Secure Caches for Compartmentalized Software

Kerem Arıkan[1], Huaxin Tang[1], Williams Zhang Cen[1],
Yu David Liu[1], Nael Abu-Ghazaleh[2] and Dmitry Ponomarev[1]

[1]*Binghamton University*
[2]*University of California, Riverside*

Abstract

Compartmentalized software systems have been recently proposed in response to security challenges with traditional process-level isolation mechanisms. Compartments provide logical isolation for mutually mistrusting software components, even within the same address space. However, they do not provide side-channel isolation, leaving them vulnerable to side-channel attacks. In this paper, we take on the problem of protecting compartmentalized software from hardware cache side-channel attacks. We consider unique challenges that compartmentalized software poses in terms of securing caches, which include performance implications, efficient and secure data sharing, and avoiding leakage when shared libraries are called by multiple callers. We propose SCC - a framework that addresses these challenges by 1) multi-level cache partitioning including L1 caches with a series of optimizations to minimize performance impact; 2) the concept of domain-oriented partitioning where cache partitions are created per memory domain, instead of per compartment; and 3) creating a separate partition instance of a shared library code for each caller. We formally prove the security of SCC using operational semantics and evaluate its performance using the gem5 simulator on a set of compartmentalized benchmarks.

1 Introduction

Modern programs are becoming increasingly complex software systems that often integrate code developed by independent and mutually untrusting parties. The interactions between code components often take place using insecure interfaces [7,16,26,34,50,53,63,65,72,75,82,84,85,90,97,104]. As an example, consider a browser that incorporates a just-in-time compilation module that compiles and executes a web application which is linked to a cryptographic library storing secret keys. In this case, a malicious web application can potentially compromise secrets held in the cryptographic library, as well as the data of the compilation module itself. Traditional process-centric isolation security models are insufficient to protect systems and applications from such vulnerabilities within the same address space.

In response to these emerging threats, several *in-process compartmentalization* mechanisms have been developed to provide intra-process memory isolation [2, 4, 19, 39, 40, 42, 66, 70]. In the above browser example, in-process isolation can prevent the web application code from accessing memory regions of the cryptographic library or the compilation engine, creating isolated compartments in memory within a single process. In-process compartmentalization solutions come in various forms, including secure enclaves (such as Intel SGX [1, 19]), page-based memory access control schemes [75], or capability-based systems [90]. Regardless of the implementation and security principles behind these designs, current proposals are developed around protecting memory accesses and ensuring computation integrity. Unfortunately, side-channel attacks through shared hardware resources have remained outside the threat model these solutions consider. At the same time, side-channel attacks, particularly those exploiting shared caches, have been demonstrated in diverse software environments that benefit from in-process compartmentalization, including browsers [30, 64, 77], cloud services [71], virtual machines [32, 44, 102], and graphics frameworks [86].

In this paper, we propose new cache hierarchies that protect compartmentalized software from side-channel attacks; to the best of our knowledge, this is the first paper that defines and explores this problem. At a high level, our solution augments existing memory protection schemes with *fine-grain cache partitioning*, where in-process compartments can control the data that is isolated from other compartments in the cache across multiple cache levels. Designing efficient and secure partitioned caches supporting compartment isolation requires solving several performance, functionality, and security-related challenges, which we describe next.

First, applying fine-grain partitioning to L1 caches is expensive because L1 caches are accessed frequently, have stringent latency constraints, and are likely to be on the critical path of execution. Previous work leverages flushing the L1 cache on a

context switch or a system call to avoid partitioning overheads in traditional systems [3, 5, 11, 28, 73], while using partitioning for lower-level caches for security. However, flushing the L1 cache on compartment switches is too expensive for in-process compartments since compartment switches can occur frequently (Section 3.1). Instead, we propose performance-friendly partitioning of L1 caches, using locality-aware optimizations. The new design supports leakage-free L1 caches in compartment-based systems with only a modest impact on performance.

Second, compartments of the same program often share data with other compartments. From the standpoint of a partitioned cache design, a naive compartment-oriented scheme (where a partition is created for each compartment) will introduce significant performance inefficiencies, security problems, and complications for cache coherence protocols. To address these issues, we propose the concept of *domain-oriented partitioning*, which shifts the partitioning boundaries to be around secure memory regions (referred to as domains, Section 3) rather than software compartments.

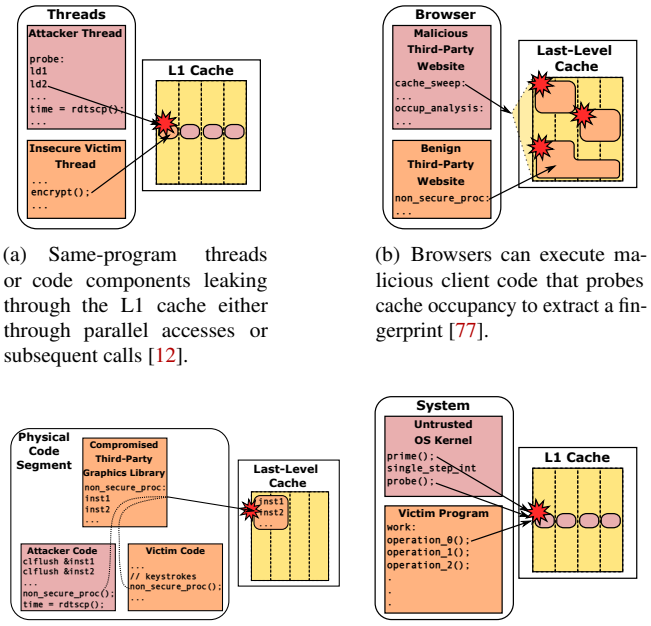
Third, for complete protection, we consider information leakage caused by external code usage. For example, calling commonly used libraries can leak secrets through cache side-channels even when the attacker does not share common data with the victim [33, 86, 95]. This threat is exacerbated for compartmentalized software due to the crossing of compartment boundaries. To this end, our design allows an instance of a compartmentalized library to be maintained in the cache for each of its callers, effectively preventing library call leakages. In the rest of the paper, we refer to our design as **S**ecure **C**aches for **C**ompartments (SCC).

We describe the hardware architecture of SCC, develop a formal model based on operational semantics to prove its security and evaluate its performance using the gem5 cycle-accurate microarchitectural simulator. Our results show that protection from cache side-channel attacks can be achieved with 7% performance loss on average for compartmentalization schemes that support coarse-grained memory protection. SCC incurs area and power overhead below 1%.

2 Threat Model and Assumptions

We consider compartmentalized programs where compartments are mutually mistrusting. Specific attacks that we consider include the following code components vulnerable to side-channels: browser clients [30, 64, 77], cloud services [71], compromised shared libraries [32, 37, 86, 95], APIs [30, 64], untrusted OS [12, 37, 61], virtual machines [32, 44, 102], as well as enclaves in TEE systems such as Intel SGX [12, 20, 29, 51, 61, 62, 100].

Figure 1 shows four recent cache-based attacks in multi-component systems. Figure 1(a) shows a scenario where two threads of a program run on a Simultaneous Multi-Threading (SMT) core and leak sensitive information from the victim to



(a) Same-program threads or code components leaking through the L1 cache either through parallel accesses or subsequent calls [12].

(b) Browsers can execute malicious client code that probes cache occupancy to extract a fingerprint [77].

(c) A library accessed by multiple callers can cause collisions in the cache [86].

(d) Adversarial OS probing high-level caches [37].

Figure 1: Variations of cache side-channel attacks in multi-component software environments.

the attacker in a TEE-based system [12]. Figure 1(b) shows a cache occupancy attack where malicious client code probes total cache occupancy, obtaining a fingerprint of the website being accessed by the victim [77]. Figure 1(c) shows an attack in which an attacker invokes functions in a graphics library to infer timing information about other callers by probing the time it takes to complete utility functions [86], inferring their keystrokes. Finally, Figure 1(d) illustrates an attack in which the untrusted OS utilizes single stepping to control the victim workflow and probe each operation [37]. Even though compartmentalization schemes take security measures against an adversarial OS, none assumes an OS that exploits external library dependencies for cache attacks. Therefore, we develop systems to protect caches even under kernel interrupts.

We do not consider cross-compartment leakage that can occur through program direct/indirect flows. For example, if a secret value is used to control the access to data that is shared between the victim and the malicious compartments, the secret-dependent access patterns to shared data can be established by the attacker without side channels. We assume that it is the responsibility of the baseline compartmentalized system to avoid such leakage either through programming or compiler support [34, 59].

3 Mitigating Cache Side-Channel Attacks on Compartments: SCC

Conventional programs perform arbitrary accesses to data within their user-level address space without authorizing the permissions of the code components, a model called *ambient*

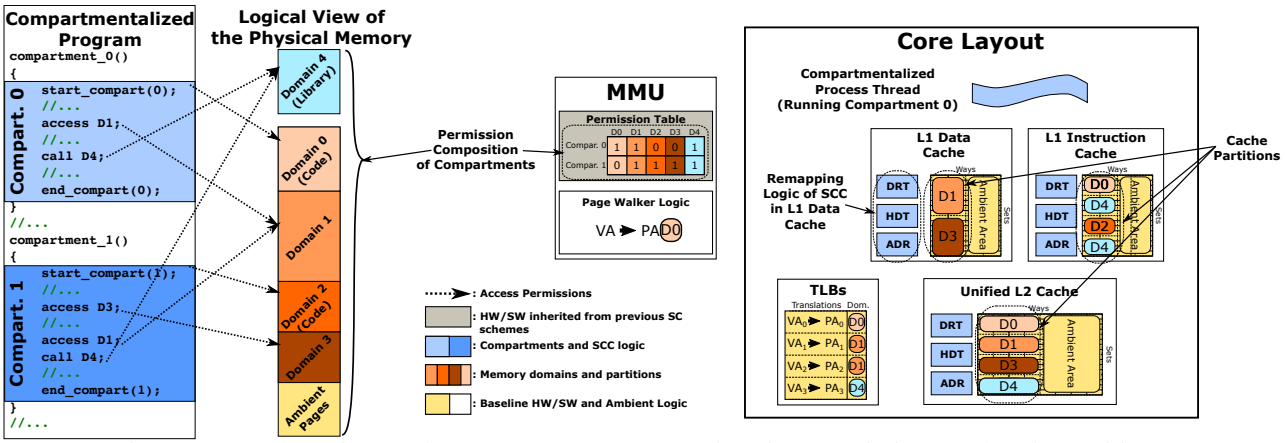


Figure 2: SCC Overview: code compartments, memory domains, permissions, and cache partitions.

authority. However, the ambient authority model exposes the program to potential attacks if any of its untrusted components is compromised. *Software Compartmentalization (SC)* is an emerging paradigm that is designed to mitigate security issues of ambient authority by segregating privileges of different code components by structurally modifying the program [7,16,26,34,50,53,59,63,65,72,75,82,84,85,90,97,104]. The components of the program are disassembled into isolated entities called *compartments*. To control access rights of these compartments, memory is divided into subregions, where each region is associated with a set of permissions given to compartments. SC literature refers to these memory regions using different terms; we use *domains*. SC systems typically allow domains and ambient memory areas (areas unprotected beyond traditional means) to coexist, assuring compatibility with legacy code. Each domain is associated with a set of permissions that define specific compartments that are allowed to access data in that domain. This allows domains to be exclusively allocated to specific compartments while being isolated from other compartments. In this section, we describe SCC - an architecture that augments SC memory protection schemes with secure cache hierarchies (based on cache partitioning) to thwart in-process cache-based side-channel attacks.

