access descriptor τ. v computes the read result when the access is a read, and it carries the value to be written when the access is written. The access produces a set of observations in O. The first rule demonstrates the intra-level behavior, including both a hit to the requested cache unit, or a localized miss where the content can be found in a peer cache unit on the same cache level. This rule is defined over

$$\frac{\kappa \xrightarrow[O]{\lambda, v, \tau} \kappa' \qquad O \neq \{\langle \mathsf{GM}; c; \lambda'\rangle\} \text{ for any } c \text{ and } \lambda'}{\kappa, \mu \xRightarrow[O]{\lambda, v, \tau} \kappa', \mu}$$

$$\frac{\begin{array}{c}\kappa \xrightarrow[O]{\lambda, v, \tau} \kappa'' \\ O = \{\langle \mathsf{GM}; c; \lambda'\rangle\} \text{ for some } c \text{ and } \lambda' \qquad \kappa'', \mu \xRightarrow[O']{H(\lambda), v, \tau} \kappa', \mu'\end{array}}{\kappa, \mu \xRightarrow[O \cup O']{\lambda, v, \tau} \kappa', \mu'}$$

$$\frac{\tau = \langle l; \mathsf{R}; \epsilon\rangle}{\kappa, \mu \xRightarrow[\emptyset]{\top, \mu\{l\}, \tau} \kappa, \mu} \qquad \frac{\tau = \langle l; \mathsf{W}; \epsilon\rangle}{\kappa, \mu \xRightarrow[\emptyset]{\top, v, \tau} \kappa, \mu\{l \mapsto v\}}$$

Fig. 10: TEE-SHirT Multi-Level Cache Access

[MULTI]

$$\frac{\kappa, \mu, \rho, \epsilon, \mu(l) \xrightarrow[q, n]{O} \kappa', \mu', \rho', \epsilon' \qquad \mu(l+n) = \iota \text{ for some } \iota}{\kappa, \mu, \rho, \pi[p \mapsto \langle \epsilon; l; q\rangle] \xRightarrow{O@p} \kappa', \mu', \rho', \pi[p \mapsto \langle \epsilon'; l+n, q\rangle]}$$

[CONTEXTSWITCH]

$$\frac{q \neq q'}{\kappa, \mu, \rho, \pi[p \mapsto \langle \epsilon; l; q\rangle] \overset{\emptyset}{\Longrightarrow} \kappa, \mu, \rho, \pi[p \mapsto \langle \epsilon; l; q'\rangle]]}$$

Fig. 11: Parallel Operational Semantics

another relation Relation $\kappa \xrightarrow[O]{\lambda, v, \tau} \kappa'$. It corresponds to the "intra-level" behavior when cache unit λ is accessed; we defer this definition to the Appendix, which captures the essence of MOESI. The second rule shows the inter-level behavior if a miss cannot be resolved at one cache level. The third and fourth rules define the read and write behavior when main memory is (ultimately) accessed.

b) A Parallel Model with Context Switches: We discuss the small-step operational semantics of programs running on TEE-SHirT next (they are defined in Figure 11 in the appendix). Reduction relation $\Sigma \overset{\Omega}{\Longrightarrow} \Sigma'$ says that runtime state Σ reduces to Σ', producing global observation trace Ω.

A simple but important observation is that parallelism support is inherent in our formal system: different threads may be executed in parallel on multi-core CPUs where SMT is allowed. Any thread in π may take a reduction step (the [MULTI] rule described in the appendix). Some of these threads may host the victim execution, while others may host the attacker execution. The reduction sequences of these parallel threads may interleave in an arbitrary manner allowed by the SMT hardware, OS scheduler, and program logic of the application. With minimal restrictions on the parallel semantics, the properties we formally establish make minimal assumptions on the software side, regardless of, e.g., what victim programs are run, how the attacker constructs her program, how many attacker threads may collude, and how the victim program and the attacker program interleave.

As context switches have non-trivial implications on secure cache hierarchy design, a [CONTEXTSWITCH] rule captures this behavior: the CPU core a thread resides on may change at any arbitrary reduction step, either due to the SMT-level context switches or OS scheduler.

[LOAD]

$$\frac{\tau = \langle l; \mathsf{R}; \epsilon\rangle \qquad \kappa, \mu \xRightarrow[O]{H(q), v, \tau} \kappa', \mu'}{\kappa, \mu, \rho, \epsilon, \texttt{LOAD}\ l\ r \xrightarrow[q,1]{O} \kappa', \mu', \rho[r \mapsto v], \epsilon}$$

[STORE]

$$\frac{\tau = \langle l; \mathsf{W}; \epsilon\rangle \qquad \kappa, \mu \xRightarrow[O]{H(q), \rho(r), \tau} \kappa', \mu'}{\kappa, \mu, \rho, \epsilon, \texttt{STORE}\ r\ l \xrightarrow[q,1]{O} \kappa', \mu', \rho, \epsilon}$$

Fig. 12: Selected Reduction Rules of Single-Process Operational Semantics

The [MULTI] rule also shows the program counters at work. In our system, program counters are memory addresses pointing to the instruction sequence. Whenever a program executes a step, the reduction system tracks what the offset of the program counter should be for the next instruction. For convenience, we use $\bullet$ to represent the program counter when the program halts. We define $O@p$ as $[\langle p; o_1\rangle, \ldots, \langle p; o_n\rangle]$ where $O = [o_1, \ldots o_n]$ for some $n \geq 0$.

c) Instruction-Specific Behaviors: The parallel reduction system bridges with the single-thread reduction system, which defines the behavior of a thread over a single-thread state σ. The latter is defined as $\langle\kappa; \mu; \rho; \epsilon\rangle$, where κ, μ, and ρ are the states of the TEE-SHirT , memory, and registers respectively, while ϵ is the enclave that is currently under execution. Reduction relation $\sigma, \iota \xrightarrow[q,n]{O} \sigma'$ says single-thread state σ reduces to σ' while executing instruction ι at CPU core q, producing global observation trace O.

Our formal system defines the behavior of the enclave lifecycle (`CREATE`, `ENTER`, `EXIT`, `DESTROY`), memory/cache access (`LOAD` and `STORE`), and control flow (`BR`). A selected subset of the reduction rule is shown at the bottom of Fig. 12, with the rest deferred to the appendix. Our semantic model treats dynamic partitioning as the default behavior. Static partitioning is a restrictive form of semantics where the initial number of CPs is equal to the number of threads, and CP allocation/deallocation happens at the program start/termination time. As a result, our metatheory below subsumes the practical design where L1 undergoes static partitioning.

D. Metatheory

Theorem V.1. *The TEE-SHirT design is immune against side channel attacks.*

We defer the rigorous statement of this important theorem to the appendix. Formally, the theorem resembles the "Immunity Against Side Channel Attacks" theorem in prior work [63], but under the more realistic assumption that a cache hierarchy and a cache coherence protocol are at work, and parallel program executions are supported.

VI. EVALUATION

A. Experimental Methodology

To evaluate the performance of TEE-SHirT, we implemented our design in gem5 [7] cycle-accurate simulator with system call emulation mode. Since gem5 does not have support for simulating native-SGX enclaves [1], we simulated the scenario where the entire program executes in an enclave, but without modeling SGX interfaces. The only impact that our design has on SGX interfaces is the time to store partitioning metadata on a context switch out of an enclave. We account for this by adding an additional memory access(es) to store/retrieve partitioning metadata.

