Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures

Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, Emmanuel Stapf
Technical University of Darmstadt, Germany
{ghada.dessouky, pouya.mahmoody, ahmad.sadeghi, emmanuel.stapf}@trust.tu-darmstadt.de

***Abstract*— Shared cache resources in multi-core processors are vulnerable to cache side-channel attacks. Recently proposed defenses such as randomized mapping of addresses to cache lines or well-known cache partitioning have their own caveats: Randomization-based defenses have been shown vulnerable to newer attack algorithms besides relying on weak cryptographic primitives. They do not fundamentally address the root cause for cache side-channel attacks, namely, mutually distrusting codes sharing cache resources. Cache partitioning defenses provide the strict resource partitioning required to effectively block all side-channel threats. However, they usually rely on way-based partitioning which is not fine-grained and cannot scale to support a larger number of protection domains, e.g., in trusted execution environment (TEE) security architectures, besides degrading performance and often resulting in cache underutilization.**

To overcome the shortcomings of both approaches, we present a novel and flexible set-associative cache partitioning design for TEE architectures, called Chunked-Cache. The core idea of Chunked-Cache is to enable an execution context to "carve" out an exclusive *configurable chunk* of the cache if the execution requires side-channel resilience. If side-channel resilience is not required, mainstream cache resources can be freely utilized. Hence, our proposed cache design addresses the security-performance trade-off practically by enabling efficient selective and on-demand utilization of side-channel-resilient caches, while providing well-grounded future-proof security guarantees. We show that Chunked-Cache provides side-channel-resilient cache utilization for sensitive code execution, with small hardware overhead, while incurring no performance overhead on the OS. We also show that it outperforms conventional way-based cache partitioning by 43%, while scaling significantly better to support a larger number of protection domains.

I. Introduction

The outbreak of micro-architectural attacks has demonstrated the crucial implications of performance-boosting processor optimizations on the security of our computing platforms [54], [1], [60], [56], [52], [65], [100], [31], [28], [27], [58], [4], [3], [88], [68], [90], [92], [16], [17], [81], [15]. One of the most popular features, and also the subject of many recent attacks, are shared resources such as caches. Caches provide orders-of-magnitude faster memory accesses and large last-level-caches (LLCs) are usually shared across multiple processor cores to maximize utilization.

The Problem with Caches. When a sensitive (victim) and malicious (adversary) application run simultaneously on different cores and share the LLC, cache side channels can be exploited by the adversary to leak sensitive information, such as private keys. The timing difference between a cache hit and miss – which is why caches are used in the first place – is the most commonly exploited side channel to infer the memory access patterns of a victim application [38], [99], [35], [44], [34], [43], [46], [64], [32], [71], [36], [37], [98], [91]. In typical side-channel attacks [71], [43], [46], [64], [38], [99] the adversary deduces the victim's memory access patterns by exploiting that both the victim and adversary compete for shared set-associative cache resources, which are designed in such a way that a larger number of memory lines are mapped to a smaller number of cache ways/entries in each cache set.

Besides compromising cryptographic implementations [7], [64], [71], [99], more recent attacks have had even stealthier impact such as bypassing address space layout randomization (ASLR) or leaking privacy-sensitive human genome indexing computation [34], [32], [14], [35], [36], leaving millions of platforms vulnerable. Even trusted execution environment (TEE) security architectures which aim to protect sensitive services by compartmentalizing them in isolated execution contexts, called *enclaves*, e.g., Intel SGX [41], [21] or ARM TrustZone [5], have been shown vulnerable to these attacks, thereby undermining their acclaimed privacy and isolation guarantees [14], [83], [69], [30], [59], [101]. This is alarming since TEE architectures are now widely deployed by major cloud providers, e.g., Microsoft Azure, Google Cloud, Alibaba Cloud and IBM Cloud, to offer *confidential computing*, where sensitive workloads are protected in enclaves.

The Problem with Recent Cache Defenses. To mitigate cache side-channel attacks, various approaches have been proposed over the years. These solutions range from time-constant cryptographic implementations [26], [25], [55] to software- and hardware-based approaches that modify the cache organization itself. The latter can be broadly classified into either cache partitioning [29], [94], [51], [61], [23], [51] or randomization-based techniques [63], [89], [77], [78], [96], [87] that attempt to obfuscate the relationship between the memory address and the cache location to which it is mapped.

More recently, various schemes for a randomized memory-to-LLC mapping, such as CEASER, ScatterCache, and Phantom-Cache [89], [77], [78], [96], [87] have been proposed to mitigate these attacks by obfuscating the adversary's view of which cache lines actually get evicted. However, such defenses continue to evict cache lines from a small number of locations in a shared cache, thus cache set-based conflicts essentially still occur. While these defenses were shown effective against the eviction set construction algorithms and techniques at the time, subsequent more efficient eviction set construction algorithms [78] were able to undermine them. Consequently,

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23110
www.ndss-symposium.org

enhancements to these defenses were proposed [78], only to be rendered ineffective again by yet another attack vector, e.g., weak low-latency cryptographic primitives [74], [10], or alternative attack techniques that exploited design/implementation flaws in the proposed defenses [86].

Caught in an arms race, randomization-based defenses remain as good as the best known attack technique at the time and are constructed to mitigate very specific side channels and attack strategies [12], with no future-proof and well-grounded security guarantees. They only make the attacks computationally more difficult, but do not address their fundamental root cause, i.e., sharing set-associative caches across mutually distrusting processes. These schemes also assume that all execution contexts require side-channel resilience without providing mechanisms for a selective configuration of side-channel-resilience, thus, taxing the entire system with the resulting performance impact. In practice, however, only a small portion of the workload is usually security-/privacy-sensitive and requires this sophisticated security guarantee.

On the other hand, strict partitioning approaches promise well-grounded security guarantees due to their cache isolation across different execution contexts. However, these approaches usually rely on conventional way-based partitioning [6], [57], [94], [51], [23], and thus, are not fine-grained, cannot scale with an increasing number of execution contexts and large LLCs, or do not provide support for shared memory.

With these limitations in mind, we argue that a more future-proof and practical approach for side-channel resilient cache computing is to address the root cause of these attacks, namely, sharing set-associative cache structures across mutually distrusting execution contexts. Meanwhile, performance, usability, flexibility and scalability should still be preserved. We further observe that, in practice, cache side-channel resilience is most prominently a concern in dedicated security architectures, e.g., TEE security architectures. Thus, it is crucial to develop side-channel-resilient cache designs that cater for the security/functionality requirements of these architectures, e.g., with integrated support for enabling the side-channel resilience (and the performance cost) only for specific execution contexts that require it.

Our Goals. In this work, we aim to selectively enforce clean partitioning of the cache resources across mutually distrusting execution contexts that require side-channel resilience, such that all side channels are blocked (including stealthy cache occupancy channels [84] which are not mitigated by recent works [96], [23]), while maintaining the desired performance requirements.

To address this performance-security trade-off, we propose a new cache design for TEE security architectures, which we call CHUNKED-CACHE, that enables each execution context or domain to "carve" out its exclusive cache *sets*, if desired. These sets essentially constitute an independent set-associative cache, which we call the domain's *cache chunk*, that this domain can utilize exclusively but fully and efficiently, unlike in cache partitioning, e.g., way-based partitioning. A domain can flexibly request and configure 1.) whether it requires side-channel-resilient cache utilization, 2.) for which memory regions, and 3.) the required capacity of this exclusive side-channel-resilient *cache chunk*. Memory accesses by a domain that requires side-channel-resilient cache utilization are mapped exclusively to its *cache chunk*, while mainstream cache resources are freely and conventionally utilized whenever side-channel-resilience is not required. Enabling this on-demand flexibility per domain *practically* requires addressing multiple key challenges. Firstly, efficient design mechanisms are required to configure the memory-to-set mapping at runtime for each domain depending on its chunk capacity, while preserving conventional cache behavior for the rest of the execution. Secondly, it must be ensured that the operating system performance is not degraded as cache sets get allocated exclusively to domains. Finally, seamless support must be provided for shared memory between domains to meet the security and functionality requirements of different sensitive applications.

