Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks

Daniel Townley
Peraton Labs

Kerem Arıkan
Binghamton University

Yu David Liu
Binghamton University

Dmitry Ponomarev
Binghamton University

Oğuz Ergin
TOBB University of Economics and Technology

Abstract

The security of isolated execution architectures such as Intel SGX has been significantly threatened by the recent emergence of side-channel attacks. Cache side-channel attacks allow adversaries to leak secrets stored inside isolated enclaves without having direct access to the enclave memory. In some cases, secrets can be leaked even without having the knowledge of the victim application code or having OS-level privileges. We propose the concept of Composable Cachelets (CC), a new scalable strategy to dynamically partition the last-level cache (LLC) for completely isolating enclaves from other applications and from each other. CC supports enclave isolation in caches with the capability to dynamically readjust the cache capacity as enclaves are created and destroyed. We present a cache-aware and enclave-aware operational semantics to help rigorously establish security properties of CC, and we experimentally demonstrate that CC thwarts side-channel attacks on caches with modest performance and complexity impact.

1 Introduction

Isolated execution architectures have received significant traction in the industry, with Intel Software Guard Extension (SGX) [1, 28, 41] being the most prominent example. Isolated execution relies on dedicated hardware to protect sensitive parts of application code and memory within secure *enclaves* that are inaccessible to all outside programs, including operating systems and hypervisors. Unlike classical memory protections enforced by system software, isolated execution relies exclusively on trusted hardware to enforce enclave boundaries, ensuring confidentiality and integrity of sensitive information even if system software is compromised. Despite these benefits, recent research demonstrated powerful cache side-channel attacks that bypass isolated execution [10, 17, 22, 35, 43, 55, 56, 69]. These attacks exploit the fact that enclaves, like other processes, share cache lines with potential attackers, who can observe timing differences caused by collisions of their own accesses with those of a victim. Moreover, the threat model under isolated execution greatly amplifies the power of these attacks, as an attacker can leverage a compromised OS to achieve greater control over side-channel measurements [10, 42, 43, 53, 57]. As trusted computing infrastructure relying on SGX and similar technologies continues to be rapidly developed and deployed [55], it is critical to consider system designs that make enclaves immune to cache side-channel attacks.

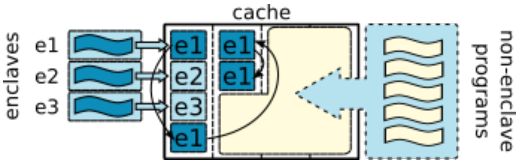

Figure 1: Overview of Partitioning in CC

Existing cache partitioning schemes [20, 29] successfully thwart some side-channel attacks, but face limitations that make them incompatible with SGX and other isolated execution proposals. Partitioned caches eliminate cache line collisions by assigning each process (or security domain) a disjoint subset of cache lines. Some of partitioning designs, however, rely on system software to enforce critical protections [32–34, 38, 52, 70, 74], and therefore do not work with isolated execution security models where the system software is not trusted. While some recent designs implement partitioning in hardware alone, they often require allocating one or more cache ways to each partition [20]. Enclaves, which typically protect small sections of critical data and instructions, will greatly underutilize these coarse-grained partitions, excessively degrading performance of non-enclave code. Moreover, the number of enclaves can rapidly exceed the number of ways available in a typical last level cache (LLC), making coarse-grained partitioning impractical, particularly in multi-tenant cloud settings where enclaves are most likely to be deployed. These limitations necessitate the development of new architectures to provide enclaves with efficient and scalable partitioning, without relying on system software to enforce isolation.

Another key insight that motivates our work is that there remains an "impedance mismatch" between software need and hardware support. Whereas the data protection need from the application is often fine-grained and dynamically evolves,

caches as the epicenter of side channel attacks are often *monolithic* in the eyes of attackers, and existing partitioning-based solutions are too *rigid* in breaking down this monolithic view. With security as a cross-layer concern, we believe the best solution is to streamline the hardware-software stack: can we endow caches with the fine granularity and dynamism that the application calls for?

To address these challenges, we propose Composable Cachelets (CC), a novel cache design that provides fine-grained, flexible cache partitioning without trusting system software to enforce isolation. CC partitions set-associative caches at the granularity of *cachelets*—fine-grained, hardware-defined cache regions that span continuous ranges of sets across one or a few cache ways (Figure 1). To support enclaves with varying memory demands, CC can either assign a single cachelet to an enclave (as is the case for enclaves 2 and 3 in Figure 1), or chain cachelets together into larger *virtual partitions* (such as the one allocated to enclave 1). Inspired by classic memory paging designs, CC can compose virtual partitions from arbitrary, non-sequential cachelets. This approach allows CC to efficiently use available cache lines as enclaves of various sizes are created and destroyed.

In addition to architectural design, we also provide a formal foundation to study program behavior in the presence of caches as side channels, and CC as the defense against side channel attacks. Through a novel cache-aware and enclave-aware operational semantics, we account for enclave program behavior in realistic scenarios where enclaves may be dynamically managed (created, entered, exited, and destroyed), cachelets may be dynamically managed (allocated, accessed, and deallocated), and the attacker may observe different types of cache events (e.g., hit, miss, resize). As a whole, this formal foundation serves as a rigorous proof of the security guarantees enjoyed by CC.

In summary, this paper makes the following contributions:

- We describe Composable Cachelets - a novel, fine-grained, and scalable cache design that efficiently protects application secrets from side-channel attacks (Section 3).
- We present a cache-aware and enclave-aware operational semantics to rigorously define the behavior of enclave programs, and establish CC's security properties (Section 4).
- We evaluate performance impact of CC on a variety of applications, including smaller cryptographic programs, as well as larger benchmarks such as SPEC and PARSEC. We also evaluate delay, area and power impact of our design. Our results demonstrate that strong security properties of CC can be achieved with modest overhead.

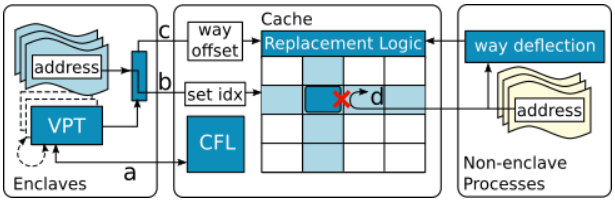

Figure 2: CC Overview

2 Threat Model

CC adopts the threat model assumed by Intel SGX. SGX assumes a powerful adversary that can engage the resources of a compromised OS and/or hypervisor to launch an attack against an enclave. Established SGX protections are assumed to be fully implemented in CC. These protections prevent attackers from directly accessing enclave memory and from extracting information through physical attacks on DRAM and interconnect. However, an attacker can attempt *Prime+Probe* against caches shared with a victim [31, 39]. Such attacks can be carried out by any number of colluding user and/or kernel threads. The attacker is also capable of initializing an arbitrary number of its own enclaves, and we conservatively assume mutual distrust among all enclaves, including those belonging to the same process. While the operating system mediates high-level events in the enclave's life cycle, such as enclave creation and destruction, entries and exits to enclave code, and enclave page faults, SGX hardware ensures that these operations are performed without exposing or altering the enclave's internal data.

While we do not directly address denial-of-service attacks by malicious enclaves, we limit the LLC space that can be allocated to enclaves. A malicious enclave can also attempt to take over the entire cachelet space, causing performance loss for other enclaves. To prevent this attack, additional quality-of-service mechanisms can be put in place even within the enclave partition.

3 Composable Cachelets

Figure 2 describes the high-level design and operation of CC. To track the allocation of cachelets to various enclaves, CC assigns each physical cachelet a unique *cachelet identifier* comprised of the cachelet's set and way offsets within the cache. CC stores the identifier of each unallocated cachelet as an entry in a global *cachelet free list* (CFL), a FIFO-like structure analogous to the free list in register renaming schemes. When an enclave is created, CC pops one or more entries from CFL, and adds them to a hardware *virtual partition table* (VPT), which holds a cachelet identifier entry for each cachelet allocated to the running enclave (a). CC intercepts memory accesses from enclaves and remaps them to cachelets defined in the enclave's VPT. The remapping logic masks the address so that it indexes to a cachelet set (b), and provides the slightly modified replacement logic with a way index to

ensure that only the cachelet's ways are evicted on a miss (c). When the enclave is destroyed, CC gang-invalidates the enclave's cachelets and returns VPT entries to CFL. CC also extends the enclave metadata, which is protected by isolated execution, to store each enclave's VPT data during context switches.

CC allows non-enclave programs to freely access any cache line that is not contained in a cachelet. The modified replacement logic prevents non-enclave accesses from evicting lines in any allocated cachelet (d). CC reserves several ways for non-enclave accesses, ensuring that cache lines are available for non-enclave programs in every cache set.

3.1 Cachelet Addressing and Allocation

In this section, we describe details of cachelet addressing and allocation logic, and also provide an example of cachelet operation.

3.1.1 Address Remapping for Enclaves

Figure 3 shows how CC transparently remaps enclave memory accesses across multiple, non-consecutive cachelets. In conventional caching, addresses are mapped to a cache set using set index bits extracted from the address (Figure 3). CC uses the high-order bits of the set index to select an entry from the VPT. For example, for a set index in the range of 0000-0011, CC uses the high order bits 00 to select the first VPT entry in a four-entry VPT. CC uses the selected VPT entry to remap the original address to a line in a specific cachelet. The VPT entry specifies a cachelet with two fields: a set offset and a way offset. To remap the memory access to a *set* within this cachelet, CC overwrites a portion of the original address' set index with the cachelet set offset, as shown in Figure 4. Whereas the original set index may have mapped to any cache set (Figure 4-a), fixing the high order bits limits the mapping range to sets in the cachelet (Figure 4-b). For example, to force all addresses to access sets `1000-1011`, CC pins the high-order set index bits to `10`.

If the number of sets in all enclaves assigned to cachelets is smaller than the total number of cache sets, CC will remap addresses with different set indexes to the same set within an enclave. To disambiguate these references, CC adds the overwritten index bits to the *tag* bits used to distinguish addresses that map to the same cache line. For instance, the

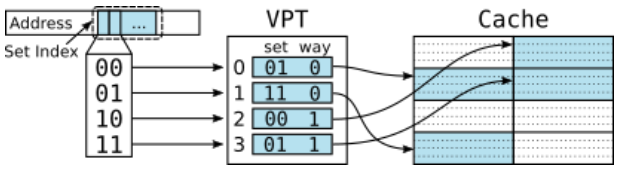

Figure 3: Enclave Set Remapping for CC

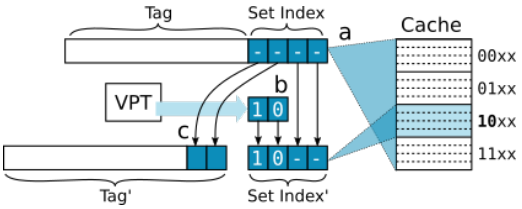

Figure 4: CC strategy to remap memory accesses to cachelet sets

four-cachelet way shown would require two additional tag bits. Only cache ways designated for use by cachelets need these additional bits.