Figure 2 depicts the high-level overview of SCC and the relationship between compartments, domains, and cache partitions. This example considers a process with two compartments and five domains. Domains D0 and D2 are exclusive code domains belonging to compartments C0 and C1, respectively. D1 is a data domain and is shared by C0 and C1. D3 is an exclusive domain that belongs to C1. D4 is an external library code domain that is accessible by both compartments. We also assume that the system allows ambient code to coexist with protected compartments at runtime.

SCC partitions the cache space based on memory domains rather than code components (i.e. compartments), an approach detailed in Section 3.2. To support isolation of domains in caches, SCC equips each cache with three hardware structures: 1) Domain Remapping Table (DRT) - the structure that keeps the address remapping metadata (Section 3.1), 2) Active Domain Register (ADR) - the structure that maintains a single recently used entry from the DRT for optimization purposes (Section 3.1), and 3) Horizontal Domain Table (HDT) - the structure that enables horizontal compartmentalization (Section 3.3). These structures are used in combination to direct memory domain accesses to their respective cache partitions. SCC implements set-based partitioning for scalability, where each partition is composed of several consecutive sets, and the number of sets in a partition is a power of two.

The cache ways are statically divided into areas, one for domain-specific data and the other for ambient data. For example, in the L1 data cache, two partitions are designated for domains D1 and D3 in ways 0 and 1, while ways 2 and 3 are reserved for the ambient area. Allocating separate space for the ambient area allows ambient accesses to be unimpacted by the partitioning logic and preserves the performance of program sections that rely on ambient authority.

SCC ensures seamless compartment switches by retaining partitions in the cache. When a compartment is switched out, its private domains are preserved in the cache for future use as long as the cache capacity is not exceeded, avoiding unnecessary cache evictions. In the scenario shown in Figure 2, compartment C0 is currently running on the core, and the private domain D3 (along with its associated data) is stored in the cache. This allows low-overhead compartment switches as compartment C1 does not have to reallocate its domains and repopulate the cache once it starts executing.

The permission configuration for the domains is stored in a permission table integrated into the MMU. This table resembles the mechanisms used by other SC schemes with some important differences [7, 16, 75, 85]. Permission tables are typically used for boundary checks during page walks. This implies that recently used domain permission metadata is kept in some form of cache for performance reasons. These caching mechanisms can be implemented in the form of dedicated hardware structures [75], TLB extensions [7], or a com-

bination of both [85]. SCC uses a TLB-based approach, where TLB entries are extended to contain domain metadata along with address translation information (Section 3.1).

3.1 Latency-Aware Partitioned L1 Caches

Cross-compartment attacks primarily target the L1 caches, as co-location is naturally established when both the attacker and the victim operate within the same process. Since L1 caches are on the critical path of execution in modern processors, the timing impact of securing them must be low. One simple and low-complexity mitigation strategy for protecting L1 caches is to flush them during switches between mistrusting parties (compartments). While flushing L1 caches is feasible for regular programs with monolithic threat models [3, 11, 28], it entails significant performance degradation for compartmentalized programs. The reason is that frequent cross-compartment calls result in severe underutilization of the cache space (shown in Appendix 2) and low hit rates, due to frequent cache flushes. Figure 18 (in Section 5.2) shows the performance impact of flushing caches on compartments for our benchmarks - on average, there is 60% performance degradation.

To support secure L1 caches without relying on flushes, we propose latency-aware partitioning. To implement this, the original cache addresses have to be remapped to the addresses within a targeted partition. To this end, SCC maintains a hardware structure called Domain Remapping Table (DRT). Each entry in the DRT consists of three fields: domain metadata, partition ID, and partition mask. When a domain is accessed, the DRT is consulted to determine its location in the cache. Since the DRT access is needed to determine the partition set index, it adds to the latency of L1 cache accesses.

Figure 3 shows the timeline of a cache access with SCC compared to the traditional Virtually Indexed Physically Tagged (VIPT) L1 caches. In the baseline VIPT, cache access is overlapped with a TLB access by using bits from the virtual address directly to index the cache set (Baseline VIPT in Figure 3). With domain-oriented partitioning of SCC, determining which physical page (and thus, the domain ID) is being accessed is only possible after the address translation completes either through a TLB access or through a page table walk. Therefore, the DRT access occurs only after the TLB access is complete, and the cache access itself can only start after the DRT access is complete (Unoptimized SCC in Figure 3). The result here is sequential access to the TLB, the DRT, and the cache, likely resulting in two additional cycles to the L1 cache access compared to the baseline VIPT cache. This has a significant performance impact. To address this problem, we observe that *domain accesses typically exhibit a strong locality of references: the ID of the accessed domain is very likely to match that of the previous access*. We can therefore predict that the same domain ID will be used again and speculatively perform the access before the

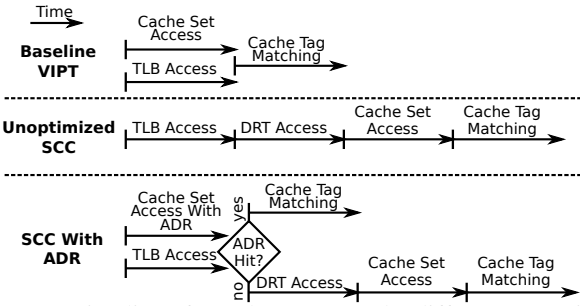


Figure 3: Timeline of a cache access under different scenarios.

TLB translation is completed. To support such predictive accesses, SCC maintains a register that holds the most recently accessed DRT entry. This register is called an Active Domain Register (ADR). In case of domain mispredictions, the correct DRT entry is retrieved and is recorded in ADR (SCC with ADR in Figure 3). This optimization eliminates sequential TLB→DRT→cache assesses and only adds two gates to the critical path compared to VIPT caches.

Figure 5 shows the domain prediction accuracy with ADR for L1 instruction (left plot) and data (right plot) caches. The L1 instruction cache has above 90% prediction accuracy across all programs. However, the ADR hit rate of the data cache is workload-dependent as domains are more frequently switched in the data space. Indeed, workloads with a high number of domains such as *lame*, *jpeg*, *rijndael*, and *nab* exhibit higher misprediction rates: 31%, 30%, 68%, 32%, and 32% respectively. The increased latency introduced during instruction fetch (during L1 instruction cache accesses) is more critical for performance due to the pipeline stalls. At the same time, the extra cycles required for L1 data cache accesses can often be hidden by out-of-order execution. Performance results presented in Section 5.2 account for all these effects.

SCC inherits TLB entry extensions proposed by prior work [7, 85]. The metadata kept in these additional fields of the TLB entries are implementation-specific; these can be details such as protection keys or capabilities depending on the compartmentalization scheme. This is orthogonal to the cache protection (the main goal of this work) but is needed for the complete design to work.

Figure 4 depicts the cache access dataflow for a 4-way and 16-set L1 cache with four partitions. As shown in ❶, the VIPT cache access starts at the same time as the TLB access. For the set index to fit into a location within the partition boundary, SCC transforms the set index into the effective set index through a partition ID and a partition mask. Annotation ❷ depicts the combinational logic used to remap accesses within the partition. The higher bits of the set index are omitted by using the active partition mask. This operation creates a partition offset, which then gets prefixed by the *partition ID* to generate the effective set index. This extension adds only two simple gates to the logic. In Section 5.3, we demonstrate how the latency of these two gates can be hidden behind the banking and subarray access logic of a typical L1 cache.

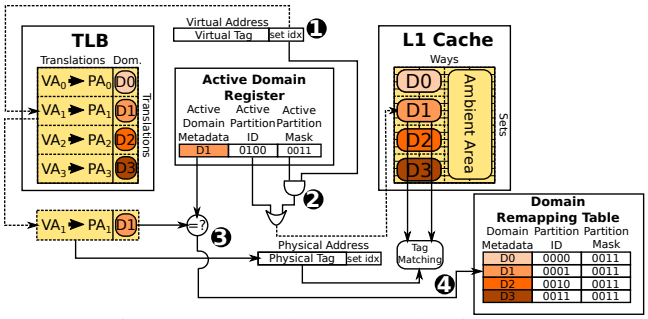


Figure 4: L1 Cache Access Flow in SCC.

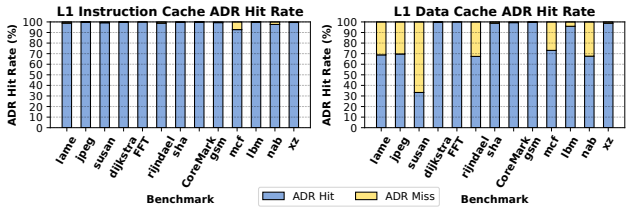


Figure 5: ADR hit rate for L1 caches.

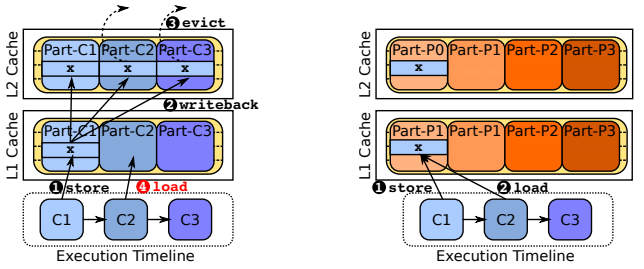
After the set index remapping operation completes, the new index now points to the appropriate partition. Note that ambient accesses go through the traditional set indexing logic and do not incur any additional access latency. Since cache partitions have fewer sets than the entire cache, larger tags are required in combination with smaller indexes to avoid tag aliasing. This is common to all set-based cache partitioning schemes [3, 73, 81]. We evaluate the area overhead incurred by additional tag bits in Section 5.3.

The correctness of the domain prediction must be verified after the TLB entry is accessed, as shown in ❸. If the domain metadata field within the TLB does not match the ADR, then the access has to be aborted and repeated for the correct domain. In this case, the DRT is accessed to retrieve the actual domain's partition ID to be placed in the ADR as shown in ❹. ADR mispredictions impact performance, but they do not occur frequently.

ADR speculations do not cause leakage across domains since cache accesses through the ADR do not fetch data into a register until the domain security check is complete. ADR accesses only target a speculated cache block, and the access is invisible to software until the value is written to a register. However, it is possible to infer the last accessed domain by probing the access latency differences caused by the ADR. Therefore, the ADR must be flushed upon compartment switches to prevent cross-compartment leakage.

3.2 Domain-Oriented Cache Partitioning

Isolation schemes that target monolithic software typically assume that page sharing between programs is minimal or non-existent. This allows such systems to tolerate the infrequent delegation overhead when data sharing is needed [60, 98]. However, with our threat model, compartments access common pages through varying sets of permissions and continuously interact with one another (after all, they are part of the same process). This means that SCC should allow flexible sharing among compartments while complying with permission composition. Compartments may have exclusive and shared partitions coexisting in the cache as separate domains.