Our Contributions. Our main contributions are as follows:

- We present CHUNKED-CACHE, a novel cache architecture for TEE security architectures, which enables a selective, flexible and scalable configuration of side-channel resilient caches for execution domains, without degrading the OS performance.
- We address the performance-security trade-off by enforcing clean cache partitioning that blocks all cache side channels by allocating exclusive cache chunks for different domains. In doing so, future-proof and solid security assurances are guaranteed, while still preserving performance, functionality and compatibility requirements.
- We extensively evaluate the cycle-accurate performance overhead of CHUNKED-CACHE for compute-intensive SPEC CPU2017 workloads and I/O-intensive real-world applications. We show that it outperforms shared cache utilization in some cases, that the OS performance even improves owing to CHUNKED-CACHE's flexible cache utilization, and that CHUNKED-CACHE outperforms partitioning (way-based) by 43% while also scaling better to support a larger number of protection domains.
- We implement and evaluate a hardware prototype of CHUNKED-CACHE. We show that it incurs a minimal 2.3% memory overhead relative to a 16 MB LLC, 1.6% logic overhead relative to a single-core RISC-V processor, and 12.3% LLC power consumption overhead.

II. CACHE ATTACKS & DEFENSES

Next, we briefly introduce recent cache side-channel attacks that are relevant for our work and a summary of the shortcomings of recent defenses that our work overcomes.

A. *Cache Side-Channel Attacks*

Cache side-channel attacks have been shown to constitute a profound threat that underlies popular attacks such as Spectre [54] and Meltdown [60], besides threatening a wide spectrum of platforms and architectures [59], [64], [43], [102], and even TEE architectures [14], [83], [69], [30], [59], [101]. The attacks usually work by provoking controlled evictions of the victim's cache line, such that the inherent information leakage from the access-timing difference between cache hits and misses can be exploited by the adversary. This can be achieved using three main approaches:

- Access-based approaches where the target address is explicitly accessed and flushed [38], [99], [35], [44], [34].

- Conflict-based approaches where the adversary triggers a controlled cache contention in the same cache set of the target address to evict the corresponding victim cache lines [71], [43], [46], [99], [64], [98], [24], [32], [71], [37], [91], [7], [11].
- Occupancy-based approaches [84] where the adversary observes an eviction of its own cache lines and uses this information to infer the size of the victim's working set.

B. *Recent Defenses and their Shortcomings*

Various defenses against side-channel attacks have been proposed, focusing on access-based and conflict-based attacks.

Side-channel Resilient Implementation. This aims at implementing algorithms, e.g. cryptographic algorithms, in a time-constant (thus side-channel-resilient) fashion [42], [8]. Time-constant algorithms vary between hardware platforms [19] and require considerable effort that is not generalizable and scalable for all software.

Attack Detection. Other approaches aim to detect attacks in progress by observing hardware performance counters (e.g., on cache miss rates) [18], [72] and killing the suspicious process. However, being based on heuristics, attacks can only be discovered with a certain probability and no guaranteed protection is provided. Moreover, some attacks have been shown to not cause an abnormal cache behavior [35].

Noisy Measurements. Another group of defenses aims to impede a successful attack by preventing the adversary from performing precise time measurements, e.g., by restricting the access to timers [71], [73], [66], by injecting noise into the system [93], [40] or deliberately slowing down the system clock [39], [67]. However, workarounds have been found to create timers [82] or to perform attacks without relying on timers [24]. Moreover, such defenses cannot protect TEE architectures since they assume a strong adversary that can compromise the OS kernel and circumvent such restrictions.

Cache-level Defenses. Other approaches tackle the side-channel problem directly where it originates, i.e., at the cache level. These defenses fall under one of two paradigms: 1.) randomized cache line mapping to make the attacks computationally impractical [89], [77], [78], [96], [87], [95], [63], [62] or 2.) cache partitioning to provide strict isolation [29], [50], [101], [61], [22], [33], [103], [47], [97], [57], [6], [94], [51], [95], [23]. We discuss the works most related to CHUNKED-CACHE in more detail in Section VII.

Randomization-based defenses cannot provide comprehensive future-proof security guarantees, e.g., advances in attack strategies and minimal eviction set construction techniques, besides alternative attack techniques have been shown to undermine such defenses [78], [12], [75], [74], [86]. Moreover, many rely on cryptographic primitives which have been shown vulnerable to cryptanalysis, while deploying more secure primitives would further degrade performance [10], [74].

Cache partitioning defenses provide strict resource isolation which allows to give solid security guarantees on side-channel protection. However, existing partitioning defenses suffer from high performance penalties, restrictive and inflexible cache utilization [95] and their inability to scale with a larger number of protection domains [94], [51], [33]. Several approaches do not directly cater for the use of shared libraries [29], [94], are architecture-specific [47], [97] or do not defend against occupancy-based attacks. Memory page coloring approaches [22], [50], [29] are impractical since they require invasive modifications of the memory management of commodity software and cannot sufficiently support Direct Memory Access (DMA). Most importantly, existing partitioning defenses to date apply their side-channel cache protection for the entire execution workload, impacting overall system performance, which is not even required in most scenarios.

To fundamentally address all these shortcomings, we propose a modified cache microarchitecture, which we call CHUNKED-CACHE, that provides strict, yet configurable partitioning across the mutually distrusting execution domains. For each domain, CHUNKED-CACHE carves out and isolates an exclusive cache share only as the domain requires. This effectively mitigates all interference across domains, thus, defending against even stealthy cache occupancy attacks unlike recent cache defenses, while activating side-channel resilience only for sensitive execution domains that require it. All other execution domains can freely utilize mainstream cache resources at the same performance or even improved performance than conventional non-secure cache sharing.

III. SYSTEM & ADVERSARY MODEL

In the following section, we describe our assumptions regarding the system and adversary model.

A. *System Model*

CHUNKED-CACHE targets computing systems which implement a TEE security architecture and contain a set-associative cache architecture. In the following, we first present our standard assumptions regarding the cache architecture, followed by our assumptions on the TEE security architecture which are aligned with existing academic [22], [57], [13], [6] and industry solutions [41], [45], [5].

Cache Architecture. In CHUNKED-CACHE, we assume a typical modern set-associative cache architecture with multiple cache levels, where some cache levels are core exclusive (typically L1 and L2) and others shared between multiple cores (L3), whereby the L3 can be a sliced cache, e.g., sliced Intel LLCs. While CHUNKED-CACHE can be deployed to provide partitioning for smaller L1 and L2 caches in principle, we assume, however, that core-exclusive caches are flushed at context switching (similar to most recent TEE architectures [6], [22], [57]), and thus, that CHUNKED-CACHE is deployed for the last-level L3 cache. Moreover, we assume that the cache controller can be configured via dedicated configuration registers, in line with typical platforms.

TEE Architecture. We assume that the computing systems which deploy CHUNKED-CACHE implement a TEE architecture. TEE architectures already have established mechanisms for protecting sensitive code in compartmentalized execution contexts called *enclaves* or **I**solated **D**omains (I-Domain), as we refer to them in this work. All non-sensitive code which does not require enhanced protection is consolidated in a **N**on-**I**solated **D**omain (NI-Domain). The domains are also each assigned a unique identifier (domain ID). The separation between the I-Domains and the NI-Domain is enforced by access control mechanisms already implemented in the TEE architectures, e.g., at the MMU in Intel SGX [41] or Sanctum [41], at the system bus in CURE [6] or by the Physical Memory Protection (PMP) unit in Keystone [57]. The access control mechanisms are either configured by microcode [41], [45] or by a small software component which consists only of a few thousand lines of code (to be formally verifiable) and which runs in the highest software privilege level of the system [22], [57], [13], [6], [5]. We refer to this component as a *trusted software component*. The trusted software component is also responsible for all other security-sensitive operations, e.g., assigning the domain IDs, and, in the case of CHUNKED-CACHE, configuring our novel protection mechanisms in the cache controller which we describe in detail in Section IV.