To evaluate the impact on the performance of enclave programs, we simulated a variety of benchmark suites, namely SPEC2017 [12], MiBench [34], PARSEC [6], and three important functions from the WolfSSL cryptography library [2] (*fp_gcd*, *fp_gcd*, and *wc_ecc_mulmod_ex*, all executed repeatedly in a loop). This range of programs allows us to evaluate the impact of TEE-SHirT on applications with various memory demands. The primary metric we track for performance is Instructions Committed Per Cycle (IPC).

TABLE I: Parameters of the Simulated System

Hardware Parameters	
Core	x86 ISA Out-of-Order cores
Cache Hierarchy	Snooping-based MOESI coherence policy, inclusive write-back caches
L1i/d Caches	32KB total size, 8-ways, 4-cycle access latency (one for each core)
L2 Cache	512KB total size 8-ways, 16-cycle access latency (one for each core)
L3 Cache	4MB total size 16-ways, 32-cycle access latency (shared)
DRAM	32GB size, 4GB channel capacity, DDR4-2400 x64 channel, 4 devices per rank, 1 rank per channel, 1GB per device

Mixes of SPEC2017 Benchmarks	
Mix-1	cam4, perlbench, bwaves, fotonik3d
Mix-2	mcf, exchange2, blender, cactusBSSN
Mix-3	nab, x264, namd, parest
Mix-4	povray, imagick, omnetpp, gcc
Mix-5	wrf, lbm, xalancbmk, leela

For the CPTC virtualization experiments, we execute single-core simulations of SPEC2017 and MiBench benchmarks. To emulate context switches, we assumed three time quantum values: 1ms, 10ms, and 25ms, and extended the simulator to emulate the gang-invalidation process including write-backs of dirty data, as described in detail in Section IV-A. We also flush the caches in system calls as well as emulated context switches. We block the caches from being accessed until all the data is written back.

For TEE-SHirT performance experiments, we simulated a 4-core system in gem5's system call emulation mode where all four cores execute five mixes of SPEC2017 programs concurrently. The mixes are specified in Table I. In this design, we assume that every enclave access is extended by two extra cycles to the cache latency at L2 and L3 levels - one cycle for the extra delay through the remapping logic and one cycle to account for the extra complexity in cache replacement logic. This is consistent with the estimates in prior work [56], [63]. Saileshwar et al. [56] showed that when the extra LLC latency due to partitioning increases from 1 to 6 cycles, performance decreases minimally - from 1% to 2%. Therefore, the additional latency at the level of L2/L3 caches has minimal impact on the overall program throughput. The L1 cache is flushed with a time quantum of 25ms to simulate

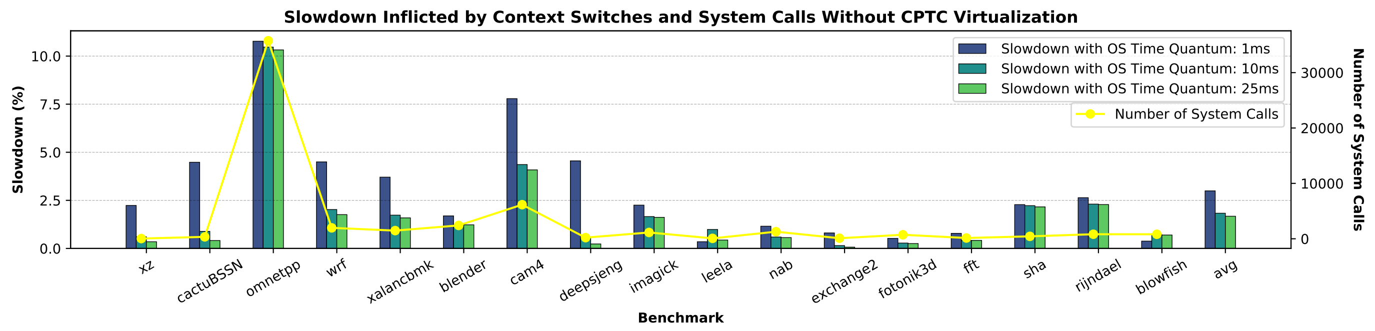


Fig. 13: The slowdown percentage inflicted by cache flushing when CPTC virtualization is not implemented. The yellow plot indicates the number of system calls invoked by the benchmark on the right-hand side y-axis.

a Linux context switch interval.

We enhanced gem5's snooping-based classic cache hierarchy to emulate our coherence optimizations as outlined in Section IV-C. We utilize PARSEC [6] benchmarks with four threads on a 4-core system with full system simulation. We allocate a quarter of the capacity in all caches for each of the threads, where the L3 CP is utilized by all threads due to partition merging. We perform detailed simulations until any of the threads reach 100 million instructions after the kernel boot.

We evaluated multi-core system performance using both small crypto programs and larger SPEC 2017 benchmarks. First, to evaluate the cache impact of the mixed crypto and regular workloads, we combined three cryptography programs (from Mibench suite and WolfSSL functions) and the *xz* benchmark (from SPEC2017). Furthermore, to evaluate cache performance for more memory-intensive workloads, we ran five mixes from the SPEC2017 suite (specified in Table I) under different L2 and L3 configurations. We fast-forwarded the execution for the first one billion instructions and performed detailed simulations for the one billion instructions. We had to limit the number of instructions executed because of the simulation time. Skipping the first Billion instructions bypasses the initial phase of the program and warms up caches, branch predictors and other microarchitectural structures of the processor. Simulating for one billion instructions provides sufficient statistics and is in line with the size of simulation samples used by typical computer architecture studies [20], [21], [56], [63], while all cryptography benchmarks were run to completion.

To evaluate TEE-SHirT's impact on area, we implemented it in the cache subsystem of the open-source ESP SoC platform [47]. Our implementation consists of a configurable number of CPTs in the L2 caches and the LLC. As a metric for hardware area overhead estimation, we measure the increase in the utilization of FPGA resources relative to the baseline SoC, which has one 512KB L2 cache per core, and a shared 4MB LLC in a 4-core LEON3 [27] CPU. We used the Vivado Design Suite [26] to synthesize an FPGA prototype of the design and collected utilization values for several different configurations. As our target platform, we used the AMD Virtex UltraScale+ VCU118.

B. Performance of CPTC Virtualization

Figure 13 shows the performance advantage that can be gained by CPTC virtualization over Scenario 2 in Section IV-B1. Recall that this scenario requires flushing the LLC partition on a context switch to preserve cache consistency. The baseline shown in this figure is the CPTC virtualization (zero across all benchmarks), where the data is kept in the LLC across context switches. As expected, the largest performance gain was observed for 1ms time quantum, where benchmarks such as *cactusBSSN, omnetpp, wrf, xalancbmk,* and *cam4* yield 4.4%, 10.8%, 4.5%, 3.7%, and 10.4% speedup, respectively. For a more conventional 25ms time interval, *omnetpp* generated the highest speedup of 10.3%.

The main takeaway from these results is that the system call intensity of a benchmark is more correlated to the speedup it gains from CPTC virtualization compared to context switches. Most benchmarks with infrequent system calls have modest performance losses. On the other hand, system call intensive benchmarks such as *omnetpp* and *cam4* tend to experience performance degradation even when the time quantum is increased (at around 10% and 4% respectively). These results indicate that workloads that frequently invoke kernel operations experience the greatest benefit from data retention achieved through CPTC virtualization.

C. Performance of Cache Coherence Optimizations

Figure 15 shows the average normalized snooping time between different approaches. We selected a baseline case where snoops between threads do not introduce any additional delays. In the remaining bars of the graph, we present the snooping delays incurred when implementing different configurations: snooping-CPT and loading the CPTC upon a snoop.