(a) Subsequent compartments will use common data lines, which renders partitioning on a compartment level infeasible.

(b) When partitions are established, data lines are mapped to the same boundary regardless of compartment.

Figure 6: Compartment-oriented partitioning vs DOP.

Figure 6(a) shows the leakage and performance inefficiencies caused by having a single domain per compartment. In this example, compartments C1, C2, and C3 all have permission to line `X` and are executed in that order. C1 writes to `X` and then gets switched by C2, which loads line `X` again. Since all compartments have access to `X`, `X` should be written to all partitions of other compartments, introducing both performance and security issues. From a performance standpoint, multiple lines must be written back to multiple partition domains, increasing cache pressure. From a security perspective, write-backs may evict lines from exclusive pages, creating opportunities for attackers to extract cross-domain eviction sets.

In response, we introduce the notion of *Domain-Oriented Partitioning* (DOP). As opposed to compartment-oriented partitioning, DOP ensures that caches provision partitions that are allocated per domain rather than per compartment. Figure 6(b) shows an example of DOP, where four partitions with different permissions coexist in the cache: P0 is a partition shared by all compartments, and P1, P2, and P3 are private partitions for compartments C1, C2, and C3 respectively. If `X` is written to the shared partition P0, it does not evict `X` from any private partitions, as evictions are limited only to partitions with the same permission composition.

When a new domain is introduced to the system, the partition layout of caches has to be dynamically readjusted. While each introduction (or deletion) of a domain results in flushing of some part of the domain area of the cache to make space, other permission operations such as permission grants, revocations, transfers, and page allocations do not affect the partition layout.

To integrate a new partition into the cache, SCC shrinks the size of the largest currently allocated partition by half and allocates the new partition to the newly freed-up area. If all partitions have the same size, SCC shrinks the oldest allocated

partition. Since the partition size only depends on the current layout configuration, it does not leak any secret-dependent information.

We illustrate the partition layout reorganization process in SCC in Figure 7, where domains are progressively introduced to a cache with 16 sets. During the allocation of a new partition, only the bottom half of the victim partition has to be flushed and the rest of the cache remains intact. For example, when D1 is allocated, the lines in sets 8 to 15 are evicted and invalidated, while sets 0 to 7 continue to retain data from partition D0; those lines do not need to be evicted or reallocated in the cache because their addressing does not change. All changes occurring during layout reorganization are reflected in the DRT to maintain consistency.

While the partition allocation process may leak information, this leakage is coarse-grained and not dependent on secret data, as established in Section 2. Furthermore, the leakage is bounded by the number of domains that are maximally created for a given application. For example, if only 8 domains are used, then only 8 layout adjustments will be made, most likely in the initial stages of the execution. Once all domains are allocated, they will remain in the cache without further reorganization throughout the rest of the execution. Although the resizing-induced leakage is small and difficult to exploit, it can be mitigated by various techniques such as decoupling the timing of partition allocation decision from actual allocation [103] or even pre-allocating partitions ahead of time.

SCC can be integrated with several specific SC implementations. Table 1 summarizes these schemes. SCC supports page, range, and VMA-level granularity schemes. Prior works on process-level cache isolation [3, 5, 21] demonstrated effective schemes for managing cache partitions across multiple cache levels. We discuss how SCC can be integrated with multi-level cache isolation schemes in Section 3.4. Current SC schemes typically impose a hardware limit on the number of memory domains. For example, the maximum number of domains supported by Intel MPK [42] (also the same as ARM Memory Domains [4]) and IBM Power [40] are 16 and 32 respectively. Our current implementation also imposes similar limits on the number of cache partitions (we assume the limit of 64 in our evaluations).

In multi-threaded executions, the same compartment—a code entity—could potentially be called from different threads with multiple *concurrent* instances. If they happen to interleave, race conditions would occur for DRT maintenance. To address this, SCC *seals* (i.e., locks) the compartment upon its entry and *unseals* (i.e., unlocks) the compartment upon its exit. This is implemented by extending the domain metadata with a seal bit. Sealing/unsealing ensures correctness for SCC, and interestingly, pays a minimal price in performance. The latter is due to the fact that it SCC, if a compartment is meant to be called from multiple threads, horizontal compartmentalization is used, as we shall see next.

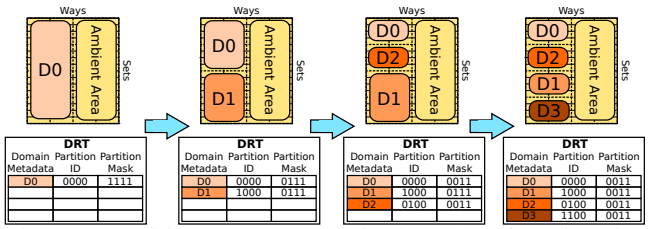


Figure 7: Partition layout reorganization when four domains are created sequentially.

SC System	Granularity	Dom. Management	Design Space
CODOMs [85]	Range/Page*	Hardware	Capability/MMU
Donky [75]	Page	OS/Hardware	MPK/MMU
LOTRx86 [53]	Page	OS/Hardware	MPK/MPX/MMU
MemSentry [50]	Page/Byte**	Compiler/ Hardware	MPK/MPX/MMU
Nested Enclaves [65]	Page	Hardware	SGX/MMU
SecureCells [7]	VMA	Hardware	MMU
Shreds [16]	Range	OS/Compiler	MMU

*Hybrid granularity. **Depends on the configuration.
Table 1: SC mechanisms compatible with SCC.

3.3 Horizontal Compartments in Caches

Programs that utilize shared libraries (or call common code) can leak sensitive information through caches even when the two callers share no pages. [33, 86, 95]. We illustrate this issue in Figure 1(c) in Section 2, where two caller compartments invoke the same function in a shared library. Depending on the victim code's function arguments, different instructions from the library are executed (and therefore brought into the instruction cache). By measuring the timing of its own subsequent call to this common function, the attacker can deduce which input-dependent instructions have been fetched into the cache by the victim code [86]. This leakage may also be possible through two objects from the same class invoking the same member function. Furthermore, library call leakage persists even when compartmentalization and cache partitioning are deployed since partitions assigned to a library code still maintain the same code instance for all callers.

To mitigate this problem, we propose to create a separate partition in the instruction caches for each caller. The general concept is known as *horizontal compartmentalization* in the SC literature [34]. We extend this notion to cache partitions to mitigate library call leakages at the cache level. SCC supports manual annotation for publicly used code areas (such as libraries or APIs) to be horizontally compartmentalized.

We call each compartment with this feature enabled *HComp*. We extend the baseline ISA to include `horiz_compart` operation. This instruction allows the programmer to annotate necessary address boundaries within the code segment as an HComp. This instruction also notifies the cache subsystem to treat accesses from HComps to map to different domains called *HDoms*. Access to HDoms is determined by the caller's identity. HDoms are entirely managed by the SCC hardware and are not defined by the compartmentalization framework or the rest of the system. Due to the possibility of a diverse range of callers, SCC has a hard limit on the number of co-existing HDoms for a single

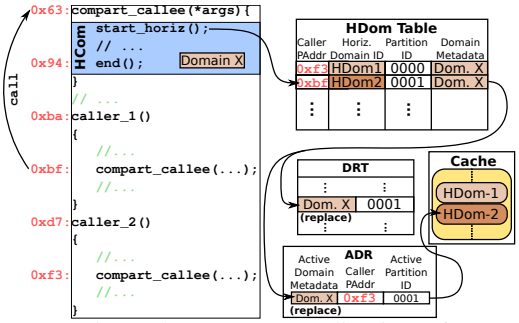


Figure 8: Horizontal compartmentalization of an HComp library in the cache.

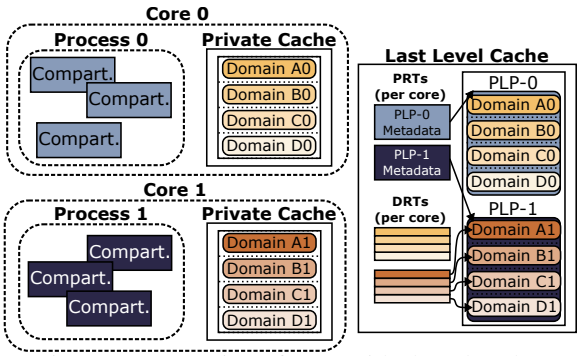


Figure 9: Integration of SCC with shared caches

domain. This may require occasional flushes in caches.

Figure 8 depicts how common code invocations from different callers are remapped into separate partitions. In the figure, there are two callers to the code `compart_callee`, which has two HDoms allocated for it. Here, `caller_2` makes the call before `caller_1`. Therefore, the ADR contains an outdated metadata. To avoid access collision, we include the physical address of the caller instruction in the ADR to indicate the caller's identity. Upon being called, the HComp at address `0x63` accesses an additional hardware called the HDom Table (HDT). The HDT maintains the caller's physical address, HDom ID, partition ID, and domain metadata in each entry to verify access validity. In the example, Domain X has two HDoms since there are two callers to the callee. Accesses to HDoms are redirected to the HDT first rather than the DRT. After the HDT entry is read, the DRT and ADR are updated with the correct partition ID. This process enables caches to divert subsequent accesses within HDom without additional latency. If the HComp has not previously allocated a domain for the caller compartment, it allocates a new entry (thereby a new HDom) in the HDT. Caches flushes are not incurred as long as the HDom cap is not reached. Note that the ADR was augmented with the caller's physical address as a new field.

This mechanism is applicable exclusively to read-only cache lines, where aliasing and data coherence do not present problems. However, write requests to any horizontal partition trigger a full flush of other partitions of the corresponding HDom. This guarantees that dirty and outdated instances of a cache line are not simultaneously retained in the cache.

3.4 Integration with Shared Caches

SCC can be seamlessly integrated with secure shared caches [3,5,21]. In this case, domain-level partitions is nested within traditional process-level partitions (PLPs) that isolate processes from each other. This approach enforces both inter-process and intra-process permission compositions while maintaining isolation within the cache hierarchy.

This approach is illustrated in Figure 9, which shows two compartmentalized processes running on separate cores that share a last-level cache (LLC). Each isolated process is assigned a PLP, containing multiple domain-level partitions. To enforce PLP boundaries, the LLC maintains per-core PLP metadata (remapping tables). In SCC, these tables are only used during the allocation of domain-level partitions to ensure that they are confined within the corresponding PLP. Once a domain-level partition within a PLP is allocated, the existing structures of SCC (DRTs and ADRs) are used to perform cache accesses without going through PLP metadata structures. SCC inherently addresses cache coherence for shared memory, since partitions are created per memory domain, instead of per compartment, thus avoiding multiple copies of shared memory locations in the same cache. Other consistency and scalability issues associated with partitioned private and shared caches can be addressed using mechanisms described in TEE-SHirT [3]. SCC does not introduce any new challenges in this respect.

4 A Formal Security Analysis

We now rigorously establish the security guarantees provided by SCC. Similar to prior work [3], our approach is to define the essential cache-aware program behavior through small-step operational semantics. We present an outline of the formal model here, with a focus on unique aspects of SCC. The full model is included in the Appendix.