Although I-Domains are security-sensitive, they might still require to share data with another domain, e.g., to enable communication with the operating system. Thus, TEE architectures typically provide the possibility to mark parts of an I-Domain's memory as *shared*, whereby this information is again managed by the trusted software component. In many TEE architectures, e.g., TrustZone [5], CURE [6] or AMD SEV [45], security-relevant metadata, which is required to perform access control, is sent as part of every memory request. For CHUNKED-CACHE we assume the same, namely, that the domain ID of the domain issuing a memory access request and the information whether the requested memory address is *shared* or *non-shared*, are sent within the memory request.

B. Adversary Model

Since we focus on the deployment of CHUNKED-CACHE on systems with TEE architectures, we assume the same strong adversary model where the operating system kernel and hypervisor are untrusted [22], [57], [13], [6], [41], [45], [5].

With regard to cache side-channel attacks, we assume the adversary has access to the CHUNKED-CACHE specification and is able to mount access-based and conflict-based side-channel attacks, which are the most sophisticated and applicable cache attacks (cf. Section II-A), to leak information about a sensitive execution domain (I-Domain). Since the adversary is also able to control the OS kernel, we assume a worst-case scenario where an adversary can easily mount the described attacks, i.e., has knowledge about the CHUNKED-CACHE design and specs, and knows the virtual to physical address mapping of the victim domain. Moreover, the adversary can mount attacks from all privilege levels (except the highest privilege level that contains the trusted software component), has access to precise timing measurements and eviction instructions (e.g., `clflush`), can attack from the same CPU core executing the victim domain or a different core (cross-core), freely interrupt the victim domain and even keep the system noise to a minimum. In contrast to related work [89], [77], [78], [96], [87], we also consider the stealthier cache occupancy-based attacks (cf. Section II-A). Collision-based attacks [11], which exploit cache collisions at the victim caused by the victim's own cache utilization, are, aligned with related work, kept out of scope. Collision-based attacks have not been widely shown and are very specific to particular software implementations (e.g., table-based).

Apart from cache side-channel attacks, an adversary who compromises the OS kernel has full control over the memory management and thus, can easily map physical memory pages of a victim domain into its own memory. This allows an adversary to perform rogue cache accesses to sensitive data *directly* without the need of a cache side channel.

In line with related work [29], [94], [51], [61], [23], [95], [63], [89], [77], [78], [96], [87], we do not consider physical attacks on caches, e.g., physical side-channel attacks [53], fault injection attacks [9], and attacks that exploit hardware flaws [88], [48], [76]. We do not consider denial-of-service attacks from a security point of view. However, to avoid the performance impact on the OS, CHUNKED-CACHE ensures that a certain amount of cache resources are always available to the OS (described in Section IV). Based on our system model (Section III-A) , we assume that the adversary cannot compromise the trusted software component.

IV. CHUNKED-CACHE DESIGN

We first describe the high-level idea of CHUNKED-CACHE, a novel cache microarchitecture that provides flexible and on-demand assignment of cache resources to execution domains (Section IV-A). We follow with a detailed explanation of our design (Section IV-B) and the required cache tag store and cache controller modifications (Section IV-C).

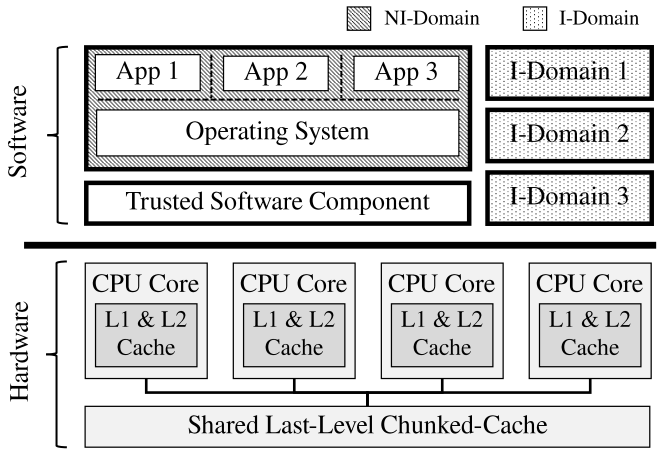


Fig. 1. Computing system with TEE architecture and CHUNKED-CACHE as the shared last-level cache.

A. High-Level Design

In Figure 1, we show how CHUNKED-CACHE is integrated as the last-level cache in a computing system which implements a TEE architecture, aligned with our system model detailed in Section III-A. Figure 2 steers the focus to the design of CHUNKED-CACHE itself and illustrates its architecture abstractly. As described in Section III-A, all TEE architectures provide built-in mechanisms to protect sensitive code in **I**solated **D**omains (I-Domains), whereas non-sensitive code is running in a **N**on-**I**solated **D**omain (NI-Domain).

Each active domain (NI-Domain and I-Domains) is uniquely identified by an ID: `DID`. The operating system (OS) and all workloads which do not require protection (and are combined in the NI-Domain) are assigned the `DID` 0 by default. Every I-Domain can request exclusive cache resources

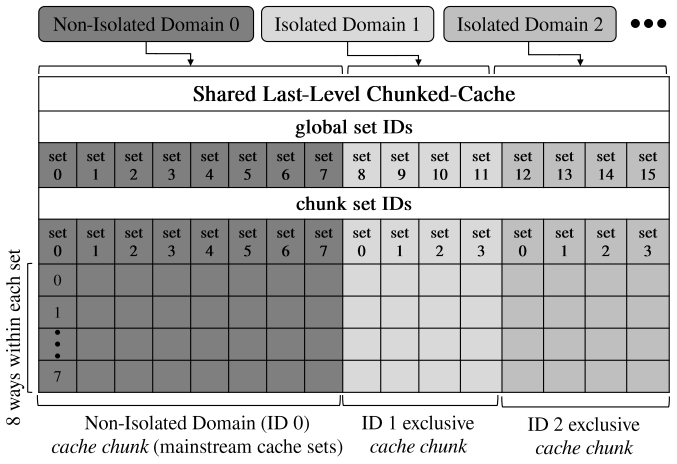

Non-Isolated Domain 0
Isolated Domain 1
Isolated Domain 2
Shared Last-Level Chunked-Cache
global set IDs
set 0 set 1 set 2 set 3 set 4 set 5 set 6 set 7 set 8 set 9 set 10 set 11 set 12 set 13 set 14 set 15
chunk set IDs
set 0 set 1 set 2 set 3 set 4 set 5 set 6 set 7 set 0 set 1 set 2 set 3 set 0 set 1 set 2 set 3
8 ways within each set
0
1
7
Non-Isolated Domain (ID 0) cache chunk (mainstream cache sets)
ID 1 exclusive cache chunk
ID 2 exclusive cache chunk

Fig. 2. CHUNKED-CACHE high-level design: each domain gets an exclusive *cache chunk* allocated on-demand.

of desirable capacity, forming the domain's exclusive *cache chunk*, that is only utilized by the owner domain. The NI-Domain utilizes the cache sets which are not exclusively allocated to I-Domains, which we call *mainstream* cache sets.