To map enclave addresses to cachelet ways, CC adapts mechanisms from previous, coarse-grained partitioning proposals. Specifically, lightweight hardware described in [34] masks the bit vector used by the cache replacement logic, limiting evictions to a designated range of ways. To generate masking bits for a given cachelet, CC extends the addressing bus to include the way offset bits retrieved from VPT. Simple hardware similar to a decoder converts these bits to the way mask used for the enclave access if a cache miss occurs. Logic described in [34] also prevents cachelet accesses from updating replacement bits for ways outside the cachelet boundaries, eliminating a potential replacement-logic side-channel.

3.1.2 Cachelet Allocation

The number of entries in VPT determines the size of the virtual partition for each enclave in CC. By changing the number of set bits used to index into VPT, CC defines virtual partitions ranging in size from a single cachelet, up to the maximum capacity of VPT. The granularity of indexing is determined by a VPT index mask register shown in Figure 5, which shifts left to widen the indexing range, as cachelets are added from CFL in powers of two. When one partition is present, the register indexes all VPT accesses to `00`, forcing the use of a single cachelet for all enclave addresses. When another entry is added from CFL, the mask is left-shifted to allow access to the first two entries. When two additional cachelets are allocated, the mask shifts again, allowing indexing into all VPT entries.

To enforce strict isolation between cachelets, CC must guarantee that VPT contents for each enclave are disjoint, with no single cachelet identifier entry duplicated between different enclaves. CC accomplishes this by making CFL global and coherent relative to the cache that it services. Similar to the LLC itself, CFL is a unified structure shared by all cores. Pop requests to CFL from different cores are handled in order, so that no enclave can use a particular entry until it has been removed from CFL. This prevents different enclaves from holding the same entry simultaneously, eliminating the possibility of overlapping virtual partitions that experience

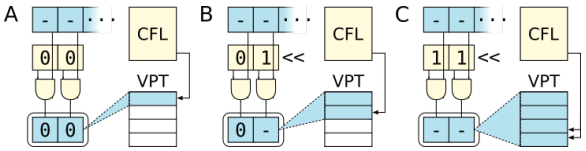

Figure 5: Resizing a virtual partition in CC

information leakage. The use of a global CFL gives CFL pop operations a latency similar to an LLC cache access. However, CFL accesses are rare, occurring only during enclave creation, destruction, and resizing operations.

3.1.3 Optional Partition Sizing

As an optional feature, CC can support the capability to resize virtual partitions to accommodate enclaves with different memory requirements. This can be securely performed in several ways. One approach is to add a new `CCREQ` instruction that allows enclaves to request additional cachelets directly. `CCREQ` transfers control to the OS which can use a new `CCGRANT` instruction to set a flag that triggers a CFL pop to the enclave. Similar to page fault handling in SGX, this interface allows the OS to manage cachelet allocations at a high level to balance memory resources, without exposing enclave's contents.

Newly created enclaves can invoke `CCREQ` to request a static enclave assignment for the duration of its execution. CC can also support dynamic resizing strategies, in which enclaves request additional cachelets during memory-intensive phases of execution, and relinquish them when they are no longer needed. Dynamic resizing can further limit the impact of partitioning on cache performance, and is secure as long as the act of resizing does not depend on sensitive data. Such dependencies are unlikely to occur in practice, and can be identified and avoided during enclave development.

3.1.4 Securing Replacement Policies

Recent research has documented the vulnerabilities that are caused by cache replacement policies. Specifically, if care is not exercised, then simple partitioning schemes can be targeted by attacks on cache replacement logic [11, 68]. The CC design follows DAWG [34], which described lightweight logic to prevent these attacks (Section III.J in [34]). The main idea is to make replacement decisions within a partition independent of the cache accesses to other partitions, thus providing metadata isolation. DAWG considered several replacement policies, including pseudo-LRU, SRRIP [30], and NRU. This logic is compatible with CC, and can be readily integrated into our system with various replacement policies.

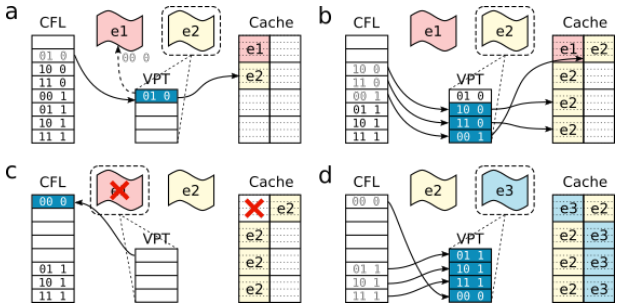

Figure 6: Example of CC operation

3.1.5 Example of CC Operation

To illustrate the flexibility of CC's partitioning policy, Figure 6 shows the operation of CC through the creation, resizing, and destruction of several enclaves. The cache initially contains a single cachelet assigned to enclave $e1$. In the first stage of the example (Figure 6-a), $e1$ is not running on the core, and its VPT state is stored as part of its enclave metadata. When a new enclave $e2$ is created, VPT is initialized with the cachelet identifier entry at the head of CFL. This defines a new 4-set cachelet for $e2$ in way 1. In Figure 6-b, CC expands the virtual partition belonging to $e2$ by popping three additional entries from CFL. In Figure 6-c, $e1$ finishes executing, and CC reclaims its VPT entries. In addition to popping the entries used by $e1$ from VPT and returning them to CFL, CC gang-invalidates the lines in the cachelet. In Figure 6-d, enclave $e3$ is scheduled, and is subsequently scaled to the maximum virtual partition size. The free list assigns $e3$ the remaining entries, including the first entry recovered from $e1$.

3.2 Replacement Deflection for Non-Enclave Accesses

To prevent accesses from non-enclave processes from colliding with cachelet lines held by any enclave, CC extends the cache replacement logic with a small table indicating whether each cachelet position in the cache is occupied. Using the high order set number bits as an index, incoming addresses from non-enclave programs are checked against this table to determine which ways in the accessed set contain enclaves. On a cache miss, CC augments the cache replacement hardware with additional masking logic that uses the enclave-assignment vector for the accessed set to prevent the replacement of enclave ways during evictions.

A commonly used pseudo-least-recently-used (PLRU) policy illustrates the subtleties of the process of replacement deflection. In PLRU replacement, selection bits define a binary tree whose leaf nodes correspond to cache ways. On a cache miss, PLRU selects the eviction target by descending the tree along the path indicated by selection bits at the tree's inner nodes. In Figure 7, a "zero" indicates that the

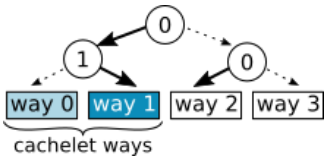

Figure 7: Modified PLRU Replacement Logic

left sub-tree should be descended, while "one" indicates the right sub-tree. After a line is replaced, the nodes along the path to the selected way are then inverted to point away from the last accessed path, approximating a least-recently-used replacement algorithm.

In addition to preventing evictions in cachelet ways at the final branches of the tree, the CC replacement logic must ensure that the selection path never follows any branch that does not have non-enclave ways as children. In Figure 7 for example, the path of replacement must not follow the left edge from the root node, since the sub-tree in this direction leads only to enclave ways that should not be victimized. CC must include logic to divert access down the right edge from the root node, where it will find ways that can be replaced.

The algorithm in Figure 8-a enforces this policy at each level of the PLRU replacement tree traversal. For each node along the traversal path with the selection bit b, the policy selects a new selection bit, b', based on the conditions l and r, which are true, respectively, *iff* cachelets occupy all the ways in the node's left or right subtree. If both sub-trees contain non-cachelet ways, then the existing replacement bit b is used; otherwise, b' is selected to divert the replacement policy away from subtrees whose ways are entirely occupied by cachelets. Note that the final case in the algorithm is an invalid state that is unreachable from the root of the selection tree if at least one way in the set is available for non-enclaved accesses.

From this algorithm, the PLRU masking hardware can be easily derived. Figure 8-b shows the truth table corresponding to the selection algorithm, which can be realized in the gate structure in Figure 8-c. For a given node, the viability of the left subtree, l is determined by taking the logical AND of all the cachelet allocation bits that the subtree leads to.

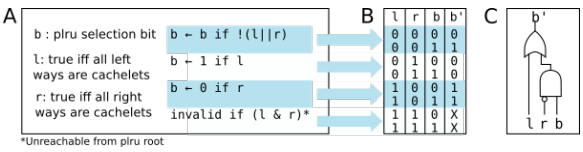

Figure 8: Path Selection Algorithm for Modified PLRU

Figure 9 shows the gate level implementation of the replacement deflection hardware for 8-way PLRU replacement logic. This hardware can be easily scaled to support different cache associativities. Section 5.2 describes how this hardware can be parallelized with existing cache access logic to avoid

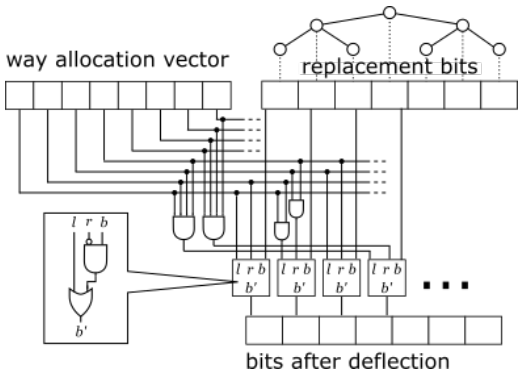

Figure 9: Replacement logic with CC deflection hardware

imposing additional delays.

3.3 Secure Cachelet Eviction

Cachelets in CC are finite resources, which can be exhausted if a large number of enclaves run simultaneously. To prevent cachelet starvation, CC provides logic to safely evict and reassign cachelets allocated to context-switched enclaves. This mechanism, closely modeled on existing page eviction logic in Intel SGX, allows an untrusted OS to instigate evictions through a restricted ISA interface, and performs hardware check to enforce inter-enclave isolation.

To support cachelet evictions, CC hardware maintains a table that maps each currently allocated cachelet to the unique ID of the enclave that currently holds it. To forcibly deallocate a cachelet in response to an empty CFL exception, CC introduces a new instruction called `CSD` (Cachelet Shoot-down). `CSD` takes a cachelet ID as its argument, looks up the associated enclave ID from the mapping table, and uses the enclave ID to access the enclave's metadata. CC extends the metadata with a valid bit for each VPT entry, and the bit associated with the provided cachelet ID is invalidated. CC then adds the invalidated cachelet to CFL. Each time an enclave is scheduled, CC hardware checks the for invalidated VPT entries. If any are found, CC flushes any invalidated VPT entries and replaces them with fresh entries from CFL before the enclave resumes execution, preventing the enclave from accessing reassigned cachelets. Additionally, CC gang invalidates the contents of any *valid* cachelets retained by the enclave.