4.1 Definitions

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, $\overline{x \mapsto x'}$, a *mapping* when the elements in $\overline{x}$ are distinct. For any mapping M, we use notations $M[x \mapsto x']$, $M \backslash x$, $dom(M)$, $ran(M)$ to refer to the update, restriction, domain, and range of M with standard definitions. We say a sequence X is a non-contiguous subsequence of X', denoted as $X \sqsubseteq X'$ iff there exist sequences $X_1,\ldots X_m, X_{m+1}$ such that $X' = X_1, x_1, X_2, x_2, \ldots, X_m, x_m, X_{m+1}$ where $X = \overline{x}^m$ for some $m \geq 0$.

Compartments, Domains, and Permissions We use metavariable $\chi \in \mathbb{COMPART}$ for compartment labels, $\Delta \in$

$\mathbb{DOMAIN}$ for domain labels. We define permissions $\Phi \in \mathbb{PERM}$ as a function $\mathbb{COMPART} \rightharpoonup \mathcal{P}(\mathbb{DOMAIN})$. In other words, Φ represents the entries in the permission table with value 1. Set $\mathbb{COMPART}$ contains a special value $\perp$ for ambient code. Set $\mathbb{DOMAIN}$ contains an (overloaded) value $\perp$ for the "ambient domain", i.e., memory locations accessible by all compartments, including the ambient code.

SCC Cache Our formal model models the cache hierarchy κ by relating single-level cache units ψ. To simplify formalization, we associate identifiers to single-level cache units ($\lambda \in \mathbb{CU}$) and CPU cores ($q \in \mathbb{CORE}$). Following prior work [3], a cache hierarchy is captured by a static mapping $H : \mathbb{CU} \cup \mathbb{CORE} \to \mathbb{CU} \cup \{\top\}$, 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. H is a total and surjective function, and the relation it defines forms a poset. The definitions in this paper are implicitly parameterized by H.

Each single-level cache is defined as a tuple $\langle F;V;C;R\rangle$, where R is the metadata support for cache replacement, a structure deferred to the Appendix. A standard way-set cache C is a mapping $\mathbb{WAY} \times \mathbb{SET} \to \mathbb{STATUS} \times \mathbb{TAG} \times \mathbb{DBLOCK}$. We use metavariables w,s,vb,t,D to represent the elements in the 5 sets, and use C to represent the cache. We further call $\langle w;s\rangle$ a cache block index, and use metavariable c to represent it. Sets $\mathbb{WAY}$, $\mathbb{SET}$, $\mathbb{TAG}$ are subsets of $\mathbb{NAT}$. Status set $\mathbb{STATUS} = \{\mathsf{VC},\mathsf{VD},\mathsf{IC},\mathsf{ID}\}$ contains 4 labels, denoting the cache block as valid-clean, valid-dirty, invalid-clean, and invalid-dirty, respectively. Metadata V and F, will be elaborated in the next two subsections. Given $\psi = \langle F;V;C;R\rangle$, and $C(c) = \langle vb;t;D\rangle$, we define the following convenience functions for updating various components of the cache:

$$\begin{array}{rcl}
\psi\{c \mapsto vb'\} & \triangleq & \langle F;V;C[c \mapsto \langle vb';t;D\rangle];R\rangle \\
\psi\{c \mapsto D'\} & \triangleq & \langle F;V;C[c \mapsto \langle vb;t;D'\rangle];R\rangle \\
\psi\{c,\delta \mapsto v\} & \triangleq & \langle F;V;C[c \mapsto \langle vb;t;D[\delta \mapsto v]\rangle];R\rangle \\
\psi\{s \mapsto T\} & \triangleq & \langle F;V;C;R[s \mapsto T]\rangle
\end{array}$$

Constant `AMBIENT` is defined as the lowest way index assigned for ambient computations. We define the operator of invalidation $C \wr c$ as $C[c \mapsto \langle \mathsf{IC};t;D\rangle]$ if $C(c) = \langle \mathsf{VC};t;D\rangle$, as $C[c \mapsto \langle \mathsf{ID};t;D\rangle]$ if $C(c) = \langle \mathsf{VD};t;D\rangle$, and C otherwise. We further define $\mathit{flush}(C,s,n)$ as flushing all n sets of non-ambient cache, starting from set index s. It is defined as $C \wr \langle s;0\rangle \wr \langle s;1\rangle \ldots \wr \langle s;\mathtt{AMBIENT}-1\rangle \ldots \wr \langle s+n-1;0\rangle \wr \langle s+n-1;1\rangle \ldots \wr \langle s+n-1;\mathtt{AMBIENT}-1\rangle$.

Memory Memory $\mu \in \mathbb{BLOCKID} \to \mathbb{DOMAIN} \times \mathbb{DBLOCK}$ is block-based. Note that each memory block is associated with a domain ID ($\mathbb{DOMAIN}$). Structurally, a data block $D \in \mathbb{DBLOCK}$ maps the offset $\delta \in \mathbb{OFFSET}$ to the instruction $\iota \in \mathbb{INST}$ ("code segment"), or a piece of data $n \in \mathbb{NUM}$ ("data segment"). Sets $\mathbb{BLOCKID}$, $\mathbb{OFFSET}$, and $\mathbb{NUM}$ are subsets of $\mathbb{NAT}$.

Given a memory address $l \in \mathbb{ADDR}$, we define a bijective function $\alpha : \mathbb{ADDR} \rightleftharpoons \mathbb{BLOCKID} \times \mathbb{DOMAIN} \times \mathbb{OFFSET}$ to compute its block index, domain, and offset. Given a $b \in \mathbb{BLOCKID}$, we define a bijective function $\beta : \mathbb{BLOCKID} \rightleftharpoons \mathbb{SET} \times \mathbb{TAG}$ to compute its set index s and the tage value t. We use $\mu\{l\}$ for $D(\delta)$ where $\mu(b) = \langle \Delta;D\rangle$ and $\alpha(l) = \langle b;\Delta;\delta\rangle$. We use $\mu\{l \mapsto n\}$ for $\mu[b \mapsto D']$ and $D' = D[\delta \mapsto n]$ where $\alpha(l) = \langle b;\Delta;\delta\rangle$.

4.2 Domain-Based Access with Permissions

We first formalize our DOP design in the cache, followed by a rigorous definition of permission-based access.

Domain Remapping and DRT DRT plays a central role in domain-based cache partitioning. We define a DRT (denoted as V) as $\mathbb{DOMAIN} \to \mathbb{SET} \times \mathbb{NAT}$, where $s \in \mathbb{SET}$ is the remapped set index for a memory block that would have been cached to set 0, and $n \in \mathbb{NAT}$ is the size of the cache partition for the domain is allocated to. The two components have a one-on-one correspondence with the hardware DRT columns but are more friendly for formalization.

Example 4.1. *Assuming 16 sets, the DRT in the third subfigure of Fig. 7 can be defined as* $[\Delta_{\mathtt{D0}} \mapsto \langle 0;4\rangle, \Delta_{\mathtt{D1}} \mapsto \langle 8;8\rangle, \Delta_{\mathtt{D2}} \mapsto \langle 4;4\rangle]$.

Two key functions are defined over the DRT:

- **[DRT Lookup]** Function $\mathit{remapped}(V,l)$ computes the set index (after remapping) and tag for a memory location l according to DRT V, defined as $\langle s_0 + (s \bmod n);t\rangle$ where $\alpha(l) = \langle b;\Delta;\delta\rangle$, $\beta(b) = \langle s;t\rangle$, $V(\Delta) = \langle s_0;n\rangle$.
- **[DRT Update]** Function $\mathit{updateDRT}(V,\Delta)$ computes the DRT after a new domain Δ requests partitions. The function computes to V if $\Delta \in \mathit{dom}(V)$. Otherwise, it computes to $V[\Delta \mapsto \langle s;n\rangle, \Delta' \mapsto \langle s+n;n\rangle]$ where $V(\Delta') = \langle s;n'\rangle$ and for any $\Delta'' \neq \Delta'$, $V(\Delta'') = \langle s'';n''\rangle$ and $n'' \leq n'$, and $n = n'/2$ and $n \geq 1$ and $n' \bmod 2 = 0$.

Sealing/Unsealing Metadata $F \in \mathbb{COMPART}$ is used for sealing (when it is not $\perp$) and unsealing (when it is $\perp$). When sealed, it keeps the ID of the compartment currently in execution. We define $\mathit{seal}(\langle F;V;C;R\rangle,\chi)$ as $\langle \chi;V;C;R\rangle$ iff $F = \perp$. The function is undefined otherwise. We define $\mathit{unseal}(\langle F;V;C;R\rangle)$ as $\langle \perp;V;C;R\rangle$.

Cache Access with Permissions At the core of domain-based access is how a location is accessed through the memory hierarchy, as regulated by permissions. To use a *location access descriptor* τ to capture the intention of access, as a tuple $\langle l;a;\Phi\rangle$: l is the memory address to access, a is the mode of access, and Φ is the permissions. The access mode can be either read (R) or write (W).

Location access through the memory hierarchy is formally defined in Fig. 10, through the $\psi \Diamond \tau$ operator, which says a

$$\frac{\tau = \langle l;a;\Phi\rangle \quad \Delta = domain(l) \quad \psi = \langle F;V;C;R\rangle \quad permissible(\Delta,\Phi,F) \quad remapped(V,l) = \langle s;t\rangle \quad c = \langle w;s\rangle \quad C(c) = \langle vb;t;D\rangle \quad vb = \mathsf{VC} \text{ or } \mathsf{VD} \quad T = update(R(s),w,\Delta)}{\psi\Diamond\tau \triangleq (c,\delta,vb,D,\psi\{s\mapsto T\})}$$

$$\frac{\tau = \langle l;a;\Phi\rangle \quad \Delta = domain(l) \quad \psi = \langle F;V;C;R\rangle \quad permissible(\Delta,\Phi,F) \quad remapped(V,l) = \langle s;t\rangle \quad c = \langle w;s\rangle \quad C(c) = \langle vb';t;D\rangle \quad vb' = \mathsf{IC} \text{ or } \mathsf{ID} \quad w' = replace(R(s),\Delta) \quad c = \langle w';s\rangle \quad C(c) = \langle vb;t'';D''\rangle \text{ for some } t'' \text{ and } D'' \quad T = update(R(s),w',\Delta)}{\psi\Diamond\tau \triangleq (c,\delta,vb,\emptyset,\psi\{s\mapsto T\})}$$

$$\frac{\tau = \langle l;a;\Phi\rangle \quad \Delta = domain(l) \quad \psi = \langle F;V;C;R\rangle \quad permissible(\Delta,\Phi,F) \quad remapped(V,l) = \langle s;t\rangle \quad \forall w \in [0..\texttt{AMBIENT}).t \neq t' \text{ where } C(\langle w;s\rangle) = \langle vb';t';D'\rangle \quad w' = replace(R(s),\Delta) \quad c = \langle w';s\rangle \quad C(c) = \langle vb;t'';D''\rangle \text{ for some } t'' \text{ and } D'' \quad T = update(R(s),w',\Delta)}{\psi\Diamond\tau \triangleq (c,\delta,vb,\emptyset,\psi\{s\mapsto T\})}$$

Figure 10: Permission-based Location-Cache Lookup

cache level ψ in the memory hierarchy is accessed for location access descriptor τ. It computes a 5-tuple (c,δ,vb,D,ψ'), where c is the accessed cache way/set (within the level), δ is the offset being accessed within the cache data block, *vb* is the validity bit after access, D is the data block after the access, and ψ' is the cache level itself after access. This 5-tuple result is consistent with our understanding that when a cache level is accessed, a particular way/set/offset is where the access indeed happens, and the access may alter the validity bit (*vb*), the data block itself (D), and metadata related to replacement logic (ψ'). The first subcase of Fig. 10 specifies the cache behavior when the accessed way/set has valid data (with validity bit being VC or VD); the second subcase is defined when the accessed way/set has invalid data (with validity bit being IC or ID); the third subcase is defined for the scenario where the data is no longer available (tag does not match). Functions *replace* and *update* are related to replacement logic, which we defer to the appendix. Note that in all 3 cases, the *remapped* function we defined earlier for DRT lookup plays an important role in cache lookup.