Each I-Domain requests its dedicated *cache chunk* consisting of the required number of cache sets, e.g., I-Domain 1 in Figure 2 requested 4 sets. Thus, at I-Domain 1 setup, 4 available (unallocated) sets are located in the cache (sets with global IDs 8-11 here) and allocated to I-Domain 1 such that they form its *cache chunk*. The allocated sets are mapped to I-Domain 1's chunk set IDs 0-3, and they are used to exclusively cache all and only memory accesses issued by I-Domain 1. Enabling each I-Domain to request its desired *cache chunk* capacity exclusively provides strict partitioning and completely isolates its cache utilization on-demand. Besides enabling selective cache-based side-channel resilience, this also allows that each I-Domain acquires the performance that corresponds to the cache capacity it has requested, without any competition from other workload. In contrast to partitioning schemes [61], [57], [6], [94], [51] that provide each domain with only 1 or 2 ways within each set of the full cache structure, CHUNKED-CACHE also partitions the cache but more efficiently. CHUNKED-CACHE carves out a full *cache chunk* (with all its ways per set) of configurable capacity for the I-Domain and configures all its memory accesses to be mapped to the *cache chunk*, thus promising maximum and unshared utilization of the allocated *cache chunk*. We show in Section VI that CHUNKED-CACHE provides better performance and enhanced scalability than partitioning schemes.

By allowing each I-Domain a custom and configurable *cache chunk* capacity on-demand, in contrast to fixed allocation, CHUNKED-CACHE enables an adaptive security-performance trade-off in the cache microarchitecture. On one hand, non-sensitive workload can be allowed to freely utilize the shared mainstream cache resources. On the other hand, if side-channel resilience is a concern, a *cache chunk* with default capacity can be allocated to each I-Domain without any further intervention from the developer. Only if the developer requires to further optimize the performance of the workload in a particular I-Domain, then the *cache chunk* capacity (its number of sets) can be accordingly calibrated, i.e., assigning an I-Domain more cache resources if affordable/available.

B. *Design Details of* CHUNKED-CACHE

In the following, we discuss the key design goals and challenges of CHUNKED-CACHE, and the mechanisms we propose to achieve them.

Configurable Per-Domain Isolation Modes. One of our key design goals for CHUNKED-CACHE is to support configurable cache isolation modes that provide different security guarantees, thus catering for different use cases and their requirements. In line with the design paradigm of TEEs, it is not reasonable to assume that all workloads require cache isolation and side-channel resilience. Thus, in CHUNKED-CACHE, we provide 2 different ISOLATION MODES that each I-Domain can selectively configure for the workload it protects: 1.) MAINSTREAM-CACHE MODE: where cache isolation and side-channel resilience is not a security requirement, and thus, the I-Domain can utilize the mainstream cache. However, the cached I-Domain data must still be protected from malicious OS accesses. 2.) EXCLUSIVE-CACHE MODE: where cache isolation is required since side-channel resilience is a security requirement and thus, an exclusive *cache chunk* is required by this I-Domain. The latter mode is configured for I-Domain 1 and I-Domain 2 shown in Figure 2. In addition to the ISOLATION MODE, the I-Domain can also configure its SHARED MEMORY settings, i.e., if it requires to share memory regions (and thus cache lines) with the OS, e.g., when using OS services. To cache shared memory, the mainstream cache that the OS uses is utilized. Typically, the developer of the workload decides which ISOLATION MODE an I-Domain uses and identifies which memory regions need to be shared, which is on par with the requirement in TEE architectures where the developer must identify the security-sensitive parts of the overall workload [41], [5], [22]. If a developer is not sure whether cache side-channel attacks are a threat, the EXCLUSIVE-CACHE MODE should be selected out of caution. At setup, an I-Domain configures: 1.) the desired ISOLATION MODE for its cache utilization and 2.) its SHARED MEMORY regions if required. This metadata is securely configured by the trusted component (as shown in Figure 1). The ISOLATION MODE is communicated to the cache controller at domain setup, whereas the SHARED MEMORY information is transmitted at every memory request, aligned with our assumed system model (Section III-A).

Mainstream Cache vs. Shared Memory Support. When an I-Domain is in MAINSTREAM-CACHE MODE, it uses the mainstream cache sets also used by the OS (`DID` 0). To prevent a malicious OS from mapping the memory of an I-Domain in its own memory space and accessing it directly in the cache, CHUNKED-CACHE requires that cache lines are tagged with the domain ID `DID`. The hardware mechanisms integrated into the CHUNKED-CACHE controller enforce this tagging when caching the data, and that only the owner domain which cached the data can access it. Being hardware managed, the OS has no means to modify the `DID` stored in the cache lines.

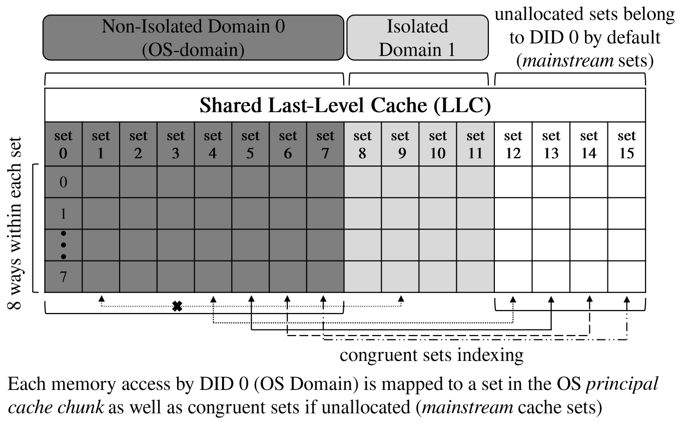
When an I-Domain is also sharing memory with the OS, the corresponding cache lines for the defined SHARED MEMORY regions are cached in the mainstream cache sets, and are to be accessed by both the owner domain and the OS. To support that, cache lines need to be tagged with an additional `SHARED` flag that indicates whether the cache line is shared with the OS. For typical TEE architectures, the developer of the workload protected in the I-Domain configures which of its memory regions are to be shared.

Exclusive-Cache Mode Chunk Set Indexing. The *index* bits of a memory address are used to locate the cache set to which it is mapped. In a conventional cache, the number of index bits is fixed and depends on the number of sets the cache supports. However, for Chunked-Cache to support *cache chunks* of different sizes for different domains, *configurable set indexing* is required.

When an I-Domain is in Exclusive-Cache Mode and requests a number of cache sets for its *cache chunk*, the number of set index bits that will be used to map its memory lines has to be computed individually for this domain. Therefore, the cache controller keeps track of the global IDs of sets which constitute the *cache chunk* (Figure 2), and the index bits for each domain. When a memory access is issued by a domain, this metadata is looked up, and the pertinent *cache chunk* sets correctly indexed. Moreover, when an I-Domain is torn down and its sets are de-allocated, the relevant metadata needs to be updated accordingly, besides flushing and invalidating the cache lines. Chunked-Cache also enables support for dynamic cache allocation, i.e., allocating additional cache sets to an I-Domain's *cache chunk* at runtime and reconfiguring the index bits accordingly. In Section IV-C, we describe how the cache microarchitecture and controller are modified to enable this configurability efficiently.

NI-Domain Chunk Set Indexing. Another design challenge in Chunked-Cache is managing the sets allocated to the OS, which represents the NI-Domain with `DID` 0, such that both flexibility as well as maximum utilization (as in an unmodified insecure cache architecture) are preserved. At bootup, when no domains are set up yet besides the OS, the OS should ideally be able to utilize all the available cache capacity, i.e., all cache sets are allocated to the OS by default. We refer to these as the *mainstream* cache sets. Then, once domains are set up and request exclusive cache sets, these get "torn away" from the OS's cache and are allocated to the domains. This would, however, incur an impractical performance degradation for the OS since every time some of the OS's cache resources are allocated to another domain, its own capacity is changed, and so would its set indexing. This renders all memory lines already cached by the OS inaccessible unless complicated remapping is performed. Essentially, the OS would need to cache these memory addresses once again, thus suffering a high number of cold misses every time a new domain is set up and subjecting the OS to an unreasonably high performance overhead.