By immediately replacing reassigned CFL entries and invalidating the contents of the entire virtual partition, CC prevents *controlled-channel* style attacks from exploiting cachelet eviction. In a classical controlled-channel attack, a malicious operating system deliberately invalidates pages belonging to an enclave, and uses the resulting page faults to detect which pages are accessed. If access to a page depends on the value of sensitive data, the attacker can use the sequence of page faults to extract that data. Such attacks are a consideration for CC because because cachelet evictions, like page evictions,

are controlled by the untrusted OS. However, by forcing invalidated cachelets to be replaced from the CFL when an enclave is re-scheduled (rather than deferring replacement to the first time the invalid cachelet is accessed), CC makes cachelet re-assignment independent of the enclave's memory accesses, thus preventing data leakage. Additionally, by clearing the contents of the entire virtual partition, rather than only the invalidated cachelets, CC also prevents an attacker from localizing accesses to a recently evicted cachelet on the basis of timing delays.

Though this invalidation strategy is aggressive, three considerations mitigate the impact on enclave performance. First, a performance-aware, benign operating system can prioritize for eviction of small or single-cachelet enclaves, which will quickly re-populate their virtual partitions when rescheduled. Second, enclaves are likely to be used in multi-tenanted clusters, in which a load balancer could strategically distribute enclave workloads to limit the occurrence of evictions. Finally, additional logic could be introduced to detect controlled-channel style attacks on CC and relax the cachelet replacement rules when no threat exists. This could be accomplished by adding a simple hardware counter to detect the anomalous rates of cachelet evictions needed to mount a controlled-channel attack. A CC system with this optimization would enable the strict invalidation policy described above only if the counter exceeded the threshold for an effective controlled channel attack, and would otherwise operate in a mode that retained the contents of valid cachelets and defer cachelet reassignment until the invalidated cachelet is accessed.

The cachelet eviction mechanism assumes that the victim enclave is context-switched when an eviction occurs. Otherwise, a running victim enclave may access lines in a reassigned cachelet before the cachelet has been invalidated. Because the operating system is not trusted, CC must guarantee that the targeted enclave is context switched before the `CSD` instruction updates CFL. To enforce this requirement, CC provides a new `CTRACK` instruction, which must be issued before the `CFL` update takes effect. `CTRACK` causes the hardware to track all processors running the enclave that owns the specified cachelet, and prevents `CSD` from completing until the OS has issued interrupts to evict all of that enclave's threads. This mechanism is modeled on the `ETRACK` that SGX uses to safeguard enclave page evictions.

3.4 Compatibility of CC with Cache Slicing

Some modern architectures divide the cache into two or more *slices* that can be accessed in parallel, and use a hash function to map memory accesses to alternating slices. Using the hardware described in Sections 3.1 or 3.2, CC can enforce a shared cachelet layout accross each cache slice, and apply address remapping after the slice has been selected. Because the cachelet layout is replicated across each slice, each enclave receives the same number of cache lines as under a simple addressing scheme; the lines are simply distributed across multiple slices to exploit parallel access.

Top-Level Structures		
Σ	$::= \langle \kappa; \mu; \rho; \pi \rangle$	*runtime state*
κ	$::= \langle F; V; C; R \rangle$	*composable cache*
μ	$::= \overline{b \mapsto D}$	*memory*
ρ	$::= \overline{r \mapsto v}$	*registers*
π	$::= \overline{p \mapsto \langle \varepsilon; l \rangle}$	*processes*
CC-Related Structures		
F	$::= \overline{c}$	*cachelet free list*
V	$::= \overline{e \mapsto L}$	*VPT*
C	$::= \overline{c \mapsto \langle vb; t; D \rangle}$	*way-set cache*
L	$::= \overline{s \rightarrow c}$	*remapping list*
D	$::= \overline{o \mapsto n}$	*data block*
c	$::= \langle w; s \rangle$	*cachelet index*
vb	$::= \{\mathsf{valid}; \mathsf{dirty}\}$	*validity bit*
b	$\in \mathbb{BLOCK}$	*block index*
o	$\in \mathbb{OFFSET}$	*offset*
w	$\in \mathbb{WAY}$	*way index*
s	$\in \mathbb{SET}$	*set index*
t	$\in \mathbb{TAG}$	*cache tag value*
PLRU-Related Structures		
R	$::= \overline{s \mapsto T}$	*set-indexed PLRU*
T	$::= \langle \varsigma; e; T; T \rangle \mid A$	*PLRU tree*
A	$::= \langle w; e \rangle$	*PLRU leaf*
ς	$::= \mathtt{LMRU} \mid \mathtt{RMRU}$	*selection bit*
Memory/Register-Related Structures		
l	$\in \mathbb{ADDR}$	*memory address*
v	$::= \iota \mid n$	*memory value*
ι	$\in \mathbb{INST}$	*instruction*
n	$::= \emptyset \mid \mathsf{num}$	*data*
num	$\in \mathbb{UINT}$	*number*
r	$\in \mathbb{REG}$	*register name*
Enclave-Related Structures		
ε	$::= \langle e; E \rangle$	*enclave state*
E	$::= \overline{\mathsf{e} \mapsto \langle l; n \rangle}$	*enclave memory range*
e	$::= \mathsf{e} \mid \bot$	*enclave ID*
e	$\in \mathbb{ENCLAVE}$	*raw enclave ID*

Figure 10: Runtime Definitions

4 A Formal Security Analysis

CC provides strong security guarantees for enclave programs, which we rigorously establish in this section. The key results are cache-aware enclave access isolation (Theorem 4.2) and immunity from side-channel attacks (Theorem 4.3).

4.1 Structures and Definitions

The runtime state consists of the states of the composable cache (κ), the memory (μ), the register file (ρ), and multiple

name	function & argument	return value	description
CC allocation	$\Uparrow^{n}_{\mathsf{e}}\ \kappa$	κ'	allocates n CCs to enclave e in κ, resulting in κ'
CC deallocation	$\Downarrow_{\mathsf{e}}\ \kappa$	κ'	deallocates CCs for enclave e in κ, resulting in κ'
CC hit read	$\kappa \uparrow^{\varepsilon}_{c} l$	$\langle n;\kappa'\rangle$	reads data n from location l through CC c held by enclave ε, resulting in κ'
CC hit write	$\kappa \downarrow^{\varepsilon}_{c} (l,v)$	κ'	updates CC c held by enclave ε in κ when location l is updated to v, resulting in κ'
CC miss read	$(\kappa,\mu) \uparrow^{\varepsilon}_{c} l$	$\langle n;\kappa'\rangle$	reads data n from l in memory μ, while updating CC c held by enclave ε in κ, resulting in κ'
CC miss write	$(\kappa,\mu) \downarrow^{\varepsilon}_{c} (l,v)$	(κ',μ')	updates cachelet c held by enclave ε in κ when location l in memory μ is updated to v, resulting in κ' and μ'
CC resize	$\circlearrowleft_{\mathsf{e}}\ \kappa$	κ'	resizes (doubles) the cachelets in e, resulting in κ'
memory read	$\mu\{l\}$	n	read from location l of memory μ, resulting in n
memory write	$\mu\{l \mapsto v\}$	μ'	writes v to location l of memory μ, resulting in μ'
memory reinitialization	$\triangledown^{\mathsf{e}}_{\varepsilon}\mu$	μ'	memory in μ for enclave e in executions ε is reinitialized, resulting in μ'
enclave creation	$\varepsilon_{\mu}\{\mathsf{e} \mapsto \langle l;n\rangle\}$	ε'	adds an enclave e with memory range $\langle l;n\rangle$ to enclaves ε when memory is μ, resulting in enclaves ε'
active enclave update	$\varepsilon \blacktriangleleft e$	ε'	updates the active enclave in ε to e, resulting in enclaves ε'
enclave removal	$\varepsilon - \mathsf{e}$	ε'	removes enclave e from ε, resulting in enclaves ε'

Figure 11: Auxiliary Functions (see definitions in the appendix)

execution sequences (π). The structure of the runtime state Σ is formally defined in Fig. 10.

Throughout this section, notation $\overline{X}$ represents a sequence of $X_1,\ldots,X_m$ for some $m \geq 0$. Sequence in the form of $\overline{X \mapsto Y}$ is also called a mapping when $\overline{X}$ are distinct. For any mapping, we use notations $M[X \mapsto Y]$, $M\backslash X$, $dom(M)$, $ran(M)$ to refer to the update, restriction, domain, and range of mapping M, with standard definitions.

Cache Replacement Logic Our formal system is capable of modeling the behavior of PLRU. The PLRU replacement logic is indexed by set IDs, where the replacement of ways for each set is maintained by a PLRU tree, denoted as T, as we described in § 3.2. For each node in a PLRU tree, selection bit $\varsigma = \texttt{LMRU}$ (1 in § 3.2) means the left side of the binary tree is accessed more recently; $\varsigma = \texttt{RMRU}$ (0 in § 3.2) means the right side is accessed more recently. Occupancy e is the enclave ID if the entire subtree is occupied by enclave e.

Function $replace(T,e)$ computes the way to be evicted for enclave e given the current state of the PLRU tree T, defined as follows:

$$\begin{array}{rcl} replace(\langle \texttt{LMRU};e;T_1;T_2\rangle,e) & = & replace(T_2,e) \\ replace(\langle \texttt{RMRU};e;T_1;T_2\rangle,e) & = & replace(T_1,e) \\ replace(\langle w;e\rangle,e) & = & w \\ replace(\langle \varsigma;e;T_1;T_2\rangle,e') & = & replace(T_1,e') \\ & \text{if} & e' \neq e, e' \sqsubset T_1 \\ replace(\langle \varsigma;e;T_1;T_2\rangle,e') & = & replace(T_2,e') \\ & \text{if} & e' \neq e, e' \sqsubset T_2 \end{array}$$

where auxiliary function $e \sqsubset T$ is a predicate that holds when enclave e occupies some ways defined in PLRU (sub-)tree T. It is is defined as:

$$\begin{array}{rcl} e' \sqsubset \langle \varsigma;e;T_1;T_2\rangle & = & (e=e') \vee (e' \sqsubset T_1) \vee (e' \sqsubset T_2) \\ e' \sqsubset \langle w;e\rangle & = & (e=e') \end{array}$$

The *replace* definition here subsumes non-enclave cache replacement, when $e = \perp$. The definition includes the following cases: (1) If a PLRU node indicates its entire subtree is occupied by e and the left subtree is more recently accessed, the eviction way is in the right subree; (2) If a PLRU node indicates the entire subtree is occupied by e and the right subtree is more recently accessed, the eviction way is in the left subree; (3) If the leaf node is currently occupied by enclave e, the way it represents is to be evicted; (4, 5) If a PLRU node indicates its subtree is not entirely occupied by e, the eviction way resides in the subtree that contains nodes/subtrees that enclave e occupies.

The other operation for the PLRU tree is its update. Function $update(T,w,e)$ computes the updated tree T' given the current state of the PLRU tree is T, and its way w is going to be accessed/allocated by/to enclave e. We defer its formal definition to the appendix.