Among the workloads examined, the benchmarks *fluidanimate* and *raytrace* exhibit a significant number of snoops between L2 caches, resulting in higher average snoop delays. Specifically, *fluidanimate* demonstrates an average snoop delay of 1.91 times that of the baseline, while *raytrace* exhibits an average snoop delay of 2.11 times the baseline. However, when snooping-CPTs are employed, these benchmarks experience improved performance, with average snoop delays reduced to 1.08 times and 1.09 times the baseline for these workloads respectively. This is due to high snooping-CPT hit rates across

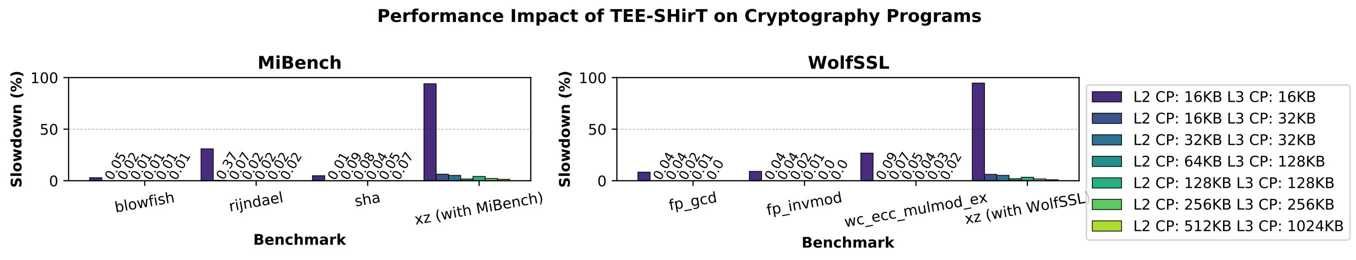


Fig. 14: Multi-core simulation for MiBench and WolfSSL. The legend shows CP sizes for L2 and L3 caches

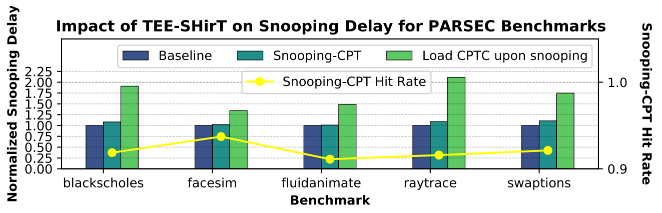


Fig. 15: Normalized average snoop delay for PARSEC benchmarks under three conditions. The yellow plot indicates the Snooping-CPT hit rate on the right-hand side y-axis

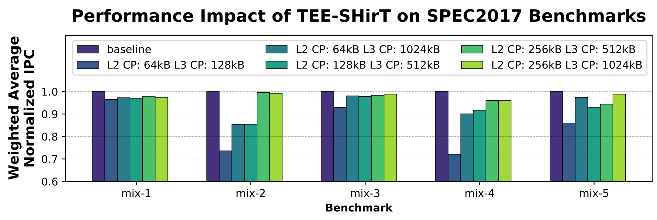


Fig. 16: Multi-core simulation results for mixes of SPEC2017 benchmarks. We use the number of committed instructions by the benchmarks as the weights for the average IPC. The legend shows CP sizes for L2 and L3 caches.

the board, all of them being between 91.5% and 93.7%. These results show the effectiveness of the caching mechanism for CPTCs.

D. Overall Performance of TEE-SHirT

Figure 14 shows the results for cryptography programs, which may often be isolated in TEEs. The four leftmost bar groups show the MiBench experiments, while the four rightmost ones show the WolfSSL functions. The smallest overall configuration we provide is 16KB CPs for both L2 and L3 caches, which produces the highest slowdown percentages: 1.2%, 32.2%, 6.4%, 96.3%, 9.5%, 9.6%, 24.7%, and 96.9% for *blowfish, rijndael, sha, xz (with MiBench), fp_gcd, fp_invmod, wc_ecc_mulmod_ex*, and *xz (with WolfSSL)*, respectively. However, the large slowdowns decrease significantly once we increase the L3 cache's CP to 32KB. In this case, all security benchmarks exhibit a slowdown below 0.5%. These results demonstrate that some benchmarks can perform very well with small CPs at the LLC level - something that would not be practical if L2 caches were not partitioned as well.

One of the key insights from these results is that when L2 and L3 CPs are the same sizes, they seem to perform worse than the cases where the L2 cache's CP is smaller. For *xz* in both runs, when CPs in L2 and L3 are 128KB, the overall slowdown is higher than in the case with 64KB L2 CP and 128KB L3 CP (5.3% and 1.3% respectively). This is due to the inclusivity effect we discussed in Section III, where the miss penalty in the L2 cache effectively increases by the redundant L3 access latency since the cache line is not present in L3. This result shows that arbitrarily increasing the size of CPs without considering the interaction of cache levels can have a detrimental performance impact.

Figure 16 shows the performance impact of TEE-SHirT on large SPEC2017 programs, where individual benchmarks included in the mixes are outlined in Table I. The highest degradation is observed for *mix-2* and *mix-4*, where even in the smallest configuration they have 26.4% and 28% performance loss respectively. However, when a 256KB CP is allocated in the L2 cache and 512KB is allocated in the L3 cache (a quarter of the total both cache sizes) per enclave, all of the benchmarks experience slowdowns lower than 3%. This shows that TEE-SHirT can be implemented with modest slowdowns even for large workloads.

E. Area Overhead

Table II shows the LUTs and FFs utilized for several possible configurations of TEE-SHirT. Each row corresponds with a separate hardware synthesis run. The CPT Configuration values are in the format (x, y), where there are x CPTs for each L2 and y CPTs in the LLC. Each CPT has 16 entries, and half the ways of each cache are allocable to enclaves. With a (1, 4) configuration, the modifications incur only 0.7% additional LUTs and 1.3% additional FFs. Though this configuration can support fewer concurrently executing enclaves (i.e., one enclave using the L2 and four using the LLC), it requires less hardware overhead than a (4, 16) configuration (1.4%-3.7%). CPTC virtualization enables the use of low-area configurations such as (1, 4) or (2, 8) with minimal performance impact.

TABLE II: Area estimates for TEE-SHirT configurations

CPT Configuration	Total LUTs	Total FFs
Baseline	209972 (100%)	151304 (100%)
(1, 4)	211476 (+0.7%)	153299 (+1.3%)
(2, 8)	211986 (+1.0%)	154475 (+2.1%)
(4, 16)	212798 (+1.4%)	156851 (+3.7%)
(8, 32)	214297 (+2.1%)	161500 (+6.7%)

Notably, the additional FFs required to implement more CPTs outpace the additional LUTs. This is expected due to the register-related components of the CPT being structurally assigned to FFs. Hence, for our largest configuration, (8, 32), TEE-SHirT causes a 6.7% increase in FFs, but only a 2.1% increase in LUTs. The increased LUT and FF utilization overhead both track the exponential spacing of the number of CPTs, thus maintaining a near-linear relationship. Note that the cache's data and metadata are stored in Block RAMs, which have separate utilization figures that are not affected by our implementation.

VII. Related Work

Recent research efforts addressed cache side-channel leakage using primarily two approaches: partitioning [20], [21], [23], [41], [44], [63], [66], [67] and randomization [8], [9], [22], [53], [54], [68], [69]. While both lines of research produced insightful defenses, randomization-based approaches have several limitations. First, security guarantees of randomization schemes are often probabilistic and even advanced schemes are not immune from elaborate attacks [3], [60]. Second, randomization-based schemes can be susceptible to attacks using low-resolution channels such as cache occupancy attacks [59]. In contrast, partitioning schemes that completely isolate applications from one another in the cache provide more robust security guarantees, and can even be backed up by formal security guarantees [63].