To avoid this performance penalty on the OS, the OS is allocated a fixed (sufficiently large) number of the cache sets in Chunked-Cache which remain always dedicated to the OS, while still allowing it to utilize the other cache sets so long as they remain unallocated. We demonstrate this in Figure 3 where the OS is always allocated a fixed number of 8 sets (0-7) which form its principal *cache chunk*. Since the 8 sets are always available for the OS, the memory address indexing and the number of index bits do not change at runtime. In other words, no OS memory lines cached in this principal *cache chunk* must ever be flushed out when any other domain requests to allocate additional cache sets, since the OS *cache chunk* sets are never torn away from the OS. However, the OS can still utilize unallocated sets (sets 12-15) in parallel until they get allocated to another domain, thus also guaranteeing maximum utilization of the available cache resources. This works by indexing cache sets in parallel which are congruent to the set to which a memory address is mapped. In Figure 3, 3 index bits are required to map a memory address to the correct set for a *cache chunk* of size 8 sets. Thus, if the index bits, e.g., map to set 4, then set 12 can also be utilized by the OS (set ID + OS *cache chunk* size) to cache that memory line. The same applies for memory lines that are mapped to sets 5, 6 and 7; they also map to sets 13, 14 and 15, respectively. However, memory lines mapped to sets 0-3 cannot utilize the congruent sets 8-11 because these are already allocated to I-Domain 1.



Non-Isolated Domain 0 (OS-domain)
Isolated Domain 1
unallocated sets belong to DID 0 by default (*mainstream* sets)
Shared Last-Level Cache (LLC)
set 0
set 1
set 2
set 3
set 4
set 5
set 6
set 7
set 8
set 9
set 10
set 11
set 12
set 13
set 14
set 15
8 ways within each set
0
1
7
congruent sets indexing
Each memory access by DID 0 (OS Domain) is mapped to a set in the OS *principal cache chunk* as well as congruent sets if unallocated (*mainstream* cache sets)

Fig. 3. Chunked-Cache OS-specific chunk set indexing.

C. Cache Tag Store & Cache Controller

Cache lines need to be additionally tagged with the domain ID (`DID`) bits as well as a 1-bit `SHARED` flag bit to enforce access control and moderate sharing with the NI-Domain. For instance, to support 16 parallel active domains, we require to extend the cache tag store with 4 bits to represent the `DID`. We emphasize that the Chunked-Cache design does not limit the number of parallel domains to 16; a larger number is possible but increases the hardware overhead of Chunked-Cache (but only linearly). Moreover, the number of domains only limits how many domains can be simultaneously active on the system. It does not limit how many applications can be protected in I-Domains on the system in general.

To support the configurable set indexing, the allocation/de-allocation of cache sets to different I-Domains and to differentiate between OS (NI-Domain) cache accesses vs. I-Domain accesses, 2 table structures are required by the Chunked-Cache controller which are shown in Figure 5. The Cache Set Status Table (CST) is a 1-bit vector that is indexed by the global set ID (SID) and that stores the status of each set, i.e., whether it is allocated to a domain. The CST is used to query the status of a set when searching for free cache sets to allocate to an I-Domain.

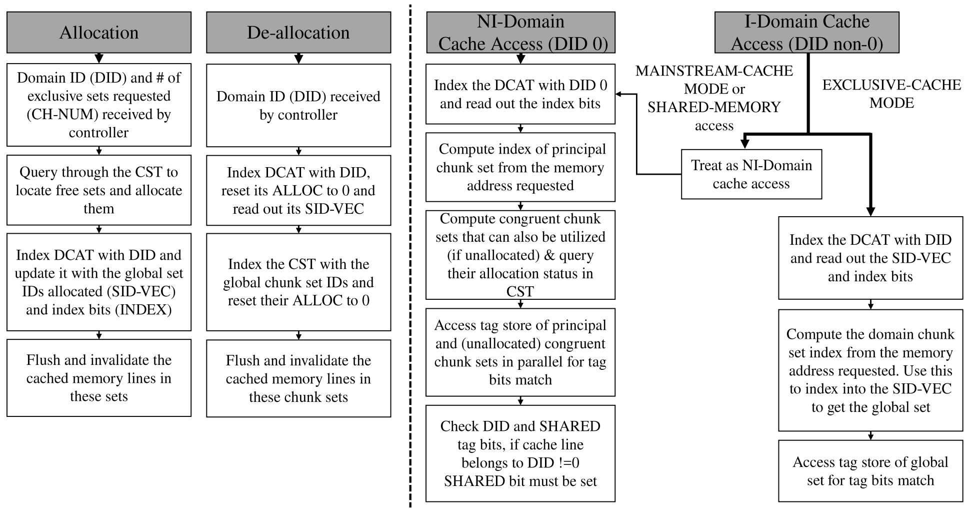

Allocation
Domain ID (DID) and # of exclusive sets requested (CH-NUM) received by controller
Query through the CST to locate free sets and allocate them
Index DCAT with DID and update it with the global set IDs allocated (SID-VEC) and index bits (INDEX)
Flush and invalidate the cached memory lines in these sets
De-allocation
Domain ID (DID) received by controller
Index DCAT with DID, reset its ALLOC to 0 and read out its SID-VEC
Index the CST with the global chunk set IDs and reset their ALLOC to 0
Flush and invalidate the cached memory lines in these chunk sets
NI-Domain Cache Access (DID 0)
Index the DCAT with DID 0 and read out the index bits
Compute index of principal chunk set from the memory address requested
Compute congruent chunk sets that can also be utilized (if unallocated) & query their allocation status in CST
Access tag store of principal and (unallocated) congruent chunk sets in parallel for tag bits match
Check DID and SHARED tag bits, if cache line belongs to DID !=0 SHARED bit must be set
I-Domain Cache Access (DID non-0)
MAINSTREAM-CACHE MODE or SHARED-MEMORY access
EXCLUSIVE-CACHE MODE
Treat as NI-Domain cache access
Index the DCAT with DID and read out the SID-VEC and index bits
Compute the domain chunk set index from the memory address requested. Use this to index into the SID-VEC to get the global set
Access tag store of global set for tag bits match

Fig. 4. CHUNKED-CACHE controller operations for cache chunk allocation, de-allocation and access control.

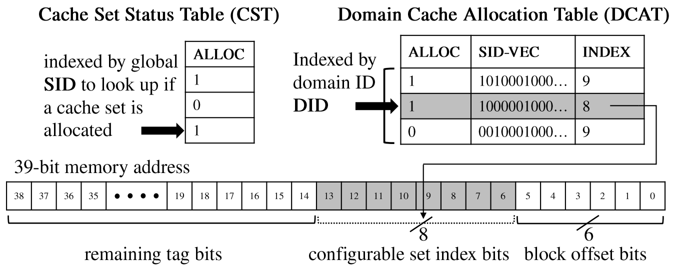


Fig. 5. CHUNKED-CACHE table structures.

The DOMAIN CACHE ALLOCATION TABLE (DCAT) is indexed by the domain ID DID. It maintains whether this domain is configured by the cache controller (ALLOC), a vector of the global set IDs that form its *cache chunk* (SID-VEC), and the corresponding number of index bits (INDEX) required to map a memory line to the correct set (log_2(number of sets in the *cache chunk*)), as shown in Figure 5.

We describe next how the CHUNKED-CACHE controller performs these cache management operations, i.e., allocation, de-allocation and access control and represent this in Figure 4. The description in Figure 4 only represents the sequence of operations for understanding, but does not reflect the temporal nature of the operations, i.e., whether they occur sequentially or in parallel.