Cachelets In addition to the cache replacement logic, a composable cache consists of a *free list* (F), a *virtual partition table* (V) and the way-set cache (C). Each cache block contains the validity bit vb, the tag t, and the data block itself D. The cache block is indexed by a pair $\langle w;s\rangle$, where w is the way index and s is the set index. Given a memory block index

Multi-Process Reduction

$$[\textsc{Multi}]\quad \kappa,\mu,\rho,\pi[p\mapsto\langle\varepsilon;l\rangle] \xrightarrow{\{\langle p;\omega\rangle\}} \kappa',\mu',\rho',\pi[p\mapsto\langle\varepsilon';l+n\rangle] \quad \text{if}\quad \kappa,\mu,\rho,\varepsilon,\mu(l)\xrightarrow[n]{\omega}\kappa',\mu',\rho',\varepsilon'$$

Single-Process Reduction

$$[\textsc{LoadHit}]\quad \kappa,\mu,\rho,\varepsilon,\texttt{LOAD}\ l\ r \xrightarrow[1]{\mathsf{H}(v,c)} \kappa',\mu,\rho[r\mapsto v],\varepsilon \quad \text{if}\quad \langle v;\kappa'\rangle=\kappa\uparrow_c^{\varepsilon} l$$

$$[\textsc{LoadMiss}]\quad \kappa,\mu,\rho,\varepsilon,\texttt{LOAD}\ l\ r \xrightarrow[1]{\mathsf{M}(v,c)} \kappa',\mu,\rho[r\mapsto v],\varepsilon \quad \text{if}\quad \langle v;\kappa'\rangle=(\kappa,\mu)\uparrow_c^{\varepsilon} l$$

$$[\textsc{StoreHit}]\quad \kappa,\mu,\rho,\varepsilon,\texttt{STORE}\ r\ l \xrightarrow[1]{\mathsf{H}(\rho(r),c)} \kappa\downarrow_c^{\varepsilon}(l,\rho(r)),\mu,\rho,\varepsilon$$

$$[\textsc{StoreMiss}]\quad \kappa,\mu,\rho,\varepsilon,\texttt{STORE}\ r\ l \xrightarrow[1]{\mathsf{M}(\rho(r),c)} \kappa',\mu',\rho,\varepsilon \quad \text{if}\quad \kappa',\mu'=(\kappa,\mu)\downarrow_c^{\varepsilon}(l,\rho(r))$$

$$[\textsc{Resize}]\quad \kappa,\mu,\rho,\varepsilon,\texttt{CCREQ} \xrightarrow[1]{\mathsf{R}} \circlearrowleft_{\mathsf{e}}\ \kappa,\mu,\rho,\varepsilon \quad \text{if}\quad \varepsilon=\langle \mathsf{e};E\rangle$$

$$[\textsc{Create}]\quad \kappa,\mu,\rho,\varepsilon,\texttt{CREATE}\ r_1\ r_2\ r_3\ r_4 \xrightarrow[1]{\top} \Uparrow_{\rho(r_1)}^{\rho(r_2)}\kappa,\mu,\rho,\varepsilon' \quad \text{if}\quad \varepsilon'=\varepsilon_\mu\{\rho(r_1)\mapsto\langle\rho(r_3);\rho(r_4)\rangle\}$$

$$[\textsc{Enter}]\quad \kappa,\mu,\rho,\varepsilon,\texttt{ENTER}\ r \xrightarrow[1]{\top} \kappa,\mu,\rho,\varepsilon\blacktriangleleft\rho(r)$$

$$[\textsc{Exit}]\quad \kappa,\mu,\rho,\varepsilon,\texttt{EXIT} \xrightarrow[1]{\top} \kappa,\mu,\rho,\varepsilon\blacktriangleleft\bot$$

$$[\textsc{Destroy}]\quad \kappa,\mu,\rho,\varepsilon,\texttt{DESTROY}\ r \xrightarrow[1]{\top} \Downarrow_\varepsilon\kappa,\triangledown_\varepsilon^{\rho(r)}\mu,\rho,\varepsilon-\rho(r)$$

$$[\textsc{BrTrue}]\quad \kappa,\mu,\rho,\varepsilon,\texttt{BR}\ r\ r' \xrightarrow[\rho(r')]{\top} \kappa,\mu,\rho,\varepsilon \quad \text{if}\quad \rho(r)\neq 0$$

$$[\textsc{BrFalse}]\quad \kappa,\mu,\rho,\varepsilon,\texttt{BR}\ r\ r' \xrightarrow[1]{\top} \kappa,\mu,\rho,\varepsilon \quad \text{if}\quad \rho(r)=0$$

Figure 12: A Core Operational Semantics

b, there is a bijective function $\beta:\mathbb{BLOCK}\rightleftharpoons\mathbb{SET}\times\mathbb{TAG}$ to compute its set index s and tag t in the cache.

The top part of Fig. 11 provides the list of auxiliary functions used for defining the behavior of composable cachelets. As expected, many functions will be defined through the *replace* and *update* functions of the PLRU logic. To improve readability, we choose to defer the definitions of these functions to the appendix.

Memory and Registers Memory is block-based, and it is represented as a mapping from block indices b to data blocks D. Each data block in turn is a mapping from offsets o to values (v). Given a memory address l, there is a bijective function $\alpha:\mathbb{ADDR}\rightleftharpoons\mathbb{BLOCK}\times\mathbb{OFFSET}$ to compute its block index b and offset o in the block. Auxiliary functions related to the lifecycle of memory use are described in Fig. 11. In addition, we overload operator $\Sigma[l\mapsto n]$ for memory update in the runtime state, defined as $\langle\kappa;\mu\{l\mapsto n\};\rho;\pi\rangle$ where $\Sigma=\langle\kappa;\mu;\rho;\pi\rangle$. Both instructions ($\iota$) and memory data ($n$) are represented as memory objects in our formal model, denoted as v.

The register file is a mapping from names (r) to values (v).

Enclaves and Processes We represent executions π as a mapping from execution (process) IDs to its *enclave state* (ε) and its *program counter* (l). Each enclave state ε is a pair $\langle e;E\rangle$, where e indicates the *active enclave*, i.e., the enclave in effect for the current execution; E is a mapping for enclave IDs to their range of enclave-private memory locations. The latter in turn is indicated by a pair, the beginning memory location and the length. When no enclave is active, we set e as $\bot$. We further use metavariable ε for this special form of enclave states. Functions related to the lifecycle of the enclaves — creation, update, removal — are listed in the Table in Fig. 11.

4.2 Operational Semantics

We model the behavior of programs with enclave lifecycle instructions (`CREATE`, `ENTER`, `EXIT`, `DESTROY`), memory/cache access instructions (`LOAD` and `STORE`), the branching instruction (`BR`), and the resizing instruction (`CCREQ`). The semantics of `CCGRANT` is captured through `CREATE`.

We define small-step operational semantics in Figure 12, including the multi-execution relation $\Rightarrow$ and single-execution relation $\rightarrow$. $\Sigma\xRightarrow{\Omega}\Sigma'$ says that runtime state Σ reduces to Σ' with observations Ω. We use $\Sigma\xRightarrow{\Omega}{}^{*}\Sigma'$ to represent the reflexive and transitive closure of $\xRightarrow{\Omega}$ where Ω is unionized. $\sigma,\iota\xrightarrow{\omega}\sigma'$ says single-execution state σ reduces to σ' with instruction ι producing single-execution observation ω. Single-execution states σ is defined as $\langle\kappa;\mu;\rho;\varepsilon\rangle$, with single-execution observation ω including cache hit (H), cache miss (M), cache resize (R), or none ($\top$). A multi-execution observation Ω is a pair which consists of both the execution ID and the single-execution observation. To bridge the two, we define restriction operator $\Omega|_p$ as identical to Ω except that any element $\langle p';\omega\rangle$ where $p'\neq p$ is removed. Program counters are memory addresses, and we use $\bullet$ to represent the program counter when the program halts.

As the reduction rules show, our semantics faithfully account for the behavior of the cache and the enclave, the focus of this work. [LoadHit] and [LoadMiss] capture the memory/cache load behavior in the presence of a cache hit and a cache miss, respectively. Observe that the two rules have different observations, H and M respectively. In other words, our semantics expose the cache hit/miss as observable states to the attacker, and we show later that even with this assumption, CC is immune to side channel attacks. Similarly, memory/cache store behavior is captured by [StoreHit] and [StoreMiss], which again produces different observations. Cachelet resizing is captured by [Resize]; in our model, we assume the attacker may observe the cache resizing event too. The lifecycle of enclaves are modeled by the next 4 rules, with [Create] for creating an enclave-protected memory, [Destroy] for enclave destruction, [Enter] for entering an enclaved execution, and [Exit] for exiting an enclaved execution. The four parameters of the `CREATE` instruction are the registers that keep the enclave ID, the number of cachelets requested by the enclave, the starting location of the enclave-protected memory, and the size of the enclave-protected memory area. CCs are allocated upon the `CREATE`

instruction, and deallocated upon the DESTROY instruction. Our execution model for enclaved executions is general, as it captures the dynamic nature of the enclave lifecyle. Finally, [BRTRUE] and [BRFALSE] captures the behavior of the common control flow construct, branching.

4.3 Metatheory

We now establish two important properties of our design, whose proofs can be found in the Appendix.

We start with an account of the PLRU replacement logic. First, let us define a *tree context* $\mathbb{T}$ as either a hole $[]$ or $\langle \varsigma; e; []; T \rangle$ or $\langle \varsigma; e; T; [] \rangle$.

Lemma 4.1 (PLRU Tree Update Isolation)**.** *If* $T' = \text{update}(T, w, e)$*, then for any tree context* $\mathbb{T}$ *such that* $T = \mathbb{T}[T_0]$ *and* $T_0 = \langle \varsigma; \mathsf{e}; T_1; T_2 \rangle$ *for some* ς*,* T_1*,* T_2 *and* $\mathsf{e} \neq e$ *then there exists some tree context* $\mathbb{T}$ *such that* $T' = \mathbb{T}'[T_0]$*.*

The lemma above is simple, but it lays the foundation on why our system can thwart side channel attacks that rely on the PLRU tree logic. In other words, a non-enclave access ($e = \perp$), or an access from another enclave (e is some enclave ID but not e, cannot alter the PLRU tree state associated with enclave e. As the *update* definition is used by several cachelet operations, it ultimately ensures cachelet operations cannot lead to side channels through the PLRU logic.

Theorem 4.2 (Cache Isolation across Multiple Enclave Programs)**.** *Given two executions* $p_1 \neq p_2$ *in* Σ *and* $\Sigma \overset{\Omega}{\Longrightarrow} \Sigma'$*, if* $\Omega = \{\langle p_1; \omega \rangle\}$*, then* $\omega \neq M(n,c)$ *and* $\omega \neq H(n,c)$ *and* $\omega \neq S(n,c)$ *for any* $c \in \text{ran}(V(\mathsf{e}_2))$ *where* $\Sigma = \langle \kappa; \mu; \rho; \pi \rangle$ *and* $\kappa = \langle F; V; C; R \rangle$ *and* $\pi(p_i) = \langle \langle \mathsf{e}_i; E_i \rangle; l_i \rangle$ *for* $i = 1, 2$*.*

This theorem says that two programs with active enclaves cannot access the same cachelet. This theorem states how the *software* behaves under CC *hardware*, i.e., each enclave occupies a unique, non-overlapping virtual cache partition.