To see domain-based access at work, note that all Fig. 10 cases check permissions before granting access. This is supported: (1) function $domain(l)$ is defined as Δ where $\alpha(l) = \langle b;\Delta;\delta\rangle$. (2) function $permissible(\Delta,\Phi,F)$ is defined as $\Delta \in \Phi(F)$ if $F \neq \bot$, or $\Delta = \bot$ if $F = \bot$. The latter showcases the essence of domain-based access, which we now illustrate through examples.

Example 4.2 (In-Compartment Access to Domains with Permissions)**.** *Imagine the execution reaches a state where a compartment χ_3 is active whose permission Φ is $\chi_3 \mapsto \{\Delta_2,\Delta_5\}$. Predicate* permissible$(\Delta_2,\Phi,\chi_3)$ *is true, i.e., access to Δ_2 is granted. Predicate* permissible(Δ_4,Φ,χ_3) *is false, i.e., access to Δ_4 is not granted.*

Example 4.3 (Ambient Access)**.** *Imagine another execution currently outside compartments (running ambient code). Predicate* permissible$(\Delta,\Phi,\bot)$ *holds iff $\Delta = \bot$, aligned with our intuition that only the public domain ($\bot$) is accessible.*

cache behavior (definition in Appendix)	$\kappa,\mu \xrightarrow[T]{\lambda,v,\tau} \kappa',\mu'$	cache κ and memory μ transitions to κ' and μ' respectively with trace T, when cache unit λ subjects to access defined by access descriptor τ. v computes the read/write data
intra-threaded behavior (definition in Fig. 11)	$\kappa,\mu,\rho,\Phi,\iota \xrightarrow[q,n]{T} \kappa',\mu',\rho',\Phi'$	cache κ, memory μ, registers ρ, permissions Φ transitions to κ', μ', ρ', Φ' respectively while executing instruction ι at CPU core q, producing single-threaded trace T and program counter offset n
multi-threaded behavior (definition in Fig. 12)	$\Sigma \overset{\Omega}{\Rightarrow} \Sigma'$	runtime state Σ reduces to Σ' with multi-threaded trace Ω

Table 2: Relations

4.3 SCC Operational Semantics

Table 2 summarizes the key relations used to define the operational semantics. The relation in the first row defines the access behavior of the multi-level memory hierarchy, i.e., the access propagates through different cache levels. This definition is standard, except that for SCC, all access is regulated through domains and permissions through the use of the $\Diamond$ operator (§ 4.2). The relations in the second/third rows define the intra-thread and multi-threaded behavior respectively, which we detail next.

Runtime State The runtime state (Σ) consists of the cache hierarchy (κ), the memory (μ), the register file (ρ), and the thread store (π). The register file maps the register names $r \in \mathbb{REG}$ to the values n. The program store $\pi : \mathbb{TID} \to (\mathbb{ADDR} \times \mathbb{CORE} \times \mathbb{PERM})$ maps thread IDs (p) to program counters, the residing CPU cores (q), and the permissions.

Traces We define a trace T as a sequence of trace elements *tr*. Each trace element can be a data flow event *fe*, a branching event *be*, a domain allocation event *de*, or an observable event (or shorthanded as observation) *oe*. Given a data store ID $ds \in \mathbb{ADDR} \cup \mathbb{REG}$, a data flow event $ds_1 <: ds_2$ denotes a piece of data flows from (memory or register) store ds_1 to ds_2. A branching event, in the form of $?r$, denotes the value of a register r used as the condition for branching. A domain allocation event, in the form of $\circ\Delta$, denotes that a domain Δ is allocated in the cache. All these three forms of events are only used for meta-theory development.

Finally, we define an observation *oe* as a tuple $\langle ot;c;\lambda\rangle$. It says c residing in cache unit λ is accessed with observation type $ot \in \{\mathsf{R},\mathsf{W},\mathsf{M}\}$. The identifiers correspond to a hit-read, a hit-write, and a miss, respectively. To capture multi-threading, we use the notation $tr@p$ to denote a multi-thread trace element, where p is the thread where the trace element *tr* is incurred. A multi-threaded trace Ω is a sequence of multi-threaded trace elements. To compute a single-threaded trace (of thread p) from the multi-threaded one Ω, we define convenience function $\Omega|_p$ as the longest sequence $\overline{tr}^m$ where $[tr_1@p,\ldots,tr_m@p] \sqsubseteq \Omega$. To compute a multi-threaded trace from the single-threaded one T of thread p, we define $T@p$ as $[tr_1@p,\ldots,tr_n@p]$ where $T = [tr_1,\ldots tr_n]$ for some $n \geq 0$.

Reduction Rules Instruction-specific behavior is defined

$$\frac{\kappa = \overline{\lambda \mapsto \psi}^m \qquad \rho(r) = \chi \qquad \rho(\bar{r}) = \overline{\Delta}^n \qquad \Uparrow_{\Delta_n} \ldots \Uparrow_{\Delta_1} \psi_i = \psi_i' \text{ for } i = 1..m \qquad \kappa' = \overline{\lambda \mapsto seal(\psi', \chi)}^m}{\kappa, \mu, \rho, \Phi, \texttt{CBEGIN}\ r\ \bar{r} \xrightarrow[q,1]{\circ\overline{\Delta}} \kappa', \mu, \rho, \Phi[\chi \mapsto \overline{\Delta}^n]}\ [\textsc{CBegin}]$$

$$\frac{\rho(r) = \chi \qquad \rho(r') = ct \qquad \rho(\bar{r}) = \overline{\Delta}^n \qquad \eta(\langle ct; \Delta_i \rangle) = \Delta_i' \text{ for } i = 1..n \qquad \kappa = \overline{\lambda \mapsto \psi}^m \qquad \Uparrow_{\Delta_n'} \ldots \Uparrow_{\Delta_1'} \psi_i = \psi_i' \text{ for } i = 1..m \qquad \kappa' = \overline{\lambda \mapsto seal(\psi', \chi)}^m}{\kappa, \mu, \rho, \Phi, \texttt{HCBEGIN}\ r\ r'\ \bar{r} \xrightarrow[q,1]{\circ\overline{\Delta}} \kappa', \mu, \rho, \Phi[\chi \mapsto \overline{\Delta}^n]}\ [\textsc{HCBegin}]$$

$$\frac{\rho(r) = \chi \qquad \kappa = \overline{\lambda \mapsto \psi}^m \qquad \kappa' = \overline{\lambda \mapsto unseal(\psi)}^m}{\kappa, \mu, \rho, \Phi, \texttt{CEND}\ r \xrightarrow[q,1]{\emptyset} \kappa', \mu, \rho, \Phi \backslash \chi}\ [\textsc{CEnd}]$$

$$\frac{\tau = \langle l; \mathsf{R}; \Phi \rangle \qquad \kappa, \mu \xrightarrow[T]{H(q), v, \tau} \kappa', \mu'}{\kappa, \mu, \rho, \Phi, \texttt{LOAD}\ l\ r \xrightarrow[q,1]{T \cup \{l <: r\}} \kappa', \mu', \rho[r \mapsto v], \Phi}\ [\textsc{Load}] \qquad \frac{\tau = \langle l; \mathsf{W}; \Phi \rangle \qquad \kappa, \mu \xrightarrow[T]{H(q), \rho(r), \tau} \kappa', \mu'}{\kappa, \mu, \rho, \Phi, \texttt{STORE}\ r\ l \xrightarrow[q,1]{T \cup \{r <: l\}} \kappa', \mu', \rho, \Phi}\ [\textsc{Store}]$$

$$\frac{\rho(r) \neq 0}{\kappa, \mu, \rho, \Phi, \texttt{BR}\ r\ r' \xrightarrow[q, \rho(r')]{\{?r\}} \kappa, \mu, \rho, \Phi}\ [\textsc{BrTrue}] \qquad \frac{\rho(r) = 0}{\kappa, \mu, \rho, \Phi, \texttt{BR}\ r\ r' \xrightarrow[q,1]{\{?r\}} \kappa, \mu, \rho, \Phi}\ [\textsc{BrFalse}]$$

$$\frac{\kappa, \mu \rightarrow_{\texttt{wb}} \kappa', \mu'}{\kappa, \mu, \rho, \Phi, \texttt{NOP} \xrightarrow[q,0]{\emptyset} \kappa', \mu', \rho, \Phi}\ [\textsc{WB}] \qquad \frac{}{\kappa, \mu, \rho, \Phi, \texttt{NOP} \xrightarrow[q,0]{\emptyset} \kappa, \mu, \rho, \Phi}\ [\textsc{NOP}]$$

Figure 11: Intra-Thread Reduction Rules

$$\frac{\kappa, \mu, \rho, \Phi, \mu(l) \xrightarrow[q,n]{T} \kappa', \mu', \rho', \Phi' \qquad \mu(l+n) = \iota \text{ for some } \iota}{\kappa, \mu, \rho, \pi[p \mapsto \langle l; q; \Phi \rangle] \xRightarrow{T@p} \kappa', \mu', \rho', \pi[p \mapsto \langle l+n, q; \Phi' \rangle]}\ [\textsc{Multi}]$$

$$\frac{q \neq q'}{\kappa, \mu, \rho, \pi[p \mapsto \langle l; q; \Phi \rangle] \xRightarrow{\emptyset} \kappa, \mu, \rho, \pi[p \mapsto \langle l; q'; \Phi \rangle]]}\ [\textsc{ContextSwitch}]$$

Figure 12: Multi-Threaded Reduction Rules

$$\frac{V' = updateDRT(V, \Delta) \qquad C' = flush(C, V'(\Delta)) \text{ if } V' \neq V}{\Uparrow_\Delta \langle \bot; V; C; R \rangle \triangleq \langle \bot; V'; C'; R \rangle}$$

Figure 13: Allocation in a Single-Level Cache

in Fig. 11. Our formal system defines the behavior of the compartment access ([CBEGIN] for its entry and [CEND] for its exit), memory/cache access ([LOAD] and [STORE]), and control flow ([BR]). Unconditional jump can be encoded through [BR]. [HCBEGIN] is a variant of compartment entry, for horizontal compartments. For convenience, we further include a "no-op" instruction [NOP], purely to facilitate (modular) formal development: with it, the moment of write-back does not have to be tethered to another instruction (such as [LOAD] or [STORE]), but as a separate rule [WB], i.e., whenever [NOP] is allowed and a dirty cache data block is available. The behavior of write-back is specified through the $\rightarrow_{\texttt{wb}}$ relation, whose predictable definition is deferred to the appendix.