Cache Allocation & De-allocation. When an I-Domain requests to allocate exclusive cache sets, this request (DID, the number of sets (CH-NUM) requested, and the corresponding number of INDEX bits ($\log_2$ CH-NUM) is securely communicated from the trusted component to the cache controller via configuration registers of the cache controller (Section III-A). The DID is looked up in the DCAT to check if it is already allocated and that the maximum sets number allowed per I-Domain is not exceeded. The maximum/minimum limits for I-Domains are configured by the trusted software component, while ensuring that each I-Domain is always assigned at least a minimum *cache chunk* size.

The CST is queried to locate free sets and to allocate them to the I-Domain by flipping the ALLOC bit, until CH-NUM sets are allocated. If CST runs out of free sets, this is communicated back to the trusted component to modify the cache request. Next, the DCAT is indexed with the DID and its metadata updated by updating the INDEX bits and the SID-VEC with the global IDs of the allocated sets.

If a domain requests to de-allocate its cache sets, DCAT is indexed with DID, ALLOC reset and the SID-VEC read out. Next, the CST is indexed with each set ID in SID-VEC and de-allocated. For both allocation and de-allocation, the cached memory lines in the relevant sets are invalidated and flushed (if dirty) to remove potentially malicious data in the allocation case and prevent information leakage in the de-allocation case.

The number of cache sets which are always assigned to the NI-Domain are hardwired, since the circuitry for the parallel tag lookup (described below) must be hardwired.

Cache Access Management. The DID of an incoming cache access request indicates whether it is an access by the NI-Domain (OS domain with DID 0) or an I-Domain. If it is an OS access, then the index bits are fixed, since its number of cache sets are hardwired (no need to look its INDEX bits up in the DCAT). The OS domain is assigned the least significant cache sets by default, thus the SID-VEC is also not needed. The correct set index in the principal chunk is computed from the memory address in the request. Because it is an OS access, congruent cache sets that are not allocated can also be utilized (see Section IV-A). Thus, they are also computed and their ALLOC status queried in the CST to locate the unallocated sets. The tag store of the ways in the principal as well as the congruent sets are looked up in parallel to locate a tag bit match (cache hit), thus, neither impacting performance nor routing delay especially since a large number of principal sets are usually allocated to the NI-Domain which minimizes the number of congruent sets that are looked up in parallel (1 or 2 more sets). The DID and SHARED tag bits are also checked in parallel. If the cache line belongs to a non-zero DID (I-Domain), the SHARED tag bit should be 1 to allow the OS to access it.

For an I-Domain (non-zero `DID`), if access is requested to a SHARED MEMORY region or if the I-Domain is in MAINSTREAM-CACHE MODE, then the access is treated by the controller as a NI-Domain access where the mainstream and congruent cache sets are accessed. However, at the tag comparison, the issuing `DID` is checked against the cache line `DID` to verify that only the owner domain accesses it. If the access is performed in EXCLUSIVE-CACHE MODE, the exclusive *cache chunk* of the domain is accessed. The DCAT is indexed with the `DID` and the SID-VEC and INDEX bits are read out. The chunk set index is computed and used to index into the SID-VEC to map to the correct global set ID. Then, the tag store is accessed for a tag bits comparison.

CHUNKED-CACHE's design is independent from the implemented cache replacement policy and thus, does not require additional modifications to it. On every cache miss experienced by an I-Domain in EXCLUSIVE-CACHE MODE, a cache line in the corresponding set in the domain's exclusive *cache chunk* is selected for eviction. On cache misses by an I-Domain in MAINSTREAM-CACHE MODE or when accessing SHARED MEMORY, and for all misses by the NI-Domain, a cache line in the corresponding set from the mainstream cache is selected.

V. SECURITY CONSIDERATIONS

In this section, we discuss how CHUNKED-CACHE protects from the adversary described in Section III-B. One key aspect of CHUNKED-CACHE is that its protection capabilities rely on a strict partitioning of cache resources. Thus, in contrast to related work, which rely on probabilistic defenses (e.g., randomized cache line mappings [89], [77], [78], [96], [87]), CHUNKED-CACHE provides certainty that the attacker cannot infer the cache accesses of a victim, if the partitioning is correctly implemented. The main security goals of CHUNKED-CACHE are to prevent an adversary from accessing (read/write) data in the exclusive cache chunk of an I-Domain and to prevent eviction interference between the adversary and victim domain. In the following, we show how CHUNKED-CACHE achieves these goals with strict cache partitioning and we discuss why CHUNKED-CACHE's security guarantees even hold in the event of a strong adversary that compromised the operating system kernel. Besides these security considerations, we verified the correctness of our implemented CHUNKED-CACHE prototype by explicitly issuing memory requests which try to read, write and evict cached data of I-Domains.

Strict Partitioning of I-Domain Cache Chunks. As described in Section IV, the trusted software component communicates the number of chunk sets which should be assigned to an I-Domain to the CHUNKED-CACHE cache controller which configures the DCAT and verifies that each cache chunk set is only assigned to a single I-Domain. At every cache memory access, the cache controller uses the domain ID to index the DCAT and to retrieve the list of assigned sets (SID-VEC). Since the assignment of domain IDs and configuration of the DCAT can only be performed by the trusted software component, the indexing logic of the cache controller will never return a cache set which does not belong to the issuer of the memory request. Thus, an adversary is never able to read an I-Domain's exclusive sets (*cache chunk*), write to them or evict them. As a result, CHUNKED-CACHE protects from access-based attacks, which require the adversary to flush memory out of the victim's sets, and conflict-based attacks, which require to fill the victim's sets and thus, evicting its cache lines. Moreover, CHUNKED-CACHE's strict cache resource separation prevents an adversary from observing evictions of its own sets caused by the victim, which protects from occupancy-based attacks, and also strictly prevents the sharing of replacement policy metadata, which has been shown exploitable [51]. In general, the adversary can only infer how many cache sets are assigned to an I-Domain but cannot infer which sets (and therefore which memory addresses) are accessed at which point in time. As described in Section III-B, collision-based attacks are not considered. Defending against them architecturally requires locking the victim cache lines. CHUNKED-CACHE could be extended to integrate this, though mitigating an attack which is very specific to particular software implementations and is not widely shown does not justify the resulting large performance overhead.

CHUNKED-CACHE allows for a dynamic assignment of cache sets to I-Domains. Whenever the *cache chunk* capacity of an I-Domain is modified, all assigned chunk sets are invalidated. This prevents leakage of sensitive I-Domain data when chunk sets are reassigned to another execution domain, and prevents an adversary from injecting malicious data into a set, when additional sets are assigned to an I-Domain. The invalidation is however only required for the I-Domain whose *cache chunk* is resized; all other I-Domains do not need to be modified and thus, their cache lines do not need to be flushed. The same applies when the *cache chunk* for an I-Domain is completely de-allocated. An adversary could also try to trick an I-Domain into storing sensitive data in a mainstream cache line that is accessible for the adversary (`SHARED` flag bit set). CHUNKED-CACHE prevents this by checking the metadata on every memory request of an I-Domain to verify that the memory region was indeed configured as *shared*.

Protecting from Compromised NI-Domain. As described in Section III-B, in the adversary model of TEE architectures, the OS (and therefore the NI-Domain) is not trusted, allowing an adversary to map physical memory pages of a victim I-Domain to its own memory space and to directly access it in the cache. If an I-Domain (represented by an enclave) demands side-channel protection (EXCLUSIVE-CACHE MODE), all data is cached in the exclusive *cache chunk* and thus, not accessible for the adversary. However, if an I-Domain is not concerned about cache side channels (MAINSTREAM-CACHE MODE), the data is cached in the shared mainstream sets and thus, must still be protected from malicious direct accesses. CHUNKED-CACHE prevents those attacks with the domain ID tag which is added to every cache line. On every cache write, the domain ID tag is set to the ID of the write request issuer. Subsequently, on every read request, the ID of the issuer is compared to the stored ID and the request only permitted if both IDs match. Evictions are permitted for every domain to achieve a perfect utilization of the shared cache sets. This is, however, not a security concern since an I-Domain's data will only be cached in the shared sets if the I-Domain is in MAINSTREAM-CACHE MODE or if the data is explicitly shared with the NI-Domain.