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

Theorem 4.3 (Immunity to Side Channel Attacks)**.** *Given* Σ *and some* l*,* e*,* p *s.t.* $\text{epriv}(l, \mathsf{e}, p, \Sigma)$*, some* $n_1 \neq n_2$*,* $p' \neq p$*, two reductions* $\Sigma[l \mapsto n_i] \overset{\Omega_i}{\Longrightarrow}{}^{*} \kappa_i, \mu_i, \rho_i, \pi_i$ *where* $\pi_i(p') = \langle \varepsilon_i; \bullet \rangle$ *for* $i = 1, 2$*, if* $R \notin \Omega_1 \cup \Omega_2$*, then* $\varepsilon_1 = \varepsilon_2$*,* $\Omega_1|_{p'} = \Omega_2|_{p'}$*.*

This important theorem says that the enclave-private value of a victim p (stored at location l) cannot be inferred by the attacker p': the attacker makes identical observations regardless of what value the location holds. This is a strong result, because we do not make any assumption on what the attacker program is, subsuming both a passive attacker or an active attacker. Furthermore, it assumes the attacker can take advantage of the (timing) difference of a load miss and a load hit, can introspect all its program values, and even know what way-sets in the cache the attacker program has accessed.

The resize-free pre-condition ($R \notin \Omega_1 \cup \Omega_2$) in the Theorem is needed because enclave-private data may guard a branching instruction (BR). If a CCREQ instruction is issued in one branch but not the other, the impact of CCREQ on cachelet allocation may influence the behavior of the attacker program differently (see Section 3.1.3 for details) and thus leak one bit of information. As cachelet resizing is a low-level system operation, the common use scenario for CCREQ is to have this instruction automatically inserted by the compiler, or automatically issued by the runtime. The pre-condition can be easily satisfied by not inserting/issuing CCREQ when the execution is within a branch, or inserting/issuing CCREQ in both branches.

The pre-condition only becomes a cause of concern if *programmable* cachelet resizing is supported, i.e., CCREQ is exposed to end programmers without any restriction, so that it becomes a *programming* construct itself. In this context, the precondition here demonstrates a trade-off between security and performance. For the same program, a performance-biased execution may follow the semantics we described in this paper (but may leak a bit), whereas a security-bias execution may treat CCREQ as a no-op (and thus no leak). Furthermore, the program pattern described above is identical to implicit information flow [44, 49], a well-studied topic in program analysis. Thus, a strengthened defense opportunity exists with software-hardware co-design. For instance, a compiler could reject a program where the pre-condition fails to satisfy, or an automatic compiler instrumentation can be defined to insert CCREQ in one branch when it appears in the other branch of a BR instruction.

5 Evaluation of CC

In this section, we present evaluation of CC from the standpoint of performance, power, area, delay and the design complexity.

5.1 Performance Evaluation

First, we describe our benchmarks and methodology, and then present the simulation results.

5.1.1 Benchmarks and Methodology

For performance evaluation of CC, we used gem5 [6] full system simulator configured with three levels of caches and targeting x86 ISA. The complete configuration of the simulated system is shown in Table 1. Our experimental analysis is based on three sets of benchmarks. Since the most expected application of CC is to secure small secrets maintained inside an enclave, we first evaluated cryptographic programs, including three traditional applications (AES, Blowfish and SHA) taken from MiBench suite [27] and five Post-Quantum Cryptography (PQC) applications, namely BIG-QUAKE [4],

CRYSTAL-KYBER [8], CFPKM [15], Compact LWE [37], and DAGS [3]. Second, to gauge the impact of isolating larger applications, we evaluated programs from SPEC 2017 suite [13]. Third, to evaluate the impact on securing parallel applications, we also evaluated CC with PARSEC benchmarks [5]. We also used selected memory-intensive SPEC 2017 benchmarks to demonstrate the impact of CC on performance of non-enclave applications. Depending on the experiment, either the entire program is assumed to be executing inside an enclave, or it is assumed to be executed in a regular mode outside an enclave. We present the performance impact of CC in terms of IPC metric (committed Instructions per Cycle) normalized to the baseline configuration with non-partitioned caches.

For SPEC 2017 benchmarks, we fast-forwarded simulations for 1 Billion instructions and simulated for the next 1 Billion instructions. Due to the short setup phase of cryptography programs, we simulated 1 Billion instructions for each of them from the beginning. For PARSEC, we used 2-core system and bypassed the booting process. We then simulated until completion.

Hardware Parameters	
Core #	1-core (Crypto Programs and SPEC2017) and 2-cores (PARSEC)
Core Parameters	8-way out-of-order cores, 64k TAGE branch predictor, 4096 BTB entries, 16 RAS entries, 192-entry ROB, 128-entry LSQ, 64-entry Instruction Queue, 256-entry float and integer registers
L1i/d Cache	private, 32KB size, 8 ways, 64 sets, PLRU replacement, 64B cache line size, tag/response/data latency 2 cycles each
L2 Cache	private, 256KB size, 4 ways, 1024 sets, PLRU replacement, 64B cache line size, tag/response/data latency 8 cycles each
L3 Cache	shared, 8192KB size, 16 ways, 8192 sets, PLRU replacement, 64B cache line size, MESI coherence protocol, tag/response/data latency 16 cycles each
DRAM	4GB size, 4GB channel capacity, DDR4-2400 x64 channel, 4 devices per rank, 1 rank per channel, 1GB per device,
CC	3-cycle additional L3 latency, 16-entry VPT, 32KB cachelet size
Software Parameters and Benchmarks	
gem5	Version 2.0
Kernel	gem5 system emulation (Crypto Programs and SPEC2017) and VM Linux 4.19.83 (PARSEC)
SPEC 2017	Version 1.0.2 (evaluated 14 benchmarks)
PARSEC	Version 3.0-beta-20150206 (evaluated 6 benchmarks with *simdev* inputs, ran to completion)
Security benchmarks	3 security benchmarks from MiBench (AES, Blowfish and SHA), plus 5 PQC applications with optimized implementations

Table 1: Configuration of the Simulated System

5.1.2 Performance of Cryptographic Programs

Figure 13 shows the IPC values for crypto benchmarks, normalized to the baseline IPC. In addition to the baseline system, we evaluated 3 configurations of CC: 1-way virtual partition with 8 cachelets (for the total partition size of 256KB), 2-way virtual partition with 4 cachelets per way, and 2-way virtual partition with 8 cachelets per way. Note that since in this work we apply partitioning only to shared LLC, the total size of a virtual partition must be at least the size of the L2 cache. Since the memory demands of crypto applications are modest, the locality exploitation and thus high performance can be achieved with minimal partition sizes in most cases, as shown in Figure 13. One outlier in this set of benchmarks was BIG-QUAKE (one of the PQC benchmarks) that exhibited 8.4% performance loss for 8 cachlets with 1 way and around 10% performance loss for 4 cachelets with 2 ways. When partition size was increased to 512KB from 256KB, the performance impact was reduced to 3.7%.

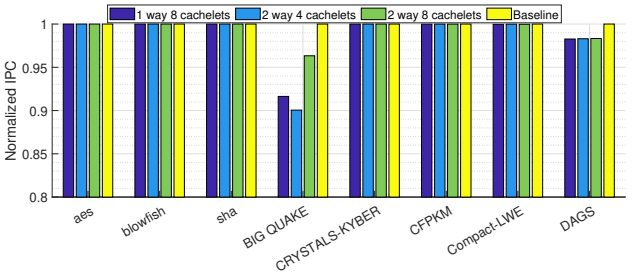

Figure 13: Performance of Cryptographic Programs

5.1.3 Performance on SPEC 2017 Benchmarks

While we envision that a primary application of CC would be to protect small enclaves from cache-based information leakage, we also evaluated performance impact of protecting larger applications, such as SPEC 2017 programs. Figure 14 shows these results as commit IPCs normalized to the baseline case. The CC configurations in this case included 1-way, 2-way, 4-way and 8-way virtual partitions. For each of these, we considered various number of cachelets per way, as indicated in the graph legend. The total partition size varies between 512KB and 4MB, it is computed as the product of the number of ways, the number of cachelets per way, and the cachelet size (32KB).

As seen from the results, CC shows a small performance degradation for multiple benchmarks even with modest virtual partition sizes. For example, benchmarks such as *cactusBSSN*, *deepsjeng*, *exchange2*, *leela*, *xalancbmk* and *xz* do not exhibit any slowdown regardless of the partition size. Specifically, all these programs have performance degradation lower than 5% for all configurations (0.8%, 0.4%, 0.2%, 3.5%, 1.6%, and 1.5% was the largest loss recorded, respectively). Not surprisingly, these applications also feature high L1 and L2 cache hit rates, demonstrating high locality of references and making them less sensitive to LLC. In these experiments, we assumed three additional cycles of delay to the LLC due to the logic required to implement CC. This is a very conservative estimate, which is justified in Section 5.2.

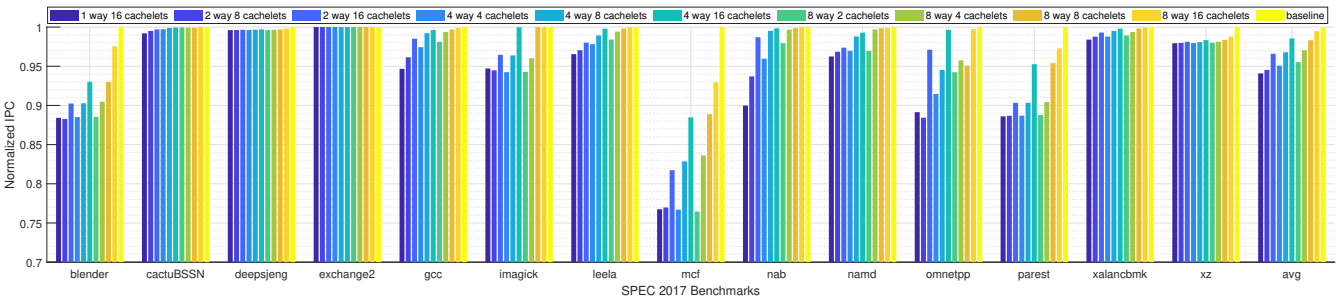

Figure 14: Performance Impact of CC on SPEC 2017 Benchmarks

Two benchmarks in our set - *gcc* and *imagick* - resulted in moderate performance losses of 5.4% and 5.8% respectively with 16 cachelets (512KB) allocated to them. Allocating a quarter of the LLC for a virtual partition brings performance losses for these programs to under 1% - this can be achieved either with 4-way partitions with 16 cachelets per way, or with 8-way partitions with 8 cachelets per way.