The most interesting instructions are related to compartment access. In [CBEGIN], note that the operator for domain-based cache allocation is defined in Fig. 13. It updates the DRT if the domain is not already allocated, and flushes the newly allocated sets. The function is only defined when the cache is unsealed. Entering a horizontal compartment, as shown in [HCBEGIN], is similar to [CBEGIN], except that its second argument now carries the context $\texttt{ct} \in \mathbb{CONTEXT}$, as a generalization of the caller address. Operationally, the key difference here is that domain identifiers are "refreshed" through a function $\eta : \mathbb{CONTEXT} \times \mathbb{DOMAIN} \rightarrow \mathbb{DOMAIN}$. The definition of η does not matter so long as it is injective. The [LOAD] and [STORE] instructions rely on the cache behavior relation we described in Table. 2. The most relevant fact to the discussion here is that location access is regulated with domains and permissions as described in § 4.2.

The multi-threaded behavior is defined over the intra-threaded semantics, with details in Fig. 12, through relation $\Rightarrow$. We further use $\Rightarrow^*$ to represent the reflexive and transitive closure of $\Rightarrow$. Rigorously: (1) If $\Sigma \xRightarrow{\Omega} \Sigma'$, then $\Sigma \xRightarrow{\Omega}{}^* \Sigma'$. (2) If $\Sigma \xRightarrow{\Omega} \Sigma'$ and $\Sigma' \xRightarrow{\Omega'}{}^* \Sigma''$, then $\Sigma \xRightarrow{\Omega, \Omega'}{}^* \Sigma''$. Here, rule [MULTI] says the (global) configuration takes a step of reduction if any of the threads takes a step. Observe that the program counter is updated. [CONTEXTSWITCH] is a simple rule to capture a realistic behavior in systems: the scheduler may migrate a thread from one core to another.

4.4 Metatheory

Definition 4.1 (Trace Realizability)**.** *Predicate* $\Sigma \vdash_p \mathrm{T}$ *says that* T *is a possible per-thread trace with initial state* Σ*. It holds iff there exists* Σ' *such that* $\Sigma \xRightarrow{\Omega}{}^* \Sigma'$ *and* $\mathrm{T} = \Omega|_p$*.*

Definition 4.2 (Accessibility)**.** *We say thread p can access the domain* Δ *with initial state* Σ *iff there exists some* T *such that* $\Sigma \vdash_p \mathrm{T}$ *and* $(\circ\Delta) \in \mathrm{T}$*.*

Next, let us introduce the notion of a *guardian*. The intuition behind it is that the value such a memory location holds may flow to the condition of a branching instruction ($?r$ as trace element), which subsequently guards whether domains are dynamically allocated ($\circ\Delta$ trace element). Here, observe that we consider the general case where the data flow and branching may span multiple threads ($p_0, \ldots, p_{n+2}$):

Definition 4.3 (Guardian)**.** *We say location l is a guardian (for some domain allocation) with initial state* Σ *iff there exist* Σ', Ω *s.t.* $\Sigma \xRightarrow{\Omega}{}^* \Sigma'$ *and* $[(l <: \mathrm{fe}_1)@p_0, (\mathrm{fe}_1 <: \mathrm{fe}_2)@p_1, \ldots, (\mathrm{fe}_n <: r)@p_n, ?r@p_{n+1}, (\circ\Delta)@p_{n+2}] \sqsubseteq \Omega$ *for some* $\mathrm{fe}_1, \ldots, \mathrm{fe}_n$*,* $p_0, \ldots, p_{n+2}$*, r,* Δ*, and* $n \geq 0$*.*

For the following theorem, we define convenience operator $\Sigma[l \mapsto n]$ for memory update in the runtime state as $\langle \kappa; \mu\{l \mapsto n\}; \rho; \pi \rangle$ where $\Sigma = \langle \kappa; \mu; \rho; \pi \rangle$.

Theorem 4.1 (Immunity to Side-Channel Attacks)**.** *Given* $\Sigma = \langle \kappa; \mu; \rho; \pi \rangle$ *and some* l *where* $\alpha(l) = \langle b; \Delta; \delta \rangle$*, and* p *that cannot access* Δ *with initial state* Σ*, then the two are equivalent for any* $n_1 \neq n_2$ *if* l *is not a guardian: (1)* $\Sigma[l \mapsto n_1] \vdash_p \mathrm{T}$*; (2)* $\Sigma[l \mapsto n_2] \vdash_p \mathrm{T}$.

The property here is defined through trace indistinguishability: for any "secret" data in location l that the (attacker) thread p does not have permission to access, the attacker cannot rely on the difference in traces — including the observations of cache hit and cache miss, or read or write — to infer the secret value. Note that this result is general in that it considers the possibility of *non-determinism*: it subsumes that from one initial state, more than one trace is possible, a norm in multi-threaded shared-memory programming. Furthermore, it *does not require terminism* in thread execution: T can be in infinite length.

5 SCC Evaluation: Methodology and Results

We implemented SCC in gem5 cycle-accurate architectural simulator [8]. We manually compartmentalized a number of benchmarks: four from the SPEC2017 suite [15] (all the ones written in C, except for gcc, which has over 1 million lines of code [79]), eight from MiBench [36] suite and CoreMark [27] benchmark. The MiBench and CoreMark benchmarks were run to completion. We fast-forwarded SPEC 2017 benchmarks for 100 billion instructions and then performed cycle-accurate simulation for the next one billion instructions. Caches were also warmed up during the fast-forwarding phase.

Figure 14 illustrates our experimental methodology, which involves four steps. We used the LLVM-based [52] SOAAP framework [34] to annotate security-critical variables in these applications (`Step-1`). For each variable, SOAAP generates a list of functions that use it (`Step-2`), and these functions are then encapsulated in a compartment using gem5 pseudo-instructions (`Step-3`). Specifically, we added three pseudo-instructions to gem5: `start_compart`, `end_compart`, and `horiz_compart`. Start/end instructions take a compartment ID value, and horizontal compartmentalization instructions take a virtual address as their arguments. The virtual address is translated at runtime before the corresponding page is marked as a common callee by the hardware. Appendix describes the rationale and specifications (in Table 1) for compartmentalizing our target benchmarks in detail.

Our methodology involves two separate runs for each benchmark: the first run generates the permission table, while the second run executes a compartmentalized application managed by the extracted permission table (`Step-4`). Permission table generation is accomplished by tracing memory pages

Hardware Parameters	
Core	2GHz x86 ISA (extended with compartmentalization instructions) Out-of-Order core modeled after Skylake μarch
Cache Hierarchy	Snooping-based MOESI coherence policy, inclusive write-back caches
L1I/D Caches	32KB total size, 8-ways, 2-cycle access and response latency (4-cycle roundtrip)
L2 Cache	1MB total size 16-ways, 12-cycle access and response latency (24-cycle roundtrip)
DRAM	32GB size, 4GB channel capacity, DDR4-2400 x64 channel, 4 devices per rank, 1 rank per channel, 1GB per device (50ns average roundtrip latency)

Table 3: Simulation parameters.

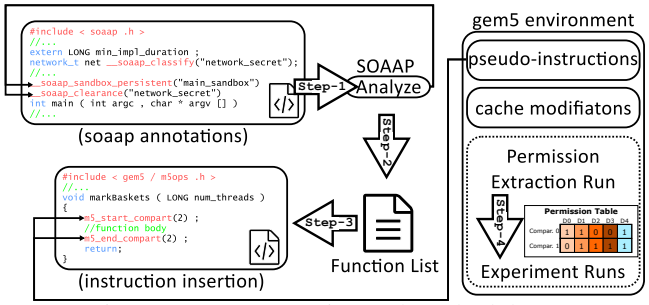


Figure 14: SCC Experimental Methodology.

accessed by each compartment. Pages sharing the same permission composition are then grouped into the same domain in the permission table, which is used by the second run. The configuration of our simulated processor is shown in Table 3. We simulate a single core with its microarchitectural configuration modeled after Skylake SP server [25].

We compare system and cache performance metrics for six cache management schemes: 1) *Baseline* – an insecure monolithic program without compartmentalization and SCC, 2) *Proportional* – an allocation scheme where the size of each partition is proportional to its domain's page count (this is not implementable in a real system and is calculated ahead of time for this experiment), 3) *SCC*, 4) *Static-16* – 16 statically allocated uniformly sized partitions that are created regardless of the number of domains used by the program, 5) *Static-32* – same as *Static-16*, but with 32 partitions allocated, and 6) *FoCS* – Flush on Compartment Switch, where the entire cache hierarchy is flushed upon compartment switches and system calls.

5.1 Security Evaluation

We evaluated SCC against two side-channels: the occupancy channel [77], and the library loading channel [86]. We implemented proof-of-concept (PoC) attacks involving two compartments - the benign compartment and the attacker compartment.

Cache Occupancy Attacks: In this experiment, the victim program is either accessing a page-size buffer and loads it into the cache, or it does not, depending on the input. Following the victim compartment's execution, the attacker compartment accesses the entire L1 cache and measures the total access time. Figure 15 shows the results of this experiment. The

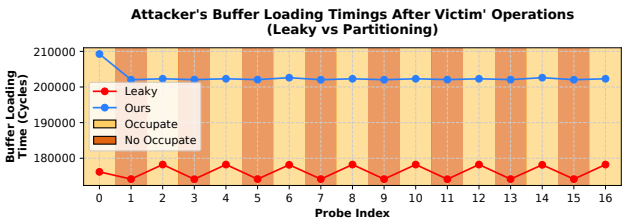


Figure 15: PoC cache occupancy attack and its mitigation

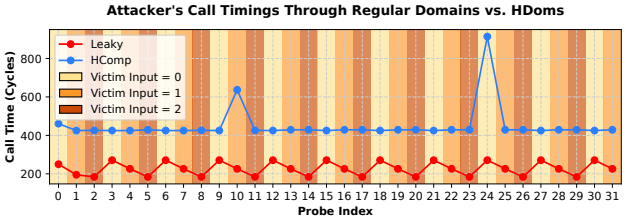


Figure 16: PoC attack using function calls and its mitigation

background colors represent the execution periods where the victim either accesses the cache (yellow) or not (orange). The red plot represents the access timings of the insecure baseline. A noticeable correlation exists between the timings recorded by the attacker and the dataflow of the victim. Specifically, when the victim loads the buffer, the attacker experiences a visibly longer delay in reloading its buffer. SCC eliminates this leakage by ensuring that the attacker's timing measurements are independent of the victim's operation.