Existing efforts in cache partitioning for security mostly targeted single cache levels, either private caches or a shared LLC. Not all of these schemes can be trivially applied to multiple cache levels; some rely on the support of trusted system software, and some have high partitioning granularity that impedes their scalability. DAWG [41] and CATalyst [44] rely on the OS to participate in partitioning decisions, which is problematic for TEE-based systems. NoMo [23] is a way-based partitioning scheme that does not rely on software support altogether, but the high granularity of allocations does not make it applicable to LLCs because of scalability and susceptibility to multi-threaded attacks. In addition, as some cache ways are shared in NoMo, leakage can occur if the victim's accesses spill into the shared portion of the cache. Other partitioning schemes, like Intel's CAT [37], [55], [57], [71], were designed for quality of service and do not guarantee isolation between processes occupying different partitions. Some partitioning approaches divide caches by sets through page coloring [10], [18], [40], [58]; however, in this case, large regions of data may need to be moved around in memory when allocating cache sets since the set allocation is bound to physical addresses.

HybCache [20] provides soft cache partitions for codes requiring isolated execution protection. HybCache requires a fully associative search within the subcache ways - this is expensive and may not easily scale to large LLCs. Moreover, HybCache does not enforce strict isolation, as normal programs can still access the entire cache. CURE [5] proposed a customizable architecture for securing enclaves from side-channel attacks. However, cache partitioning is also done at the way granularity. Bespoke Cache Enclaves is a set-based cache partitioning scheme where the cache space is divided into non-overlapping clusters composed of multiple consecutive sets [56]. A similar principle is used in Chunked-Cache [21] - a design for trusted execution environments that allows each program to have its own dedicated cache sets. Composable Cachelets [63] introduced the concept of cachelets at the LLC level, and partitioned LLC across both ways and sets.

In summary, [5], [21], [56], [63] only address security of shared last-level cache through various partitioning schemes as described above. In contrast, the key contribution of TEE-SHirT is in *coordinated partitioning* of shared and private cache levels. Our design addresses non-trivial issues of coherence and scalability that arise as an effect of implementing partitioning in private caches (such as level-2 caches in our experimental framework). Specifically, from the scalability standpoint, we demonstrated how partitioning metadata can be naturally integrated into the context of an enclave using existing SGX structures (supporting scalability at low hardware complexity). From the coherence standpoint, we showed how existing cache coherence mechanisms can be augmented to properly work with partitioned multi-level caches. We also offered a formal security model of a multi-level partitioned cache system.

The TEE-SHirT formal system is complementary to existing formal cache models designed for single-level cache units. For example, the multi-level cache $\Rightarrow$ relation is unique to our model, defined over the single-level cache $\rightarrow$ relation deferred to the appendix. We hope this modular formal development can serve as the first step for a framework to reason about a large family of cache designs against side-channel attacks. Compared with formalisms on single-level cache [63], our formal system is also unique in its support of context switches and cache coherence; both are non-trivial features critical for multi-level caches. More broadly, formal systems exist to reveal and defend against side-channels, such as those in the presence of speculation [14], [15], [32], [48].

VIII. Concluding Remarks

This paper demonstrates the importance of considering the entire cache hierarchy when reasoning about side-channel protection. TEE-SHirT is a novel design that allows the creation of arbitrary small isolated cache partitions across L3/L2 caches to support leakage-free execution of security-sensitive code, making caches a first-class citizen in the TEE ecosystem and eliminating cache side channels by design. TEE-SHirT is seamlessly integrated with existing Intel SGX data structures to support enclave context switching that preserves the TEE-SHirT state. TEE-SHirT is accompanied by a formal model to rigorously reason about the security guarantee of partitioning solutions across multiple cache levels with the support of cache coherence.

Acknowledgements

We would like to thank anonymous reviewers and the shepherd for their insightful feedback. This research was supported in part by NSF Awards CNS-2053391 and CNS-2053383.

References

[1] https://www.mail-archive.com/gem5-users@gem5.org/msg19592.html.

[2] Wolfssl. https://github.com/wolfSSL/wolfssl, 2013.

[3] Andreas Abel and Jan Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 141–142, 2014.

[4] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13. ACM New York, NY, USA, 2013.

[5] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. {CURE}: A security architecture with customizable and resilient enclaves. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[8] Rahul Bodduna, Vinod Ganesan, Patanjali Slpsk, Chester Rebeiro, and V Kamakoti. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. *IEEE Computer Architecture Letters*, 2020.

[9] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. Casa: End-to-end quantitative security analysis of randomly mapped caches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1110–1123, 2020.

[10] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Srinivas Devadas, et al. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 42–56. ACM, 2019.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. *arXiv preprint arXiv:1702.07521*, page 33, 2017.

[12] James Bucek, Klaus-Dieter Lange, and Jóakim V Kistowski. Spec cpu2017: Next-generation compute benchmark. *ICPE: ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[13] Thomas Burd, Wilson Li, James Pistole, Srividhya Venkataraman, Michael McCabe, Timothy Johnson, James Vinh, Thomas Yiu, Mark Wasio, Hon-Hin Wong, Daryl Lieu, Jonathan White, Benjamin Munger, Joshua Lindner, Javin Olson, Steven Bakke, Jeshuah Sniderman, Carson Henrion, Russell Schreiber, Eric Busta, Brett Johnson, Tim Jackson, Aron Miller, Ryan Miller, Matthew Pickett, Aaron Horiuchi, Josef Dvorak, Sabeesh Balagangadharan, Sajeesh Ammikkallingal, and Pankaj Kumar. Zen3: The amd 2nd-generation 7nm x86-64 microprocessor core. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 65, pages 1–3, 2022.

[14] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 913–926. ACM, 2020.

[15] Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *Proceedings of the Computer Security Foundations Symposium (CSF)*, 2019.

[16] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30(2):16–29, 2010.

[17] Victor Costan and Srinivas Devadas. Intel sgx explained, 2016.

[18] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 857–874, 2016.

[19] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):171–191, May 2018.

[20] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. 2020.

[21] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Chunked-cache: On-demand and scalable cache isolation for security architectures. *The Network and Distributed Systems Security Symposium*, 2022.

[22] Peter W. Deutsch, Weon Taek Na, Thomas Bourgeat, Joel S. Emer, and Mengjia Yan. Metior: A comprehensive model to evaluate obfuscating side-channel defense schemes. ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35, 2012.

[24] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62, 2017.

[25] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. *SIGARCH Comput. Archit. News*, 17(3):2–15, apr 1989.

[26] Tom Feist. *Xilinx WP416 Vivado Design Suite*, 2012.

[27] Cobham Gaisler. Leon3. www.gaisler.com/index.php/products/processors/leon3.

[28] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing os abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[31] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive Last-Level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., August 2015. USENIX Association.

[32] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: principled detection of speculative information flows. *CoRR*, abs/1812.08639, 2018.

[33] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2(11), 2011.

[34] M. Guthaus, T. Austin, D. Ernst, R. Brown, T. Mudge, and J. Ringenberg. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, Annual IEEE International Workshop*, pages 3–14, Los Alamitos, CA, USA, dec 2001. IEEE Computer Society.

[35] Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, Steve Gunther, Tom Piazza, and Ted Burton. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20, 2014.

[36] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11, 2013.

[37] CAT Intel. Improving real-time performance by utilizing cache allocation technology. *Intel Corporation, April*, 2015.

[38] Ronald Kalla and Balaram Sinharoy. Power7: Ibm's next generation server processor. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–12, 2009.