Some SPEC benchmarks showed more substantial performance impact from CC. For example, the IPC loss for *blender*, *mcf*, *omnetpp* and *parest* is 11.7%, 23.5%, 11.4% and 11.5% respectively for the smallest allocations we considered. As we increase the partition associativity and the number of cachelets per way, performance gradually improves. For example for *mcf*, performance degradation goes down to 11.6% when 4 ways of the LLC with 16 cachelets per way are allocated. Similar results are observed when 8 cachelets with 8 ways are used. If the same configurations are used for *omnetpp*, performance loss is reduced to below 1%, but it remains at 7% for *blender* and 4.7% for *parest*. On the average across SPEC benchmarks, the performance degradation for the smallest allocated partition (512 KB) is 5.9%. It is reduced to 1.45% for the quarter of the LLC (2MB) and to 0.6% for the largest allocated partition (4MB).

These results demonstrate that even if one wishes to support the entire execution of a larger program (such as a SPEC benchmark) inside an enclave with isolated cache partition, performance losses are quite modest for the majority of programs. Furthermore, since benchmarks show different levels of sensitivity to the allocated LLC space, it is important to investigate techniques to dynamically provision LLC space to application in a secure manner - future work can investigate such mechanisms.

5.1.4 Performance of PARSEC Benchmarks

Parallel shared memory applications (exemplified by PARSEC) can also benefit from cache leakage protection afforded by CC. In this case, since these applications are generally more memory intensive, larger partitions are needed to moderate their performance impact. Figure 15 shows the results for selected PARSEC applications running with quarter of the LLC size allocated for them where the total virtual partition has 8 cachelets with 8 ways. As seen from these results, the average performance loss is 7.8%.

We examined *blackscholes*, *facesim*, *ferret*, *fluidanimate*, *raytrace* and *swaptions* to completion as our benchmarks which displayed 6%, 1.4%, 1.5%, 27.7%, 3.2% and 3.4% performance degradation respectively. Of these, the largest loss by far was observed for *fluidanimate*, this benchmark can benefit from larger partitions.

5.1.5 Performance Impact on Non-Enclave Programs

Finally, we evaluated performance impact of CC on non-enclave applications, where some portion of LLC is taken away by enclaves. For this experiment, we selected five SPEC 2017 benchmarks that were the most sensitive to the LLC allocations and most susceptible to performance degradation from CC, as shown in Figure 14: *blender*, *mcf*, *omnetpp*, *parest* and *xz*. Figure 16 shows IPC of these applications executed in a non-enclave mode normalized to the baseline (baseline is the rightmost set of bars). The second rightmost set of bars shows the impact of the 3-cycle additional LLC latency due to CC. Although it is reasonable to assume that CC can be implemented with non-enclave accesses incurring no extra delay (as they do not require remapping), we nevertheless conservatively show the results where 3 extra cycles are added to the LLC latency. The difference between two rightmost bars is 1.4% on the average - that is the cost of additional cycles. On top of that, we show the performance for three different cases - when 14, 12 and 8 ways are allocated to non-enclave programs, assuming that the rest of the cache space are occupied by enclaves.

All of the benchmarks exhibited performance loss of under 10%. When 8 ways (half of the LLC) are allocated to non-enclave programs, *mcf* show the worst result with 7.1% performance loss (again, assuming that 3 additional cycles of latency are present; without that the impact will be much smaller as can also be seen from this graph). For all other benchmarks, performance loss is less than 3%. On average,

CC provides non-enclave programs shown on this graph with 97.3%, 98.2%, 98.4% and 98.6% of the baseline performance for 8, 12, 14 and 16 ways allocation respectively, assuming 3-cycle additional latency. Again, these benchmarks are the ones that were most sensitive to LLC in our previous experiments.

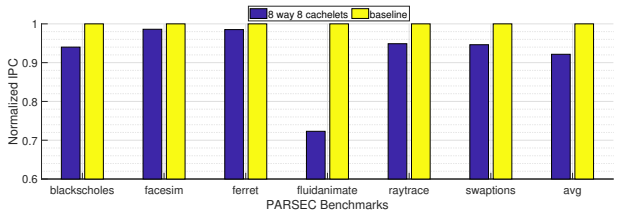

Figure 15: Performance Impact on PARSEC Benchmarks

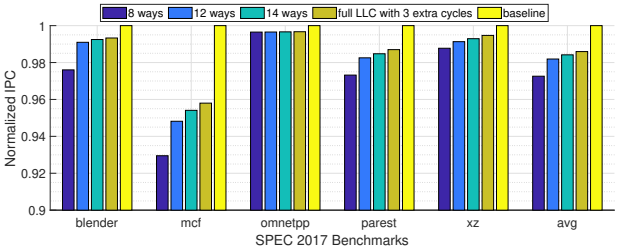

Figure 16: Performance of non-enclave SPEC'17 Benchmarks

5.2 Delay Analysis

First, we examine the complexity of VPT, which is the central piece of CC logic. The VPT and associated address remapping logic is similar in structure to a register alias table (RAT), where the RAT has the number of rows equals to the number of ISA registers, and the number of bits in each row equal to the log of the number of physical registers. Typically, renaming a register takes a single cycle even in multi-ported RATs that are necessary to implement superscalar cores. With a single-ported VPT, the access delays will reduce even more. Therefore, it is reasonable to expect that the address remapping logic in the LLC can be implemented at the cost of one additional cycle of LLC access latency. However, for conservative performance estimation, we provisioned three additional cycles to the LLC latency to support CC, and all out performance results assume three-cycle latency.

The replacement deflection logic described in 3.2 can be implemented in parallel with L3 tag checking process to avoid additional delays. The way replacement is a simple two-step process that involves 1) looking up the cachelet occupancy bits for the selected set and 2) applying the masking logic described in Figure 8. In a design where each cachelet occupies 1/16 of the sets in its assigned way(s), The first step requires indexing into a 16-entry table, similar to the VPT. In a cache with 8 columns of cachelets, the second step requires the five-gate-deep structure shown in Figure 9 to mask the replacement bits (one additional level of gates would be needed for each additional column of cachelets). In contrast, tag matching and hit/miss determination requires: 1) indexing and reading the set, which is a time consuming operation for a large LLC, and 2) comparing tags in the selected set against the tag bits of the address being accessed to determine a hit or a miss. Because the masked replacement bits are not used until a hit or a miss is determined, the identification of the victim way proceeds in parallel with the cache access (once the set is determined through the remapping logic), thus removing this operation from the critical path. In any case, we provision three additional cycles to the LLC latency to support CC for conservative performance estimation.

5.3 Area and Power Analysis

The architectural footprint of CC is limited, arising mostly from the CFL, VPTs, and additional tag bits required for each way that is equipped to host cachelets. We used the McPAT tool [36] to estimate the complexity of these components. As a reference architecture, we modified one of the baseline processor descriptions (Intel Xeon) that are provided with McPAT, and altered the Caches, Register File, TLB, BTB, LSQ, ROB, and fetch/decode/issue/commit widths to match the architecture defined in Table 1. The technology parameter was changed to 22mm. All other defaults were retained.

CFL: The CFL is structurally similar to a register free list. It consists of n entries, each containing $log_2 n$ bits where n is the number of cachelets. For example, a CFL supporting 64 cachelets contains 64 6-bit entries. We estimated the area and power of a 64-cachelet CFL by simulating a 64-entry register free list eith McPat. The area overhead was 0.04% relative to the baseline processor, and the peak and runtime dynamic were 0.09% and 0.16%, respectively. Because the CFL would be accessed only when cachelets are allocated, power results based on the register free list likely overestimate the actual CFL power consumption.

VPT: To support the largest allocation of cachelets in our evaluations, each hardware VPT requires 16 six-bit entries, or 96 D flip-flops, in addition to six 16-1 multiplexers required for address remapping (We assume a maximum of 16 cachelets per enclave, with a total of 64 cachelets in the whole system). This structure must be replicated for each hardware thread sharing the LLC, and is similar in structure to a standard register alias table (RAT) as we described previously. To estimate the area and power of a 16-entry VPT, we modeled a retirement RAT in McPAT that mapped 16 architectural registers to 64 physical registers. The resulting area was 0.0004 mm^2 for a single table, or 0.008 mm^2 for the two

VPTs needed by the two-core simulated processor. The two VPTs represent less than 0.002% of the total processor area. The runtime dynamic was 0.04% over the baseline processor, while the peak dynamic was an increase of 0.006% - all are negligible overheads.

L3 Cache: To support 16 cachelets per enclave, the cache must be extended with 4 additional tag bits, resulting in a 1.92% increase in LLC area, or a total 0.80% increase in area for the reference processor. The impact on processor peak and runtime dynamic are, respectively, 0.29% and 0.33%. Table 2 summarizes the area/power overheads of CC relative to the reference processor.

Component	Area (mm^2)	Peak Dynamic W	Runtime Dynamic W
Base CPU	45.18 (100%)	70.07 (100%)	35.12 (100%)
CFL	0.02 (+0.04%)	0.06 (+0.09%)	0.06 (+0.16%)
VPT × 2	<0.001 (+0.002%)	0.004 (+0.006%)	0.013 (+0.04%)
CC tag bits	0.36 (+0.80%)	0.21 (+0.29%)	0.11 (+0.33%)
Total	0.38 (+0.84%)	0.27 (+0.38%)	0.18 (+0.51%)

Table 2: Area/Power estimates for CC components

6 Related Work

Cache partitioning schemes with strictly non-overlapping partitions completely eliminate information leakage. However, existing schemes use way-granularity approaches that do not scale efficiently to large numbers of small enclaves, or trust system software to enforce isolation, making them incompatible with the isolated execution security model. For example, DAWG [34] and CATalyst [38] trust the OS to control partitioning; if the OS is compromised, security guarantees may no longer be sustained. Furthermore, DAWG has a hard limit on the number of supported protection domains, limiting its scalability. NoMo caches [20] offer partitioning without software support, but is not scalable (since the entire cache way gets allocated) and is thus not suitable for the LLC, especially in cloud-based systems that can simultaneously run a large number of enclaves. In addition, as some cache ways are shared in NoMo, leakage can occur if the victim's accesses spill into the shared portion of the cache. Other partitioning schemes like Intel's CAT [29] were designed for quality of service, and do not guarantee isolation between processes occupying different partitions. Other designs such as [48,51,67] also fall into this category. Some approaches [9, 18] perform set-partitioning via page coloring. The limitation is that large regions of data may need to be moved around in memory when allocating cache sets, because cache set allocation is bound to physical addresses. Page coloring is also not readily compatible with large pages, potentially impacting the TLB reach [34].

HybCache [19] provides soft cache partitions for codes requiring isolated execution protection. HybCache requires fully-associative search within the subcache ways - this is expensive and may not easily scale to large LLCs. In addition, HybCache does not enforce strict isolation, as normal programs can still access the entire cache. CURE [2] proposed a customizable architecture for securing enclaves from side-channel attacks. However, cache partitioning is also done at the granularity of ways, unlike fine-grain partitioning in CC.