VI. Implementation & Evaluation

To evaluate Chunked-Cache with respect to its hardware footprint, power consumption overheads, and performance impact, we implemented our design in hardware and on an architectural cycle-accurate simulator.

Methodology. We implemented a hardware RTL model of Chunked-Cache to extend an open-source RISC-V processor and synthesized it to evaluate the storage and logic overhead incurred. We use our hardware implementation to extract the additional cycle latencies incurred by Chunked-Cache due to individual cache management and access operations. Then, to evaluate the performance impact of Chunked-Cache on large mixed workloads, we extend an architectural cycle-accurate simulator, the gem5 simulator, with Chunked-Cache and configure it to model a multi-core architecture with a 3-level cache hierarchy which matches our system assumptions (Section III-A). We incorporate the cycle latencies derived from our hardware implementation into our gem5 setup and use it to collect performance measurements on the standard SPEC CPU2017 [20] benchmarks suite (aligned with related work [77], [78], [96], [87]) to evaluate the overall performance impact of Chunked-Cache. Complementary to the compute-intensive SPEC benchmarks, we also evaluate Chunked-Cache on the I/O-intensive web server `nginx`. In order to achieve the most realistic results, we conduct our experiments in the full-system simulation mode of gem5 which simulates the user- and kernel-space software and also I/O devices.

We describe next our hardware implementation (Section VI-A), performance evaluation (Section VI-B), and our hardware an power overhead evaluation (Section VI-C).

A. Hardware Implementation

In our hardware model, we extended the cache tag store with a 4-bit `DID` and a 1-bit `SHARED` bit to tag the owner domain of each cache line and whether it is shared with the NI-Domain (OS), respectively. We also extended the cache controller with the table structures shown in Figure 5. To track the status of the 16,384 sets of a 16 MB LLC with 16-ways, the CST is implemented as a 16,384-bit register that is indexed by the set ID to read out the corresponding 1-bit ALLOC flag. To support set allocation for 16 domains in parallel, the DCAT is implemented as a 16-row `DID`-indexed vector structure. We decided for 16 parallel domains in our hardware implementation since this is also the maximum number of enclaves supported by multiple TEE architectures in parallel [57], [6]. We define for our implementation that the maximum number of sets that can be allocated to any domain is 8,192 sets. Thus, we reserve 4 bits to represent the set INDEX bits number (to index into one of 8,192 sets), 114,688 bits (8,192 sets $\times$ 14 bits to represent each set's global ID) for the SID-VEC, and 1 bit ALLOC flag per domain. We discuss the storage overheads incurred by the tables in Section VI-C.

We implement the control finite-state-machines (FSMs) that receive cache allocation and de-allocation requests and perform the necessary management. For allocation, the FSM controls cycling through the sets sequentially to allocate free ones to the requesting I-Domain, updating their status in the CST and updating the corresponding domain status in the DCAT. For de-allocation, another FSM controls that the SID-VEC of the pertinent I-Domain is read from the DCAT, its ALLOC flag reset, and then, all sets of that I-Domain de-allocated (by sequentially indexing through the CST with the respective set IDs from the SID-VEC). Both allocation and de-allocation occur in powers-of-2 set numbers in our prototype. This is only an implementation decision in our prototype to minimize the logic complexity and overhead.

The cache access mechanisms are extended to include the DCAT lookup required for Chunked-Cache to identify which global set IDs belong to the issuing domain and to map the access to the correct set prior to tag lookup. Additionally, for NI-Domain accesses, after mapping to the correct set ID, concurrent sets are computed and looked up in the CST in parallel to identify which ones are unallocated.

B. Performance Evaluation

In this section, we first describe the latencies from our RTL model which we incorporate into our gem5 implementation. Next, we provide an evaluation of Chunked-Cache's performance impact using the gem5 implementation.

Cycle Latencies. As described in Section IV, Chunked-Cache introduces a new indexing policy. For I-Domain memory requests in Exclusive-Cache Mode, a lookup in the DCAT is required. For requests in Mainstream-Cache Mode and all NI-Domain (OS) memory requests, the mainstream sets must be looked up. The comparison of the stored `DID` with the requester `DID` is done in parallel with the address tag comparison and thus, does not introduce additional latency. For I-Domain requests in Exclusive-Cache Mode, we measure an additional latency of 1 cycle and for NI-Domain requests and I-Domain requests in Mainstream-Cache Mode of an additional 2 cycles. For the access latencies of modern LLCs on multi-core systems, we estimate a baseline of 80 cycles in line with vendor multi-core processors [2].

Whenever an I-Domain gets sets allocated, unallocated sets are looked up and the DCAT updated. At de-allocation, sets of the I-Domain must be invalidated (and possibly flushed) and the CST and DCAT updated. For allocation, the overall latency incurred is variable and is a function of: 1.) how many sets CH-NUM are requested for allocation, and 2.) how many sets have to be looked up in the CST. At the worst case, this incurs a latency of 16,384 cycles and at the best case, CH-NUM cycles. An additional 1 cycle is incurred to update the DCAT subsequently. The INDEX is computed and communicated already by the trusted component in the allocation request, thus it does not contribute additional latency.

For de-allocation, we measure an overall latency of CH-NUM + 2 cycles, where 1 cycle is required to look up the DCAT, and another cycle to update it, followed by CH-NUM cycles to de-allocate each set in the CST. At worst case, a latency of 8,194 cycles is incurred (assuming a maximum of 8,192 sets per domain). However, de-allocating the sets in CST is done in parallel to invalidating (and possibly flushing if dirty) the respective cache lines.

We emphasize that allocating new sets to any I-Domain does not require invalidating or flushing any other sets of

Parameters	L1 (I&D)	L2	L3 (gem5)	L3 (CHUNKED-CACHE)
size	64 KB & 32 KB	512 KB	16 MB	16 MB
# of sets	128 & 64	512	16,384	16,384
associativity	8-way	16-way	16-way	16-way
access latency (in cycles)	4	14	80	81 / 82

TABLE I. CACHE CONFIGURATION ON OUR GEM5 EVALUATION SETUP WITH AN INCLUSIVE 3-LEVEL CACHE HIERARCHY.

the NI-Domain or other I-Domains which would require re-caching them. This is one key design goal of CHUNKED-CACHE since it eliminates this performance overhead on other domains, particularly the NI-Domain. The allocation of sets either happens only once during the I-Domain setup or occasionally when the number of assigned sets is modified at run-time which requires a context switch out of the I-Domain. The CHUNKED-CACHE allocation/de-allocation overheads induced remain negligible when compared with the general overheads of TEE architectures [22], [13], [57], [6]. Therefore, we do not invest in increased logic complexity to optimize the cycle overheads incurred for allocation and de-allocation, since they are not in the critical path, i.e., LLC accesses.

Mixed-Workload Cycle-Accurate Evaluation. We implement CHUNKED-CACHE on the cycle-accurate gem5 simulator and construct a multi-core system which resembles a modern computing system with an inclusive 3-level cache hierarchy. Each core has access to a core-exclusive L1 and L2 cache, and an L3 LLC shared among all cores. For the L1 and L2, we use the unmodified cache implementation provided by gem5, whereas we use our CHUNKED-CACHE implementation for the L3 cache. The configuration parameters of each cache level are shown in Table I. We derive realistic values for the cache sizes, number of cache sets, associativity and access latency in line with modern caches. For the CHUNKED-CACHE L3 cache, we add our induced latencies collected from our hardware implementation. Constructing a gem5-based multi-core system with 3-level cache hierarchy in full-system simulation mode to collect representative cycle-accurate traces for large workloads involved significant engineering challenges, as also evident by recent works that rely on trace-based simulators for their evaluation with SPEC workloads [77], [78], [96], [87].