[39] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*, page 72. ACM, 2016.

[40] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the

cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

[41] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[42] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[43] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[44] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 406–418. IEEE, 2016.

[45] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[46] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. Agile soc development with open esp. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. Agile soc development with open esp. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019.

[49] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP@ ISCA*, 10, 2013.

[50] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.

[51] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 47:538–570, 2019.

[52] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. CCS '15, page 1406–1418, New York, NY, USA, 2015. Association for Computing Machinery.

[53] Antoon Purnal, Giner Lukas, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *IEEE Symposium on Security and Privacy*, pages 469–486, 2021.

[54] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 360–371, New York, NY, USA, 2019. ACM.

[55] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[56] Gururaj Saileshwar, Sanjay Kariyappa, and Moinuddin 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)*, pages 37–49. IEEE, 2021.

[57] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 57–68. ACM, 2011.

[58] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, DSNW '11, pages 194–199, Washington, DC, USA, 2011. IEEE Computer Society.

[59] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossef Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security Symposium*, 2019.

[60] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 955–969. IEEE, 2021.

[61] David Suggs, Mahesh Subramony, and Dan Bouvier. The amd "zen 2" processor. *IEEE Micro*, 40(2):45–52, 2020.

[62] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, page 414–423, Washington, DC, USA, 1986. IEEE Computer Society Press.

[63] Daniel Townley, Kerem Arıkan, Yu David Liu, Dmitry Ponomarev, and Oguz Ergin. Composable cachelets: Protecting enclaves from cache side-channel attacks. In *2022 USENIX Security Symposium*, 2022.

[64] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 991–1008, 2018.

[65] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B. Abu-Ghazaleh, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Paul L. Yu. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[66] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Secdcp: Secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 74:1–74:6, New York, NY, USA, 2016. ACM.

[67] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007.

[68] Zhenghong Wang and Ruby B Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 83–93. IEEE Computer Society, 2008.

[69] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: thwarting cache attacks via cache set randomization. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 675–692, 2019.

[70] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, Boston, MA, August 2022. USENIX Association.

[71] Yuejian Xie and Gabriel H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 174–183, New York, NY, USA, 2009. ACM.

[72] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: side-channel attacks in a non-inclusive world. In *Proceedings of IEEE Symposium on Security and Privacy*. IEEE, 2019.

[73] Mengjia Yan, Jen-Yang Wen, Christopher W. Fletcher, and Josep Torrellas. Secdir: A secure directory to defeat directory side-channel attacks. ISCA '19, page 332–345, New York, NY, USA, 2019. Association for Computing Machinery.

[74] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014.

APPENDIX

In this appendix, we present additional definition elided in the main text, and formally state the theorems presented in the paper. A detailed full proof can be found online [1].

A. Runtime State Details

Program Store and Enclave State Program store π is a mapping from thread IDs to thread states. Specifically, each thread state is a triple: its *enclave state* (ϵ), its *program counter* (l), and the CPU core it resides on (q).

As each thread may create multiple enclaves, the enclave state keeps track of both the *active enclave* — the enclave whose access is currently under execution — and all enclave-private memory regions (E). When no enclave is active, we set e as $\perp$. We further use metavariable ε for this special form of enclave state.

We define $\Omega|_p$ as the longest sequence $\overline{o}^m$ where $\langle p; o_i\rangle$ and $i = [1, ..m]$ appears in Ω, and for any $1 \le i < j \le m$, $\langle p; o_i\rangle$ appears before $\langle p; o_j\rangle$ in Ω.

B. Cache Replacement

Cache Replacement Structures			
R	$::= \overline{s \mapsto T}$		*set-indexed PLRU*
T	$::= \langle \varsigma; e; T; T\rangle \vert A$		*PLRU tree*
A	$::= \langle w; e\rangle$		*PLRU leaf*
ς	$::=$ LMRU | RMRU		*selection bit*

Function *replace*(T, e) computes the way to be evicted for enclave e given the current state of the PLRU tree T. Function *update*(T, w, e) computes the updated PLRU tree given the original one being T. Both functions have been defined in the CC paper, so we defer it to the online supplementary material here. With these definitions, the cache lookup given an acesss descriptor is defined in Fig. 22.

C. Cache Coherence

In this section, we define relevant definitions related to MOESI cache coherence. For MOESI, the transitions related to state bit setting upon a CPU request is defined in Fig. 23. The transitions related to state bit setting upon a bus request is defined in Fig. 24. To support write-back, we extend the coherence bit with label B, which can be viewed as a form of invalid state (I) before cache line write-back. The label change from B to I will be handled by the write-back relation which we will show in a later subsection.

The behavior of a single cache unit in the presence of a CPU request is defined in Fig. 25. The behavior of a single cache unit in the presence of a bus request is defined in Fig. 26. The behavior of all cache units on a single level in the presence of a bus request is defined in Fig. 27.

Finally, the behavior of a cache unit responding to a CPU request by coordinating all cache units on the same level — in conformance of MOESI — is defined in Fig. 28.

[1]https://www.cs.binghamton.edu/~davidl/papers/NDSS24-Long.pdf

$$
\begin{array}{rcl}
\psi\{c \mapsto vb'\} & \triangleq & \langle F; V; C[c \mapsto \langle vb'; t; D]; R\rangle \\
\psi\{c \mapsto D'\} & \triangleq & \langle F; V; C[c \mapsto \langle vb; t; D']; R\rangle \\
\psi\{c, \delta \mapsto v\} & \triangleq & \langle F; V; C[c \mapsto \langle vb; t; D[\delta \mapsto v]]; R\rangle \\
\psi\{s \mapsto T\} & \triangleq & \langle F; V; C; R[s \mapsto T]\rangle \\
\psi|_{\overline{\mathsf{e}}} & \triangleq & \langle F; V'; C|_{ran(V')}; R\rangle \\
 & \text{if} & V' = V|_{\overline{\mathsf{e}}} \\
\psi \# \psi' & \triangleq & (dom(V) \cap dom(V') = \emptyset) \\
 & & \wedge(dom(C) \cap dom(C') = \emptyset) \\
 & \text{if} & \psi' = \langle F'; V'; C'; R'\rangle
\end{array}
$$

Fig. 17: Functions on Physical Cache Unit. For all definitions, $\psi = \langle F; V; C; R\rangle$, and $C(c) = \langle vb; t; D\rangle$

$$
\begin{array}{rcl}
\kappa\{\lambda, c \mapsto vb\} & \triangleq & \kappa[\lambda \mapsto \kappa(\lambda)\{c \mapsto vb\}] \\
\kappa\{\lambda, c \mapsto D\} & \triangleq & \kappa[\lambda \mapsto \kappa(\lambda)\{c \mapsto D\}] \\
\kappa|_{\overline{\mathsf{e}}} & \triangleq & \overline{\lambda \mapsto \psi|_{\overline{\mathsf{e}}}} \\
 & \text{if} & \kappa = \overline{\lambda \mapsto \psi} \\
 & & \\
\kappa \# \kappa' & \triangleq & \forall\ \lambda \in dom(\kappa) \cap dom(\kappa').\kappa(\lambda)\#\kappa'(\lambda)
\end{array}
$$