In a concurrent work, Saileshwar et al. proposed Bespoke Cache Enclaves - a set-based cache partitioning scheme, where the cache space is divided into non-overlapping clusters composed of multiple consecutive sets [50]. The key of this proposal is flexible indexing mechanism that restricts access from a particular security domain only to the cache partitions belonging to that domain. In contrast, CC design offers both way and set-based partitioning and ties partitioning decisions to the operations of secure enclaves.

In the space of randomized designs, recent work proposed obfuscating the cache index [46, 47, 63]. CEASER [46] dynamically encrypts the cache index using low-latency encryption. The original CEASER proposal proved vulnerable to high-speed key recovery attacks [47], and the revised version of CEASER proposed in [47] to address this vulnerability was recently compromised by the Brutus attack proposed in [7]. While stronger address encryption can fortify CEASER, this will have an impact on access latency. Attacks similar to [47] can be used to break another randomization mechanism called SCATTER-CACHE that was proposed in [64]. Another recent work [45] also demonstrated security problems with randomization-based caches, such as CEASER-S. In general, partitioning provides fundamentally stronger security guarantees.

In other related efforts, SHARP [70] offers modifications to the cache replacement policy of the LLC to avoid cross-core inclusion victims that are determined to be the root cause of attacks. Similarly, RIC [32], avoids back-invalidations of read-only data from private caches, which avoids successive access to the LLC. Both techniques trust the OS to support critical operations: in RIC, the read-only pages need to be marked, while SHARP reports suspicious behavior to the OS (which can choose to ignore the warning), and relies on a modified *clflush* instruction. Table 3 compares CC with previous solutions in terms of scalability, OS involvement and security.

Design	**CC**	DAWG [34]	CATalyst [38]	NoMo [20]	SecDCP [61]	PLCache [62]	HybCache [19]	CEASER [47]
Fine-grained	**Yes**	No	Yes	No	No	Yes	Yes	Yes
HW only	**Yes**	No	No	Yes	No	No	Yes	Yes
Strict isolation	**Yes**	Yes	Yes	No	Yes	Yes	No	No

Table 3: A Comparison of Cache Designs for Security

Designing formal frameworks for side-channel attacks is an emerging but actively pursued direction. Several formal frameworks have illustrated the vulnerabilities due to speculation [14, 16, 25, 40], whereas our focus is on cache-based channels, and more importantly, the defense against such attacks. Formal foundations for programs running on enclaves

have also been proposed (e.g., [23, 24, 54]), without focusing on side channel attacks or their defenses. More broadly, programming language techniques have been proposed for addressing side-channel attacks, including abstract interpretation [59, 66], symbolic execution [12, 26, 60], program analysis and transformation [21, 58, 65]. This category of related work is more distant to ours, in that they propose software defense whereas ours is hardware-centric. Finally, hardware description languages have been designed to mitigate timing channels [71–73].

7 Concluding Remarks

It is important to protect caches from side-channel attacks, particularly in environments with isolated execution. In these settings, application secrets are shielded from direct access even by high-privilege software, but can still be leaked through a side channel. Composable Cachelets (CC) is a new and scalable dynamically-partitioned last-level cache design that strongly isolates secure enclaves from other applications and from each other. CC partitions can be dynamically managed as enclaves enter and leave the system, thus adjusting the cache configuration to the system demands. We demonstrate that CC provides provable protection from cache side-channel attacks through a rigorous security model based on cache-aware and enclave-aware operational semantics. We demonstrate that CC can be implemented with minimal area and power overhead, and many applications can benefit from CC protection with modest performance cost. These properties make CC an attractive design choice for SGX-style isolated execution systems.

8 Acknowledgements

We would like to thank Ms. Atsuko Shimizu for insightful discussions during the early stages of this work. We would also like to thank anonymous reviewers for their valuable feedback. This research was supported in part by NSF Award CNS-2053391.

Appendix

In this appendix, we provide additional definitions of the formal system, omitted from the main text of the paper. The proof for the theorems and lemmas can be found online [1]

9 Cache Replacement Logic Definitions

The *update* function is defined as follows:

$$
\begin{array}{rcl}
update(\langle \varsigma; e; T_1; T_2\rangle, w, e) & = & \langle \texttt{LMRU}; e; T_1'; T_2\rangle \\
 & \text{if} & w \prec T_1 \\
 & & T_1' = update(T_1, w, e) \\
update(\langle \varsigma; e; T_1; T_2\rangle, w, e) & = & \langle \texttt{RMRU}; e; T_1; T_2'\rangle \\
 & \text{if} & w \prec T_2 \\
 & & T_2' = update(T_2, w, e) \\
update(\langle \varsigma; e; T_1; T_2\rangle, w, e) & = & \langle \varsigma; e; T_1; T_2\rangle \\
 & \text{if} & \neg w \prec T_1, \neg w \prec T_2 \\
update(\langle \varsigma; e; T_1; T_2\rangle, w, e') & = & \langle \varsigma; T_1' \sqcap T_2'; T_1'; T_2'\rangle \\
 & \text{if} & e' \neq e \\
 & & T_i' = update(T_i, w, e') \\
 & & \text{for } i = 1, 2 \\
update(\langle w; \bot\rangle, w, e) & = & \langle w; e\rangle \\
update(A, w, e) & = & A \text{ otherwise}
\end{array}
$$

where auxiliary function $w \sqsubset T$ is a predicate that holds when way w occupies some leaf in PLRU (sub-)tree T. Furthermore, operator $T \sqcap T'$ computes the common enclave that holds all ways in both T and T'; when the ways in T and T' are held by more than one enclave (including non-enclave $\bot$), the operator returns $\bot$. The two functions are defined as follows:

$$
\begin{array}{rcl}
w \prec \langle \varsigma; e; T_1; T_2\rangle & = & (w \prec T_1) \vee (w \prec T_2) \\
w \prec \langle w'; e\rangle & = & (w = w') \\
\\
\langle \varsigma; e; T_1; T_2\rangle \sqcap \langle \varsigma'; e; T_1'; T_2'\rangle & = & e \\
T_1 \sqcap T_2 & = & \bot \text{ otherwise}
\end{array}
$$

Just as the *replace* function, the *update* function here is also defined into cases depending on whether the PLRU (sub-)tree is entirely occupied by the enclave of interest. Cases 1, 2, 3 says that if the PLRU (sub-)tree is indeed entirely occupied by the enclave of interest, the PLRU tree update is defined according to the PLRU algorithm itself: if its left subtree is more recently accessed, the selection bit is set to `LMRU`; if the right subtree is more recently accessed, the selection bit set to `RMRU`. Case 4 says if the PLRU (sub-)tree is not entirely occupied by the enclave of interest, we will ignore the selection bit, and only update the PLRU tree (recursively) for subtrees entiredly occupied by the enclave of interest. Case 5 is a special case that says that if a way (leaf node in the PLRU tree) is currently not occupied by any enclave, it can be allocated to the enclave of interest; this case is used for cachelet allocation. Case 6 says that for all other cases, the leaf node in the PLRU tree remains unchanged.

This function unifies several use scenarios that involves the PLRU tree update (and it will be used in definitions that capture these scenarios):

- When a way is a hit, we need to update the path on the PLRU tree to make sure the selection bit (i.e., MRU) indeed reflects this way is the most recently accessed way.
- When a way is a miss and its content is replaced with new data, we also need to update the path on the PLRU tree to make sure the selection bit (i.e., MRU) indeed reflects this way is the most recently accessed way.
- When a way is newly allocated to an enclave.

Note also that this definition subsumes the non-enclave access, where the enclave of interest (the third argument of the function) is set to $\bot$.

[1] `http://www.cs.binghamton.edu/~davidl/papers/CCProof.pdf`

10 Auxiliary Definitions

Cachelet Operators First, let us introduce a covenience function for gang invalidation. Function $inv(C,F)$ is defined as $C[c_1 \mapsto \langle \mathsf{dirty}; t_1; D_1\rangle]\ldots[c_n \mapsto \langle \mathsf{dirty}; t_n; D_n\rangle]$ where $C(c_i) = \langle vb_i; t_i; D_i\rangle$ for $i = 1..n$ and $F = c_1, \ldots, c_n$.

The following CC operations are defined, where $\kappa = \langle F; V; C; R\rangle$:

- **Cachelet Allocation**: Operator $\Uparrow^n_{\mathsf{e}}\ \kappa$ allocates n cachelets to e in κ, defined as $\langle F''; V[\mathsf{e} \mapsto L]; C; R[s_1 \mapsto T'_1]\ldots[s_n \mapsto T'_n]\rangle$ where $F = F', F''$, and $F' = c_1, \ldots, c_n$ and $L = 0 \mapsto c_1, \ldots n-1 \mapsto c_n$, and $c_i = \langle w_i; s_i\rangle$ and $R(s_i) = T_i$, $T'_i = update(T_i, w_i, \mathsf{e})$ for $i = 1..n$.
- **Cachelet Deallocation**: Operator $\Downarrow_{\mathsf{e}}\ \kappa$ deallocates the cachelets for e in κ, defined as $\langle F, F'; V \setminus \mathsf{e}; inv(C, F'); R[s_1 \mapsto T'_1]\ldots[s_n \mapsto T'_n]\rangle$ where $ran(V(e)) = F'$ and $F' = c_1, \ldots, c_n$ and $c_i = \langle w_i; s_i\rangle$ and $R(s_i) = T_i$, $T'_i = update(T_i, w_i, \mathsf{e})$ for $i = 1..n$. .
- **CC Hit Read**: Operator $\kappa \uparrow^{\varepsilon}_c l$ reads data in location l through cachelet c held by enclave ε, defined as $\langle D(o); \kappa'\rangle$ if $C(c) = \langle vb; t; D\rangle$, $c = \langle w; s'\rangle = V(\varepsilon)(s)$ where $\alpha(l) = \langle b; o\rangle$, $\beta(b) = \langle s; t\rangle$ and $\kappa' = \kappa = \langle F; V; C; R[s' \mapsto T']\rangle$ and $T' = update(R(s'), w, \varepsilon)$.
- **CC Miss Read**: Operator $(\kappa, \mu) \uparrow^{\varepsilon}_c l$ reads data in location l and updates cachelet c held by enclave ε in κ when location l in memory μ is updated, defined as $\langle D'(o); \kappa'\rangle$ such that

$$\kappa' = \langle F; V[\varepsilon \mapsto V(\varepsilon)[s \mapsto c]]; C[c \mapsto \langle \mathsf{valid}; t; \mu(b)\rangle]; R[s' \mapsto T']\rangle$$