Attacks Using Shared Function Calls: To evaluate the efficacy of SCC against attacks using calls to shared library functions, we performed an experiment to demonstrate the attack and its mitigation with HComps. In our PoC attack, the attacker first flushed the common function from the cache, waits for the victim compartment to call the function, and then calls it again and times the execution. Depending on the victim's secret input, the call by the attacker takes a different amount of time to complete due to the correlation between the number of instructions fetched into the cache by the attacker and the input.

The results are shown in Figure 16, where each data point represents the time it takes for the attacker to call the common function. The background colors represent different inputs of the victim compartment to emphasize the secret-dependent access times in the leaky scenario. Note that in the leaky scenario, regular SCC protections are still in place, but without HComps. Indeed, even when SCC without HComps is used, the attacker can reliably extract the victim's access pattern through a partitioned instruction cache.

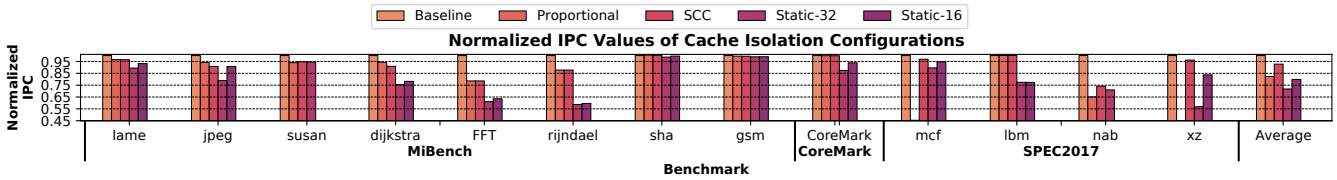
In contrast, HComp ensures that accesses from two callers are served from separate cache partitions. Aside from the occasional partitioning-induced cache misses, no pattern is observable from the attacker side. The HComp accesses take almost twice the time compared to the leaky scenario. This is caused by the attacker flushing only its instance of the function, therefore missing every cache request for each call.

5.2 Performance Evaluation

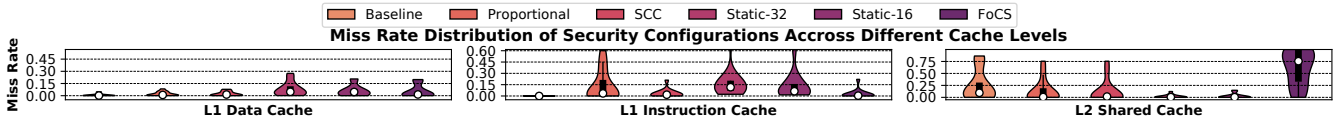
Figure 18 shows the IPC (Instructions per Cycle) values for *SCC*, *FoCS* and a scheme we only flush the L1 caches on a compartment switch and deploy SCC on other cache levels (referred to as *FL1oCS*). Next to each benchmark name, we show (in parenthesis) the number of compartment switches per million instructions. The performance results are normalized to the insecure *Baseline*. On average, *FoCS* and *FL1oCS* result in 60% and 32% performance degradation, while *SCC* degrades performance by 6%. The primary factor of this overhead for *FoCS* is frequent and prolonged invalidation and writeback procedures. Since both instruction and data L1 caches are blocked from any access until all writeback requests are completed, the entire pipeline is stalled until the execution switches to the next compartment. The same argument applies to *FL1oCS*, but to a lesser extent because the content of lower-level caches is retained and writebacks to the main memory are not initiated. While for some benchmarks that feature low frequency of compartment switches (*susan*, *FFT*, and *gsm*), *FL1oCS* performs on par or even better than SCC, on the average SCC still outperforms *FL1oCS* by 26%.

Figure 17 compares performance and cache miss rates for all schemes, excluding *FoCS*, which was discussed above. Figure 17(a) shows the normalized IPC values for compartmentalized benchmarks, while Figure 17(b) shows cache miss rate distribution for all configurations and cache levels. We omit *susan* from *Static-16* experiments, as it has 17 domains and can only be statically supported by *Static-32* scheme. *Proportional* shows the most significant performance degradation for *FFT*, *mcf*, *nab*, and *xz* at 21%, 56%, 37%, and 86%. This pattern is caused by the non-uniformity of domain sizes for these benchmarks. For instance, *mcf* generates a domain with a size of approximately 49KB. Although the domain size is larger than the L1 cache, *Proportional* allocates only 2 sets to this domain in the cache due to the high memory demand of other domains. Under the *Baseline* configuration, when this domain is accessed, it utilizes the entire L1 data cache. However, with *Proportional*, the domain can occupy only 768 bytes (2 sets and 6 ways allocated for partitions) of the L1 cache at any given time, increasing the cache pressure for the given domain. This is depicted in Figure 17(b) by elevated L1 cache miss rates. Note that *Proportional* design can also result in cache underutilization which is also demonstrated in Appendix.

SCC results in performance improvement. *rijndael*, *FFT*, and *nab* exhibit slowdowns of 12%, 22%, and 26% respectively; all other benchmarks experience slowdowns of less than 10%. The slowdown in *rijndael* and *nab* is primarily due to smaller partitions for domains, while *mcf* experiences a higher ADR miss rate for the L1 instruction cache (10% while others are 1% in Figure 5 in Section 3.1). Note that while longer access times in L1 data caches can typically be masked by out-of-order execution, ADR misses in L1 instruc-



(a) Performance impact of different cache management schemes on compartmentalized benchmarks in terms of normalized IPC. Note that the minimum value represented on the graph is 0.45.



(b) Violin graphs of cache miss rates across different cache levels under security configurations. Each distribution includes every benchmark we evaluate. White dots indicate the median value of each distribution. Black stripes denote the data between the first and third quarterlies.

Figure 17: Performance and miss rate evaluation results.

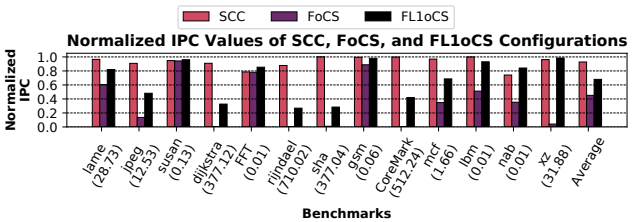


Figure 18: Performance impact of *SCC*, *FoCS* and *FL1oCS*. Values shown here are normalized by the baseline IPC. we show the number of compartment switches per million instructions in parentheses after the names of benchmarks.

tion cache lead to fetch stalls. This demonstrates that SCC's uniform partitioning approach yields more favorable results than *Proportional*, especially when domain sizes exhibit significant diversity within a single address space. Alternatively, allocating partitions statically for the maximum number of domains not only impedes scalability but also increases cache pressure. Specifically, benchmarks such as *dijkstra*, *FFT*, *rijndael*, *CoreMark*, and *nab* encounter significant miss rates in the L1 data cache, reaching up to 20%. The same effect is also present in the L1 instruction cache for most benchmarks. This has a detrimental impact on the performance of programs with *Static* schemes - up to 50% performance loss was observed.

Results summary and takeaway: On average, *Proportional*, *SCC*, *Static-16*, *Static-32*, and *FoCS* result in 18%, 7%, 28%, 20%, and 28% performance degradation respectively compared to the insecure baseline. In environments where the number of domains is limited to a modest number (such as in page-based or range-based SC schemes), SCC provides security from cache attacks with a modest performance impact of only 7%, which is comparable to the slowdown of the *Proportional* scheme. These results also demonstrate the infeasibility of flushing the cache hierarchy on compartment switches. Employing *FoCS* incurs high L2 cache miss rates of up to 99% on several benchmarks.

Component	Area (mm^2)	Peak Dynamic (W)	RT Dynamic (W)
Baseline Core	16.242 (100%)	36.615 (100%)	36.615 (100%)
ADR	<0.001 (<0.001%)	<0.001 (<0.001%)	<0.001 (+0.002%)
DRT (L1i/d)	<0.001 (+0.002%)	0.003 (+0.009%)	0.009 (+0.027%)
DRT (L2)	0.001 (+0.007%)	0.008 (+0.022%)	0.026 (+0.070%)
DRTs (Total)	0.002 (+0.011%)	0.015 (+0.039%)	0.046 (+0.124%)
Extra Tag Bits	0.101 (+0.62%)	0.212 (+0.29%)	0.113 (+0.33%)
Total	0.103 (+0.63%)	0.224 (+0.61%)	0.155 (+0.42%)

Table 4: Area/Power estimates for SCC components.

5.3 Timing, Area, and Power Analysis

To estimate the timing, area, and power impact of SCC, we used McPAT (and its submodule CACTI [93]) tool, assuming cache parameters in Table 3 under 22 nm technology.

Timing Impact: To hide the delay of two extra gates used by SCC, we leverage the cache banking logic. We assume 8 banks for L1 caches, as implemented in Intel Sandy Bridge and Ivy Bridge processors [43]. In this case, 64-byte cache lines are divided into eight 8-byte lines, one per bank [69]. Each bank contains 4 subarrays mapped by the least significant bits of the set index [99]. Most significant bits of the byte offset are used to determine the bank to assure interleaving [17].

The integration of the SCC logic with cache banking and subarray layout is shown in Figure 19. There are two possible additions to the critical path: the bank decoder and the global subarray decoder. SCC's masking and partition ID concatenation operations are hidden behind these decoders respectively. A 3×8 decoder's critical path is composed of two inverters and a 3-input NAND gate, which accounts for 0.36 ns of delay [99]. The NAND gate and the inverter used in the masking operation have a smaller critical path than the banking decoder – 0.24 ns. Similarly, the global decoder within the cache bank has a delay of 0.32 ns. The partition ID concatenation process (a NOR gate and an inverter) can also be hidden behind this delay – 0.26 ns. Lower-level caches accommodate even larger decoders, while the remapping logic remains unchanged; no additional delays are introduced for these caches as well.

Area and Power Impact: We modeled DRT and ADR structures for each private cache. We assume that the system

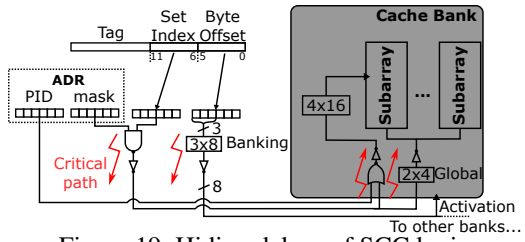


Figure 19: Hiding delays of SCC logic

supports up to 16 domains, with 16 13-bit and 18-bit entries allocated for DRTs in L1 and L2 caches, respectively. We also estimated the area overhead from the extended tags required for set indexing logic, with tags extended by 4 bits (for 16 domains). As shown in Table 4, the total area, peak power, and runtime dynamic power of a core are increased by 0.63%, 0.38%, and 0.42% respectively.

6 Related Work

Compartmentalized Software: Different SC designs propose domains at different levels of granularity, including objects [63, 90], bytes [16], pages [75], and Virtual Memory Areas (VMA) [7]. In hardware-supported compartmentalization, access privileges of compartments are maintained using either: capabilities (object or range-based) [63,90] or Memory Management Unit (MMU) [7,16,65,75,104]. Hybrid schemes were also proposed [85]. There is a trade-off associated with domain granularity: object-level granularity necessitates modifications across multiple layers of abstraction, including the Instruction Set Architecture (ISA), compilers, OS, and hardware. In contrast, MMU-based schemes are coarser-grained, typically with page and VMA-level granularity, but provide higher performance and simpler implementations. Enclave-based compartmentalization [1,2,19,65,98] is also a variant of SC, with memory permissions controlled at enclave granularity. SCC design can be applied to secure cache hierarchies for any of these schemes.