Fig. 18: Functions on Multi-Level Cache

$$
\begin{array}{rcl}
\mu|_E & \triangleq & \mu|_{\langle l_1; n_1\rangle} \cdots \cup \mu|_{\langle l_n; n_n\rangle} \\
 & \text{if} & E = [\mathsf{e}_1 \mapsto \langle l_1; n_1\rangle, \ldots, \mathsf{e}_n \mapsto \langle l_n; n_n\rangle] \\
\mu|_{\langle l; n\rangle} & \triangleq & \mu|_{\{l, l+1, \ldots, l+n-1\}} \\
\triangledown^{\mathsf{e}}_{\epsilon}\mu & \triangleq & \mu\{l \mapsto 0\} \ldots \{l+n-1 \mapsto 0\} \\
 & \text{if} & \epsilon = \langle e; E\rangle, E(\mathsf{e}) = \langle l; n\rangle \\
\mu_1 \# \mu_2 & \triangleq & dom(\mu_1) \cap dom(\mu_2) = \emptyset
\end{array}
$$

Fig. 19: Functions on Memory

$$
\begin{array}{rcl}
\epsilon_\mu\{\mathsf{e} \mapsto \langle l; n\rangle\} & \triangleq & \langle e; E[\mathsf{e} \mapsto \langle l; n\rangle]\rangle \\
 & \text{if} & \epsilon = \langle e; E\rangle, \mathsf{e} \notin dom(E), \\
 & & \mu\{l\} = \mu\{l+1\} \ldots \mu\{l+n-1\} = 0 \\
\langle e'; E\rangle \blacktriangleleft e & \triangleq & \langle e; E\rangle \\
 & \text{if} & e \in dom(E) \cup \{\perp\} \\
\langle e; E\rangle - \mathsf{e} & \triangleq & \langle e; E \backslash \mathsf{e}\rangle
\end{array}
$$

Fig. 20: Functions on Enclave

$$
\begin{array}{rcl}
\Sigma[l \mapsto n] & \triangleq & \langle \kappa; \mu\{l \mapsto n\}; \rho; \pi\rangle \\
\Sigma|_p & \triangleq & \langle \kappa|_{dom(E)}; \mu|_E; \rho|_p; p \mapsto \pi(p)\rangle \\
 & \text{if} & \pi(p) = \langle \epsilon; l; q\rangle, \epsilon = \langle \mathsf{e}; E\rangle
\end{array}
$$

Fig. 21: Convenience Functions on Runtime States. For all definitions, $\Sigma = \langle \kappa; \mu; \rho; \pi\rangle$

$$
\frac{\begin{array}{c}
\tau = \langle l; a; \epsilon\rangle \\
\psi = \langle F; V; C; R\rangle \quad \alpha(l) = \langle b; \delta\rangle \quad \beta(b) = \langle s; t\rangle \quad w \in C(\epsilon)(s) \\
c = \langle w; s\rangle \quad C(c) = \langle vb; t; D\rangle \quad T = update(R(s), w, \epsilon)
\end{array}}{\psi \Diamond \tau \triangleq (c, \delta, vb, D, \psi\{s \mapsto T\})}
$$

$$
\frac{\begin{array}{c}
\tau = \langle l; a; \epsilon\rangle \quad \psi = \langle F; V; C; R\rangle \quad \alpha(l) = \langle b; \delta\rangle \\
\beta(b) = \langle s; t\rangle \quad \forall w \in C(\epsilon)(s).t \neq t' \text{ where } C(\langle w; s\rangle) = \langle vb; t'; D'\rangle \\
w' = replace(R(s), \epsilon) \quad c = \langle w'; s\rangle \\
C(c) = \langle \mathsf{I}; t''; D''\rangle \text{ for some } t'' \text{ and } D'' \quad T = update(R(s), w', \epsilon)
\end{array}}{\psi \Diamond \tau \triangleq (c, \delta, vb, \emptyset, \psi\{s \mapsto T\})}
$$

Fig. 22: Location-Cache Lookup (In the first case, the location l is already cached in ψ, within any of the MOESI state. In the second case, the location l is not cached in ψ at all.)

We define $peers(\lambda)$ as $\{\lambda' \mid H(\lambda) = H(\lambda'), \lambda' \neq \lambda\}$. Note that if λ represents an LLC, the function still works, except that it computes $\emptyset$.

D. PCU Behavior

Structurally, a single-level physical cache unit consists of the CPT (V), way-set cache (C), replacement logic (R), and free list (F). Given a memory block ID b, we define a bijective function $\beta : \mathbb{BLOCK} \rightleftharpoons \mathbb{SET} \times \mathbb{TAG}$ to compute its set index s and the tage value t.

Fig. 29 and Fig. 30 define CP allocation and deallocation respectively, where

$$\circlearrowleft F \triangleq \langle w; s\rangle \text{ where } \forall \langle w'; s'\rangle \in F.w' >= w$$

we further define $\circlearrowleft \emptyset$ as $\perp$.

E. TEE-SHirT Behavior

The definitions of TEE-SHirT allocation and deallocation can be found in Fig. 31 and Fig. 32. The write-back reduction is defined in Fig. 33.

F. Full Operational Semantics

The reduction rules are shown in Fig. 34. We use $\Sigma \overset{\Omega}{\Longrightarrow}{}^{*} \Sigma'$ to represent the reflexive and transitive closure of $\overset{\Omega}{\Longrightarrow}$, where Ω is "concatenated": (1) If $\Sigma \overset{\Omega}{\Longrightarrow} \Sigma'$, then $\Sigma \overset{\Omega}{\Longrightarrow}{}^{*} \Sigma'$. (2) If $\Sigma \overset{\Omega}{\Longrightarrow} \Sigma'$ and $\Sigma' \overset{\Omega'}{\Longrightarrow}{}^{*} \Sigma''$, then $\Sigma \overset{\Omega,\Omega'}{\Longrightarrow}{}^{*} \Sigma''$. For convenience, we further include an imaginary instruction `NOP` to take the behavior of cache/memroy write-back. This instruction can be inserted at any step of the reduction.

G. Properties

Definition A.1 (Enclave-Private Location). $\mathrm{epriv}(l, \mathsf{e}, p, \Sigma)$ *hold iff* $\Sigma = \langle \kappa; \mu; \rho; \pi\rangle$ *and* $\pi(p) = \langle \epsilon; l'; q\rangle$ *and* $\epsilon = \langle \mathsf{e}_0; E\rangle$ *and* $E(\mathsf{e}) = \langle l_0; n_0\rangle$ *and* $l_0 \leq l < l_0 + n_0$.

Theorem A.1 (Immunity Against Side-Channel Attacks). *Given* Σ *and some* l, e, p *s.t.* $\mathrm{epriv}(l, \mathsf{e}, p, \Sigma)$, *some* $n_1 \neq n_2$, $p' \neq p$, *two reductions* $\Sigma[l \mapsto n_i] \overset{\Omega_i}{\Longrightarrow}{}^{*} \kappa_i, \mu_i, \rho_i, \pi_i$ *where* $\pi_i(p') = \langle \epsilon_i; \bullet\rangle$ *for i = 1, 2, then* $\epsilon_1 = \epsilon_2$, $\Omega_1|_{p'} = \Omega_2|_{p'}$.

$$\begin{array}{ccc}
\mathsf{M} & \overset{\mathsf{R}}{\rightharpoonup} & \mathsf{M} \\
.\mathsf{M} & \overset{\mathsf{W}}{\rightharpoonup} & \mathsf{M} \\
\mathsf{O} & \overset{\mathsf{R}}{\rightharpoonup} & \mathsf{O} \\
\mathsf{O} & \overset{\mathsf{W}}{\rightharpoonup} & \mathsf{M} \\
\mathsf{E} & \overset{\mathsf{R}}{\rightharpoonup} & \mathsf{E} \\
\mathsf{E} & \overset{\mathsf{W}}{\rightharpoonup} & \mathsf{M} \\
\mathsf{S} & \overset{\mathsf{R}}{\rightharpoonup} & \mathsf{S} \\
\mathsf{S} & \overset{\mathsf{W}}{\rightharpoonup} & \mathsf{M}
\end{array}$$