  if $\alpha(l) = \langle b; o\rangle$, $\beta(b) = \langle s; t\rangle$, $c' = \langle w'; s'\rangle = V(\varepsilon)(s)$, $C(c') = \langle vb'; t'; D'\rangle$, $t' \neq t$, $w'' = replace(R(s'), \varepsilon)$, $C(\langle w''; s'\rangle) = \langle vb; t''; D\rangle$, and $T' = update(R(s'), w'', \varepsilon)$.
- **CC Hit Write**: Operator $\kappa \downarrow^{\varepsilon}_c (l, v)$ updates cachelet c held by enclave ε in κ when location l is updated, defined as $\langle F; V; C[c \mapsto \langle \mathsf{dirty}; D[o \mapsto v]; R[s' \mapsto T']\rangle]\rangle$ if $C(c) = \langle vb; t; D\rangle$, $c = \langle w; s'\rangle = V(\varepsilon)(s)$ where $\alpha(l) = \langle b; o\rangle$, $\beta(b) = \langle s; t\rangle$, $c = \langle w; s;\rangle$, and $T' = update(R(s'), w, \varepsilon)$.
- **CC Miss Write**: Operator $(\kappa, \mu) \downarrow^{\varepsilon}_c (l, v)$ updates cachelet c held by enclave ε in κ when location l in memory μ is updated, defined as $\langle \kappa'; \mu'\rangle$ such that

$$\kappa' = \langle F; V[\varepsilon \mapsto V(\varepsilon)[s \mapsto c]]; C[c \mapsto \langle \mathsf{dirty}; t; \mu(b)[o \mapsto v]\rangle]; R[s' \mapsto T']\rangle$$
$$\mu' = \mu[b' \mapsto D]$$

  if $\alpha(l) = \langle b; o\rangle$, $\beta(b) = \langle s; t\rangle$, $c' = \langle w'; s'\rangle = V(\varepsilon)(s)$, $C(c') = \langle vb'; t'; D'\rangle$, $t' \neq t$, $w'' = replace(R(s'), \varepsilon)$, $C(\langle w''; s'\rangle) = \langle vb; t''; D\rangle$, $\beta(b') = \langle s; t''\rangle$ and and $T' = update(R(s'), w'', \varepsilon)$.
- **CC Resize**: Operator $\circlearrowleft_{\mathsf{e}}\ \kappa$ resizes (doubles) the cachelets in e, defined as $\langle F_2; V[e \mapsto L_3, L_1]; C; R\rangle$ if $F = F_1, F_2$, $V(e) = L_3$ and $|F_1| = |L_3| = n$. and $L_1 = n-1 \mapsto c_1, \ldots, 2 \times n - 1 \mapsto c_n$ and $F_1 = c_1, \ldots, c_n$.

Enclave Operators The key operators of enclaves are defined as follows:

- **Enclave Creation**: Operator $\varepsilon_\mu\{\mathsf{e} \mapsto \langle l; n\rangle\}$ is defined as $\langle e; E[\mathsf{e} \mapsto \langle l; n\rangle]\rangle$ where $\varepsilon = \langle e; E\rangle$, defined only when $\mathsf{e} \notin dom(E)$, $\mu\{l\} = \mu\{l+1\}\ldots\mu\{l+n-1\} = 0$.
- **Active Enclave Update**: Operator $\langle e'; E\rangle \blacktriangleleft e$ is defined as $\langle e; E\rangle$, defined only if $e \in dom(E) \cup \{\perp\}$.
- **Enclave Elimination**: Operator $\langle e; E\rangle - \mathsf{e}$ is defined as $\langle e; E \setminus \mathsf{e}\rangle$.

Memory Operators For memory, we use $\mu\{l\}$ to refer to $\mu(b)(o)$ where $\alpha(l) = \langle b; o\rangle$. We use $\mu\{l \mapsto v\}$ to refer to $\mu[b \mapsto D']$ and $D' = D[o \mapsto v]$ where $\alpha(l) = \langle b; o\rangle$. Operator $\bigtriangledown^{\mathsf{e}}_{\varepsilon}\mu$ says memory μ for enclave e in executions ε is reinitialized, defined as $\mu\{l \mapsto 0\}\ldots\{l+n-1 \mapsto 0\}$ where $\varepsilon = \langle e; E\rangle$ and $E(\mathsf{e}) = \langle l; n\rangle$.

References

[1] I. Anati, S. Gueron, S. Johnson, and V. 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*, vol. 13. ACM New York, NY, USA, 2013.

[2] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "{CURE}: A security architecture with customizable and resilient enclaves," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[3] G. Banegas, P. S. Barreto, B. O. Boidje, P.-L. Cayrel, G. N. Dione, K. Gaj, C. T. Gueye, R. Haeussler, J. B. Klamti, O. N'diaye, D. T. Nguyen, E. Persichetti, and J. E. Ricardini, "Dags: Key encapsulation using dyadic gs codes," *Journal of Mathematical Cryptology*, vol. 12, no. 4, pp. 221–239, 2018. [Online]. Available: https://doi.org/10.1515/jmc-2018-0027

[4] M. Bardet, E. Barelli, O. Blazy, R. Canto–Torres, A. Couvreur, P. Gaborit, A. Otmani, N. Sendrier, and J.-P. Tillich, "Big quake (binary goppa quasi-cyclic key encapsulation)." [Online]. Available: https://bigquake.inria.fr/

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," Princeton University, Tech. Rep. TR-811-08, January 2008.

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

[7] R. Bodduna, V. Ganesan, P. Slpsk, C. Rebeiro, and V. Kamakoti, "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser," *IEEE Computer Architecture Letters*, 2020.

[8] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle, "Crystals - kyber: A cca-secure module-lattice-based kem," in *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, 2018, pp. 353–367.

[9] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, S. 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*. ACM, 2019, pp. 42–56.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: Sgx cache attacks are practical," *arXiv preprint arXiv:1702.07521*, p. 33, 2017.

[11] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "Reload+ refresh: Abusing cache replacement policies to perform stealthy cache attacks," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[12] R. L. Brotzman, S. L. Liu, D. Zhang, G. Tan, and M. T. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," in *IEEE S&P 2019*, 2018.

[13] J. Bucek, K.-D. Lange, and J. V. Kistowski, "Spec cpu2017: Next-generation compute benchmark," *ICPE: ACM/SPEC International Conference on Performance Engineering*, pp. 41–42, 2018.

[14] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, 2020, pp. 913–926.

[15] O. Chakraborty, J.-C. Faugère, and L. Perret, "CFPKM : A Key Encapsulation Mechanism based on Solving System of nonlinear multivariate Polynomials," UPMC - Paris 6 Sorbonne Universités ; INRIA Paris ; CNRS, Research Report, Dec. 2017. [Online]. Available: https://hal.inria.fr/hal-01662175

[16] K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *Proceedings of the Computer Security Foundations Symposium (CSF)*, 2019.

[17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Leaking enclave secrets via speculative execution," *arXiv preprint arXiv:1802.09085*, 2018.

[18] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 857–874.

[19] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "Hybcache: Hybrid side-channel-resilient caches for trusted execution environments," 2020.

[20] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 35, 2012.

[21] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 431–446. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534804

[22] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev *et al.*, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 693–707.

[23] C. Fournet and J. Planul, "Compiling information-flow security to minimal trusted computing bases," in *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Barthe, Ed., vol. 6602. Springer, 2011, pp. 216–235. [Online]. Available: https://doi.org/10.1007/978-3-642-19718-5_12

[24] A. Gollamudi and S. Chong, "Automatic enforcement of expressive security policies using enclaves," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, E. Visser and Y. Smaragdakis, Eds. ACM, 2016, pp. 494–513. [Online]. Available: https://doi.org/10.1145/2983990.2984002

[25] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: principled detection of speculative information flows," *CoRR*, vol. abs/1812.08639, 2018. [Online]. Available: http://arxiv.org/abs/1812.08639

[26] S. Guo, M. Wu, and C. Wang, "Adversarial symbolic execution for detecting concurrency-related cache timing leaks," in *ESEC/SIGSOFT FSE*. ACM, 2018, pp. 377–388.

[27] 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*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2001, pp. 3–14. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/WWC.2001.15

[28] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions." *HASP@ ISCA*, vol. 11, 2013.

[29] C. Intel, "Improving real-time performance by utilizing cache allocation technology," *Intel Corporation, April*, 2015.

[30] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 60–71, 2010.

[31] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 72.

[32] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "Ric: relaxed inclusion caches for mitigating llc side-channel attacks," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 2017, pp. 1–6.

[33] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362804

[34] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[35] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[36] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 469–480. [Online]. Available: https://doi.org/10.1145/1669112.1669172

[37] D. Liu and S. Nepal, "Compact-lwe-mq: Public key encryption without hardness assumptions," Cryptology ePrint Archive, Report 2020/974, 2020, https://ia.cr/2020/974.

[38] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 406–418.

[39] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[40] R. McIlroy, J. Sevcík, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *CoRR*, vol. abs/1902.05178, 2019.

[41] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *HASP@ ISCA*, vol. 10, 2013.

[42] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 69–90.

[43] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "Copycat: Controlled instruction-level attacks on enclaves," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 469–486.

[44] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99, 1999, p. 228–241.

[45] A. Purnal, G. Lukas, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *IEEE Symposium on Security and Privacy*, 2021, pp. 469–486.

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

[47] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 360–371. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322246

[48] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432. [Online]. Available: https://doi.org/10.1109/MICRO.2006.49

[49] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, p. 5–19, Sep. 2006. [Online]. Available: https://doi.org/10.1109/JSAC.2002.806121

[50] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2021, pp. 37–49.

[51] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *ACM SIGARCH Computer Architecture News*, vol. 39. ACM, 2011, pp. 57–68.

[52] J. Shi, X. Song, H. Chen, and B. 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*, ser. DSNW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 194–199. [Online]. Available: http://dx.doi.org/10.1109/DSNW.2011.5958812

[53] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: enabling microarchitectural replay attacks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 318–331.

[54] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2435–2450. [Online]. Available: https://doi.org/10.1145/3133956.3134098

[55] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 991–1008.

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

[57] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 178–195.

[58] J. Wang, C. Sung, and C. Wang, "Mitigating power side channels during compilation," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'19)*, 02 2019.

[59] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, "Identifying cache-based side channels through secret-augmented abstract interpretation," in *NDSS*, 05 2019.

[60] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, 2017, pp. 235–252.

[61] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: Secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 74:1–74:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898086

[62] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ACM SIGARCH Computer Architecture News*, vol. 35. ACM, 2007, pp. 494–505.

[63] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2008, pp. 83–93.

[64] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: thwarting cache attacks via cache set randomization," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 675–692.

[65] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *ISSTA*. ACM, 2018, pp. 15–26.

[66] M. Wu and C. Wang, "Abstract interpretation under speculative execution," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, 2019, pp. 802–815. [Online]. Available: http://doi.acm.org/10.1145/3314221.3314647

[67] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555778

[68] W. Xiong and J. Szefer, "Leaking information through cache lru states," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 139–152.

[69] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 640–656.

[70] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 347–360, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3140659.3080222

[71] D. Zagieboylo, G. E. Suh, and A. C. Myers, "Using information flow to design an isa that controls timing channels," in *32nd IEEE Computer Security Foundations Symp. (CSF)*, June 2019. [Online]. Available: http://www.cs.cornell.edu/andru/papers/hyperisa

[72] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2012, p. 99–110. [Online]. Available: https://www.cs.cornell.edu/andru/papers/pltiming.html

[73] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 503–516. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694372

[74] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 871–882. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978324