Cache Side-Channels and Defenses: Cache-based side-channel attacks [13, 14, 18, 20, 29, 31–33, 35, 38, 45, 49, 55, 56, 58, 61, 62, 64, 77, 78, 83, 86, 92, 94–96, 100–102] have been a subject of active research in recent years. Many forms of attacks have been demonstrated, leaking secrets at various granularities from traditional systems and secure enclaves. The key idea of these attacks is to observe the behavior of the victim in terms of its access to the shared caches and correlate the victim's cache usage (or even the overall performance impact on the attacker) with secret information used to infer a victim's data-dependent access pattern.

In terms of defenses, partitioning [3,5,11,21,24,48,57,68,73,80,81,87,88] and randomization [9,10,23,67,68,74,89,91] emerged as two main approaches. SCC is a cache partitioning solution tailored to SC systems. Previous cache partitioning schemes considered either process-based isolation [21, 24, 73] or enclave-based isolation with minimal interaction between enclaves [3, 5, 81]. As a result, previous designs resulted in reasonable partitioning schemes for shared last-level caches [24, 48, 81], but propose L1 flushing on context switches (including enclave switching) and system calls to avoid expensive partitioning of L1 caches [3, 11, 28, 73]. In an SC environment where fine-grain mutually mistrusting in-process compartments can frequently interact, L1 flushing would be needed frequently leading to substantial performance degradation. We compare our scheme to prior works in terms of different aspects in Table 5.

Mechanism	Partition	Code	Levels	Perf.	Area
BCE [73]	Set	Process	L3	-1.3%	+1.8%
CURE [5]	Way	Enclave	L3	-10%	+1.8%
CATalyst [41,57]	Way	Process	L3	-0.5%	0%
NoMo [24]	Set	Process	L1	-10%	?
ChunkedCache [22]	Set	Enclave	L3	?	+2.7%
MI6 [11]	Set	Enclave	L3	-16%	+2%
DAWG [48]	Way	Process	All	-5%	?
SecDCP [87]	Way	Process	L3	+10%	?
HybCache [21]	Way	Enclave	All	-5%	+0.07%
CC [81]	Set/Way	Enclave	L3	-5%	+0.84%
TEE-SHirT [3]	Set/Way	Enclave	All	-10%	+6.2%
SCC	Set/Way	Compartment	All	-7%	+0.63%

Table 5: SCC's comparison against other cache partitioning mechanisms. Columns indicate partitioning/code granularity, protected caches, performance, and hardware overhead.

Another major difference from prior works is that existing hardware-based cache partitioning schemes provide partitions per processes/enclaves where sharing between parties is minimal or non-existent. In the world of compartmentalized software, this partitioning philosophy would translate to partition-per-compartment design. With this approach, it would be very challenging to support intensive sharing of memory pages between compartments of a process, since compartments are typically tightly integrated pieces of code. Therefore, this paper advocates partition-per-memory-domain approach, and all cache partitioning solutions at all levels have to be reassessed under this framework.

Page coloring [6, 46, 54] is a software-based technique that can be used to mitigate collisions in caches and thus side-channels. [47, 76]. However, assigning colors to every domain of every process would require too many colors, making it impossible to align differently-colored pages to sets without collisions. Furthermore, even if the scalability was not a concern, the OS must have a detailed information about the cache hashing and slicing logic to properly map differently colored pages into different cache sets [57].

7 Concluding Remarks

We demonstrated that compartmentalized software is vulnerable to cross-compartment cache side-channel attacks in various forms. We proposed SCC - a framework to augment compartmentalized systems with leakage-free cache hierarchies in a performance-friendly manner. SCC uses a novel DOP approach, where cache partitions are created for memory domains, instead of individual compartments, allowing efficient and secure data sharing. SCC also extends cache partitioning

to L1 caches with a modest impact on their performance and complexity. Finally, SCC incorporates protection from attacks exploiting code routines used by multiple callers.

8 Acknowledgments

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.

9 Ethics Considerations

We confirm that our research complies with the ethics policies. This work does not lead to tangible harm and does not violate human rights. All of the code and contents of this paper were conceived by the authors of the paper and were not plagiarized from other works.

10 Open Science

We attest that our work complies with the open science policy. Our source code and compartmentalized MiBench benchmarks are public and available at: https://zenodo.org/records/14736160. The appendices referred to in the paper can be found in https://www.cs.binghamton.edu/~karikan1/SCC_USENIX25_Appendix.pdf.

References

[1] *Intel Software Guard Extensions SDK Developer Reference for Linux OS*, 2016.

[2] 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.

[3] Kerem Arıkan, Abraham Farrell, Willams Zhang Cen, Jack McMahon, Barry Williams, Yu David Liu, Nael Abu-Ghazaleh, and Dmitry Ponomarev. TEE-SHirT: Scalable Leakage-Free Cache Hierarchies for TEEs. San Diego, CA, USA, 2024. The Internet Society.

[4] ARM. *ARM Developer Suite Developer Guide*. 2001.

[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] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, page 158–170, New York, NY, USA, 1994. Association for Computing Machinery.

[7] A. Bhattacharyya, F. Hofhammer, Y. Li, S. Gupta, A. Sanchez, B. Falsafi, and M. Payer. SecureCells: A Secure Compartmentalized Architecture. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 2921–2939, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the 11th USENIX Conference on Offensive Technologies*, WOOT'17, page 11, USA, 2017. USENIX Association.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.

[14] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. RELOAD+ REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[15] 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.

[16] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, 2016.

[17] Sangyeun Cho. I-cache multi-banking and vertical interleaving. In *Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, GLSVLSI '07, page 14–19, New York, NY, USA, 2007. Association for Computing Machinery.

[18] Jack Cook, Jules Drean, Jonathan Behrens, and Mengjia Yan. There's Always a Bigger Fish: A Clarifying Analysis of a Machine-Learning-Assisted Side-Channel Attack. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 204–217, New York, NY, USA, 2022. Association for Computing Machinery.

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

[20] 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, volume=2018, url=https://tches.iacr.org/index.php/TCHES/article/view/879, doi=10.13154/tches.v2018.i2.171-191, number=2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, page 171–191, May 2018.

[21] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 451–468. USENIX Association, August 2020.

[22] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. ChunkedCache: On-Demand and Scalable Cache Isolation for Security Architectures. *The Network and Distributed Systems Security Symposium*, 2022.

[23] 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.

[24] 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.

[25] 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.

[26] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. XPC: Architectural Support for Secure and Efficient Cross Process Call. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 671–684, 2019.

[27] Shay Gal-On and Markus Levy. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

[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 *NDSS*, February 2017. Pwnie Award for Most Innovative Research, DCSR Paper Award.

[31] 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.

[32] 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.

[33] 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.

[34] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1016–1031, New York, NY, USA, 2015. Association for Computing Machinery.

[35] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–Bringing access-based cache attacks on AES to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 490–505. IEEE, 2011.

[36] 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.

[37] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 299–312, Santa Clara, CA, July 2017. USENIX Association.

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

[39] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, 2013.

[40] IBM Corporation. *Power ISA version 3.0b*. 2017.

[41] CAT Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *Intel Corporation, April*, 2015.

[42] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual*. 2019.

[43] Intel Corporation. Intel® 64 and ia-32 architectures optimization reference manual volume 1. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html, 2020. Accessed: 2024-12-26.

[44] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604, 2015.

[45] 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.

[46] Richard E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10:338–359, 1992.

[47] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-Level protection against Cache-Based side channel attacks in the cloud. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, August 2012. USENIX Association.

[48] 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.

[49] 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.

[50] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 437–452, New York, NY, USA, 2017. Association for Computing Machinery.

[51] Zili Kou, Wenjian He, Sharad Sinha, and Wei Zhang. Load-Step: A Precise TrustZone Execution Control Framework for Exploring New Side-channel Attacks

Like Flush+Evict. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 979–984, 2021.

[52] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[53] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. Lord of the X86 Rings: A Portable User Mode Privilege Separation Architecture on X86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1441–1454, New York, NY, USA, 2018. Association for Computing Machinery.

[54] J. Liedtke, H. Hartig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 213–224, 1997.

[55] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, August 2016. USENIX Association.

[56] 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.

[57] 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.

[58] 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.

[59] Shen Liu, Gang Tan, and Trent Jaeger. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2359–2371, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Marcela S Melara, Michael J Freedman, and Mic Bowman. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments. *arXiv preprint arXiv:1907.13245*, 2019.

[61] 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.

[62] 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.

[63] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. FRAMER: A Tagged-Pointer Capability System with Memory Safety Applications. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 612–626, New York, NY, USA, 2019. Association for Computing Machinery.

[64] 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.

[65] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789, 2020.

[66] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

[67] 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.

[68] 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.

[69] Paul Racunas and Yale N. Patt. Partitioned first-level cache design for clustered microarchitectures. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, page 22–31, New York, NY, USA, 2003. Association for Computing Machinery.

[70] Charles Reis and Steven D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 219–232, New York, NY, USA, 2009. Association for Computing Machinery.

[71] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 199–212, New York, NY, USA, 2009. Association for Computing Machinery.

[72] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P Kemerlis, Mathias Payer, Adam Bates, Jonathan M Smith, Andre DeHon, et al. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 296–311, 2021.

[73] 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.

[74] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1379–1396. USENIX Association, August 2021.

[75] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[76] 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.

[77] 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.

[78] 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.

[79] Standard Performance Evaluation Corporation. SPEC CPU 2017: Benchmarks Overview. https://www.spec.org/cpu2017/Docs/overview.html#benchmarks, 2017. Accessed: 2024-08-27.

[80] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. SecSMT: Securing SMT Processors against Contention-Based Covert Channels. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3165–3182, Boston, MA, August 2022. USENIX Association.

[81] 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.

[82] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.

[83] 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.

[84] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *NDSS*, 2018.

[85] Lluis Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting software with Code-centric memory Domains. pages 469–480, 06 2014.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[91] 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.

[92] 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.

[93] S.J.E. Wilton and N.P. Jouppi. Cacti: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, 1996.

[94] 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.

[95] 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.

[96] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7:99–112, 2017.

[97] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[98] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson, and Prateek Saxena. Elasticlave: An efficient memory model for enclaves. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4111–4128, 2022.

[99] Chuanjun Zhang. Balanced Cache: Reducing Conflict Misses of Direct-Mapped Caches. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 155–166, 2006.

[100] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. *Cryptology ePrint Archive*, 2016.

[101] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[102] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 305–316, New York, NY, USA, 2012.

[103] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Untangle: A Principled Framework to Design Low-Leakage, High-Performance Dynamic Partitioning Schemes. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 771–788, New York, NY, USA, 2023. Association for Computing Machinery.

[104] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-Based Fault Isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 558–569, New York, NY, USA, 2014. Association for Computing Machinery.