Fig. 23: Cache Coherence State Transition with MOESI Protocol: Core Behavior (Relation $vb_1 \overset{a}{\rightharpoonup} vb_2$ says that cache coherence state vb_1 transitions to cache coherence state vb_2 when the CPU core itself encounters an event of a.)

$$\begin{array}{ccc}
\mathsf{M} & \overset{\mathsf{R}}{\rightharpoondown} & \mathsf{O} \\
.\mathsf{M} & \overset{\mathsf{W}}{\rightharpoondown} & \mathsf{E} \\
\mathsf{O} & \overset{\mathsf{R}}{\rightharpoondown} & \mathsf{B} \\
\mathsf{O} & \overset{\mathsf{W}}{\rightharpoondown} & \mathsf{M} \\
\mathsf{E} & \overset{\mathsf{R}}{\rightharpoondown} & \mathsf{S} \\
\mathsf{E} & \overset{\mathsf{W}}{\rightharpoondown} & \mathsf{B} \\
\mathsf{S} & \overset{\mathsf{R}}{\rightharpoondown} & \mathsf{S} \\
\mathsf{S} & \overset{\mathsf{W}}{\rightharpoondown} & \mathsf{I}
\end{array}$$

Fig. 24: Cache Coherence State Transition with MOESI Protocol: Bus Behavior (Relation $vb_1 \overset{a}{\rightharpoondown} vb_2$ says that cache coherence state vb_1 transitions to cache coherence state vb_2 when the bus transmits an event of a to the core.)

$$\frac{\psi \lozenge \tau = (c, \delta, vb, D, \psi') \qquad vb \overset{\mathsf{W}}{\rightharpoonup} vb'}{\psi \xhookrightarrow[\langle \mathsf{W}; c; \lambda\rangle]{D', \mathsf{W}, v, \lambda} \psi'\{c \mapsto D[\delta \mapsto v]\}\{c \mapsto vb'\}}$$

$$\frac{\psi \lozenge \tau = (c, \delta, vb, D, \psi') \qquad vb \overset{\mathsf{R}}{\rightharpoonup} vb' \qquad v = D(\delta)}{\psi \xhookrightarrow[\langle \mathsf{R}; c; \lambda\rangle]{D', \mathsf{R}, v, \lambda} \psi'\{c \mapsto vb'\}}$$

$$\frac{\psi \lozenge \tau = (c, \delta, vb, D, \psi') \qquad vb = \mathsf{I} \text{ or } \mathsf{B} \qquad D' \neq \emptyset}{\psi \xhookrightarrow[\langle \mathsf{LM}; c; \lambda\rangle]{D', \mathsf{W}, v, \lambda} \psi'\{c \mapsto D'[\delta \mapsto v]\}\{c \mapsto \mathsf{S}\}}$$

$$\frac{\psi \lozenge \tau = (c, \delta, vb, D, \psi') \qquad vb = \mathsf{I} \text{ or } \mathsf{B} \qquad D' \neq \emptyset \qquad v = D'(\delta)}{\psi \xhookrightarrow[\langle \mathsf{LM}; c; \lambda\rangle]{D, \mathsf{R}, v, \lambda} \psi'\{c \mapsto \mathsf{S}\}}$$

$$\frac{\psi \lozenge \tau = (c, \delta, vb, D, \psi') \qquad vb = \mathsf{I} \text{ or } \mathsf{B}}{\psi \xhookrightarrow[\langle \mathsf{GM}; c; \lambda\rangle]{\emptyset, a, v, \lambda} \psi'\{c \mapsto \mathsf{E}\}}$$

Fig. 25: PCU Access when Request Comes from CPU

$$\frac{\tau = \langle l; a; \epsilon\rangle \qquad \psi \lozenge \tau = (c, \delta, vb, D, \psi') \qquad vb \overset{a}{\rightharpoondown} vb'}{\psi \otimes \tau \triangleq (\psi'\{c \mapsto vb'\}, vb, D)}$$

$$\frac{\psi \lozenge \tau = (c, \delta, vb, D, \psi') \qquad vb = \mathsf{I} \text{ or } \mathsf{B}}{\psi \otimes \tau \triangleq (\psi', vb, \emptyset)}$$

Fig. 26: PCU Access When Request Comes from Bus

$$\frac{\begin{array}{c}\overline{\psi} = \{\psi_1, \ldots, \psi_n\} \\ \psi_i \otimes \tau = (\psi_i', vb_i, D_i) \qquad \exists j \in [1..n].vb_j = \mathsf{M}, \mathsf{E}, \text{ or } \mathsf{O}\end{array}}{\overline{\psi} \otimes \tau \triangleq (\overline{\psi'}, D_j)}$$

$$\frac{\begin{array}{c}\overline{\psi} = \{\psi_1, \ldots, \psi_n\} \qquad \psi_i \otimes \tau = (\psi_i', vb_i, D_i) \\ \nexists j \in [1..n].vb_j = \mathsf{M}, \mathsf{E}, \text{ or } \mathsf{O} \qquad \exists k \in [1..n].vb_k = \mathsf{S}\end{array}}{\overline{\psi} \otimes \tau \triangleq (\overline{\psi'}, D_k)}$$

$$\frac{\begin{array}{c}\overline{\psi} = \{\psi_1, \ldots, \psi_n\} \\ \psi_i \otimes \tau = (\psi_i', vb_i, D_i) \qquad \forall j \in [1..n].vb_j = \mathsf{I} \text{ or } \mathsf{B}\end{array}}{\overline{\psi} \otimes \tau \triangleq (\overline{\psi'}, \emptyset)}$$

Fig. 27: Single Cache Level Access When Request Comes from Bus

$$\frac{\mathit{peers}(\lambda) = \overline{\lambda} \qquad \overline{\kappa(\lambda)} \otimes \tau = (\overline{\psi'}, D) \\ \kappa(\lambda) \xhookrightarrow[o]{D,a,v,\lambda} \psi}{\kappa \xrightarrow[\{o\}]{\lambda,v,\tau} \kappa[\lambda \mapsto \psi][\overline{\lambda \mapsto \psi'}]}$$

Fig. 28: Single Cache Level Access

$$\frac{c =\circlearrowleft F \qquad \mathsf{e} \notin \mathit{dom}(V) \qquad c = \langle w; s\rangle \qquad T = \mathit{update}(R(s), w, \mathsf{e})}{\Uparrow_{\mathsf{e}}^{n} \langle F; V; C; R\rangle \overset{\triangle}{=} \Uparrow_{\mathsf{e}}^{n-1} \langle F - c; V[\mathsf{e} \mapsto (s \mapsto \{w\})]; C; R[s \mapsto T]\rangle}$$

$$\frac{V(\mathsf{e}) = L \qquad s \notin \mathit{dom}(L) \qquad \begin{array}{c} c =\circlearrowleft F \\ c = \langle w; s\rangle \end{array} \qquad T = \mathit{update}(R(s), w, \mathsf{e})}{\Uparrow_{\mathsf{e}}^{n} \langle F; V; C; R\rangle \overset{\triangle}{=} \Uparrow_{\mathsf{e}}^{n-1} \langle F - c; V[\mathsf{e} \mapsto L[s \mapsto \{w\}]]; C; R[s \mapsto T]\rangle}$$

$$\frac{V(\mathsf{e}) = L \qquad L(s) = W \qquad \begin{array}{c} c =\circlearrowleft F \\ c = \langle w; s\rangle \end{array} \qquad T = \mathit{update}(R(s), w, \mathsf{e})}{\Uparrow_{\mathsf{e}}^{n} \langle F; V; C; R\rangle \overset{\triangle}{=} \Uparrow_{\mathsf{e}}^{n-1} \langle F - c; V[\mathsf{e} \mapsto L[s \mapsto W \cup \{w\}]]; C; R[s \mapsto T]\rangle}$$

$$\frac{}{\Uparrow_{\mathsf{e}}^{0} \psi \overset{\triangle}{=} \psi}$$

Fig. 29: CP Allocation in a PCU

$$\frac{V(\mathsf{e}) = L \qquad L(s) = W, w \text{ for some } s, w \qquad c = \langle w; s\rangle \\ T = \mathit{update}(R(s), w, \mathsf{e}) \qquad \psi = \langle F, c; V[\mathsf{e} \mapsto L[s \mapsto W - w]]; C; R\rangle}{\Downarrow_{\mathsf{e}} \langle F; V; C; R\rangle \overset{\triangle}{=} \Downarrow_{\mathsf{e}} \psi\{c \mapsto \mathsf{I}\}\{s \mapsto T\}}$$

$$\frac{V(\mathsf{e}) = L \qquad L(s) = \emptyset}{\Downarrow_{\mathsf{e}} \langle F; V; C; R\rangle \overset{\triangle}{=} \Downarrow_{\mathsf{e}} \langle F; V[\mathsf{e} \mapsto L \backslash s]; C; R\rangle}$$

$$\frac{V(\mathsf{e}) = \emptyset}{\Downarrow_{\mathsf{e}} \langle F; V; C; R\rangle \overset{\triangle}{=} \langle F; V \backslash \mathsf{e}; C; R\rangle}$$

Fig. 30: CP Deallocation in a PCU

$$\frac{\overline{n} = n, \overline{n'} \qquad \Uparrow_{\mathsf{e}}^{n} \kappa(\lambda) = \psi \qquad \kappa' = \kappa[\lambda \mapsto \psi]}{\Nearrow_{\mathsf{e}}^{\overline{n}} \langle \kappa; \lambda\rangle \overset{\triangle}{=} \Nearrow_{\mathsf{e}}^{\overline{n'}} \langle \kappa'; H(\lambda)\rangle}$$

$$\frac{}{\Nearrow_{\mathsf{e}}^{\emptyset} \langle \kappa; \top\rangle \overset{\triangle}{=} \kappa}$$

Fig. 31: TEE-SHirT Allocation

$$\frac{\Downarrow_{\mathsf{e}} \kappa(\lambda) = \psi \qquad \kappa' = \kappa[\lambda \mapsto \psi] \qquad \Searrow_{\mathsf{e}} \langle \kappa'; H(\lambda)\rangle = \kappa''}{\Searrow_{\mathsf{e}} \langle \kappa; \lambda\rangle \overset{\triangle}{=} \kappa''}$$

$$\frac{}{\Searrow_{\mathsf{e}} \langle \kappa; \top\rangle \overset{\triangle}{=} \kappa}$$

Fig. 32: TEE-SHirT Deallocation

$$\frac{H(\lambda_1) = \lambda_2 \qquad \kappa(\lambda_i) = \langle F_i; V_i; C_i; R_i\rangle \text{ for } i = 1, 2 \\ C_i(c_i) = \langle vb_i; t; D_i\rangle \text{ for } i = 1, 2 \text{ and some } t \\ c_i = \langle w_i; s\rangle \text{ for } i = 1, 2 \text{ and some } s \qquad vb_1 = \mathsf{B}}{\kappa, \mu \rightarrow_{\mathtt{wb}} \kappa\{\lambda_1, c_1 \mapsto \mathsf{I}\}\{\lambda_2, c_2 \mapsto \mathsf{B}\}\{\lambda_2, c_2 \mapsto D_1\}, \mu}$$

$$\frac{H(\lambda) = \top \qquad \kappa(\lambda) = \langle F; V; C; R\rangle \\ C(c) = \langle \mathsf{B}; t; D\rangle \qquad c = \langle w; s\rangle \qquad \mu(b) = D' \qquad \beta(b) = \langle s; t\rangle}{\kappa, \mu \rightarrow_{\mathtt{wb}} \kappa\{\lambda, c \mapsto \mathsf{I}\}, \mu[b \mapsto D]}$$

Fig. 33: TEE-SHirT Write-Back

[LOAD]

$$\frac{\tau = \langle l; \mathsf{R}; \epsilon\rangle \qquad \kappa, \mu \xRightarrow[O]{H(q),v,\tau} \kappa', \mu'}{\kappa, \mu, \rho, \epsilon, \mathtt{LOAD}\ l\ r \xrightarrow[q,1]{O} \kappa', \mu', \rho[r \mapsto v], \epsilon}$$

[CREATE]

$$\frac{\epsilon' = \epsilon_{\mu}\{\rho(r_1) \mapsto \langle \rho(r_2); \rho(r_3)\rangle\}}{\kappa, \mu, \rho, \epsilon, \mathtt{CREATE}\ r_1\ r_2\ r_3\ \overline{r} \xrightarrow[q,1]{\emptyset} \Nearrow_{\rho(r_1)}^{\overline{\rho(r)}} (\kappa, H(q)), \mu, \rho, \epsilon'}$$

[STORE]

$$\frac{\tau = \langle l; \mathsf{W}; \epsilon\rangle \qquad \kappa, \mu \xRightarrow[O]{H(q),\rho(r),\tau} \kappa', \mu'}{\kappa, \mu, \rho, \epsilon, \mathtt{STORE}\ r\ l \xrightarrow[q,1]{O} \kappa', \mu', \rho, \epsilon}$$

[DESTROY]

$$\frac{}{\kappa, \mu, \rho, \varepsilon, \mathtt{DESTROY}\ r \xrightarrow[q,1]{\emptyset} \Searrow_{\rho(r)} (\kappa, H(q)), \nabla_{\varepsilon}^{\rho(r)} \mu, \rho, \varepsilon - \rho(r)}$$

[ENTER]

$$\frac{}{\kappa, \mu, \rho, \varepsilon, \mathtt{ENTER}\ r \xrightarrow[q,1]{\emptyset} \kappa, \mu, \rho, \varepsilon \blacktriangleleft \rho(r)}$$

[EXIT]

$$\frac{}{\kappa, \mu, \rho, \epsilon, \mathtt{EXIT} \xrightarrow[q,1]{\emptyset} \kappa, \mu, \rho, \epsilon \blacktriangleleft \bot}$$

[BRTRUE]

$$\frac{\rho(r) \neq 0}{\kappa, \mu, \rho, \epsilon, \mathtt{BR}\ r\ r' \xrightarrow[q,\rho(r')]{\emptyset} \kappa, \mu, \rho, \epsilon}$$

[BRFALSE]

$$\frac{\rho(r) = 0}{\kappa, \mu, \rho, \epsilon, \mathtt{BR}\ r\ r' \xrightarrow[q,1]{\emptyset} \kappa, \mu, \rho, \epsilon}$$

[WB]

$$\frac{\kappa, \mu \rightarrow_{\mathtt{wb}} \kappa', \mu'}{\kappa, \mu, \rho, \epsilon, \mathtt{NOP} \xrightarrow[q,0]{\emptyset} \kappa', \mu', \rho, \epsilon}$$

[NOP]

$$\frac{}{\kappa, \mu, \rho, \epsilon, \mathtt{NOP} \xrightarrow[q,0]{\emptyset} \kappa', \mu', \rho, \epsilon}$$

Fig. 34: Single-Process Operational Semantics