We measure the performance impact of CHUNKED-CACHE on real-world workloads by using the standard SPEC CPU2017 benchmarks with both the SPECspeed 2017 Integer and SPECspeed 2017 Floating Point suites which represent a wide range of compute-intensive applications such as compilers, video compression, machine learning or modeling tasks. Since running all of the benchmarks on our full-system cycle-accurate gem5-based simulation setup would be very costly in terms of memory and time, we selected benchmarks from the different application domains and different working set sizes, guided by this memory-centric characterization of the SPEC CPU2017 benchmarks [85]. Moreover, to also cover I/O-intensive workloads, we evaluate the impact of CHUNKED-CACHE on the widely used web server `nginx`. We run our experiments for 1 trillion instructions before we start to collect measurements, in order to boot the system, start the benchmarks and collect more representative metrics. We run all our experiments for a total of 1 billion instructions in the

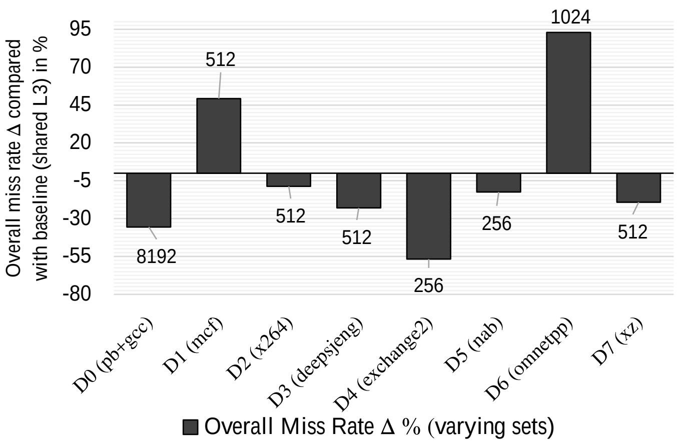


Fig. 6. Cache miss rate impact of CHUNKED-CACHE for SPEC benchmarks on a 8-domain setup; compared to a shared L3 cache.

full-system mode of gem5 and collect statistics to compute the Cycles Per Instruction (CPI) metric, in order to capture the additional latency effect, and the L3 cache miss rates for the reduced cache capacity effects. If not stated otherwise for single experiments, the miss rates are calculated as the geometric mean over the instruction and data miss rates of the page table walker and core. We compare CHUNKED-CACHE to 1.) a baseline system with an unmodified insecure L3 cache and to 2.) an L3 cache which implements a way-based partitioning scheme in which cache ways are assigned to I-Domains as provided, e.g., by CATalyst [61] which uses Intel CAT [49], SecDCP [94], DAWG [51], Keystone [57] or CURE [6]. We evaluate CHUNKED-CACHE with a set of experiments which investigate different computing scenarios. First, we show how CHUNKED-CACHE's partitioning influences the performance of mixed workloads when encapsulated in I-Domains (in EXCLUSIVE-CACHE MODE). Then, we evaluate CHUNKED-CACHE's impact on the NI-Domain (OS-domain) and compare against way-based partitioned cache schemes. We conclude our evaluation with a set of experiments which show the scalability of CHUNKED-CACHE. In general, when comparing to the baseline (unpartitioned L3 cache shared by the same workload), our experiments show a negative effect of CHUNKED-CACHE on the performance of a benchmark when only a small *cache chunk* size is assigned to it. However, when increasing the *cache chunk* size this effect vanishes. At some point, depending on the specific characteristics of a benchmark, the exclusive *cache chunk* assigned by CHUNKED-CACHE leads to a positive effect on its performance as we show in the following experiments. This gives the developer some degree of freedom to calibrate the performance of the workload by distributing the cache resources accordingly, e.g., to optimize the performance of a particular benchmark if desired given that the cache resources are available/affordable. All experiments were conducted on an x86 platform equipped with an Intel Xeon Silver 4215 CPU (2.50 GHz) and 186 GB RAM.

I-Domain Performance Impact. In the first set of experiments, we evaluate the performance impact CHUNKED-CACHE has on mixed workloads when protected in I-Domains in EXCLUSIVE-CACHE MODE. We run 7 randomly selected SPEC benchmarks in I-Domains and show our re-

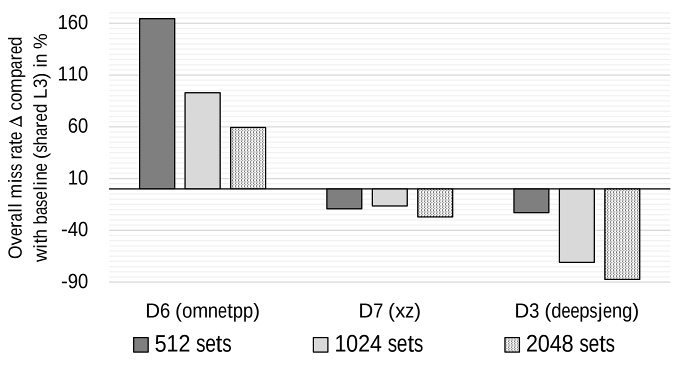


Fig. 7. Cache miss rate impact of CHUNKED-CACHE for SPEC CPU2017 benchmarks (varying sets); compared to a shared L3 cache.

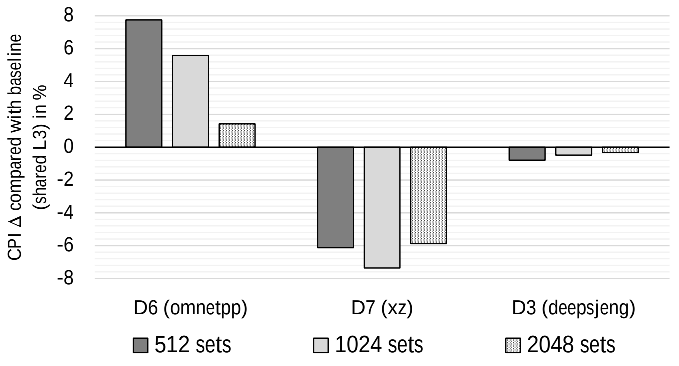


Fig. 8. CPI impact of CHUNKED-CACHE for SPEC CPU2017 benchmarks (increasing sets); compared to a shared L3 cache.

sults in Figure 6. The NI-Domain (D0) runs Linux (kernel version 4.19.83) and 2 benchmarks with large working sets (`600.perlbench_s` and `602.gcc_s`). In this experiment, we assign 8,192 sets to the NI-Domain and a varying number of sets to each I-Domain as indicated in the plot. We chose the number of sets by briefly analyzing the working set size of the benchmark running in each I-Domain, and assigning bigger working sets to more cache sets. This is only required when optimizing for performance, otherwise a default number of sets can be assigned to each benchmark. We observe in the experiment that the overall miss rate significantly decreases for most benchmarks when compared to sharing the L3 cache. This shows that the assignment of a smaller but exclusive cache portion can even reduce the cache miss rates of a workload. Moreover, our results indicate that the number of cache sets required to reduce or completely avoid the impact of CHUNKED-CACHE heavily depends on the characteristics of the workload. In our experiment, the benchmarks `605.mcf_s` and `620.omnetpp_s` would require more cache sets than the assigned 512 and 1024 sets to avoid an impact on the cache miss rates. We investigate this in another experiment where we customize the number of sets allocated to an I-Domain for some of the benchmarks and show how the miss rate decreases significantly when increasing the chunk size (Figure 7). In another experiment (Figure 8), we show how the varying chunk sizes also influence the CPI values. As for the miss rates, the CPI decreases in general. We observe, however, some outliers with the CPI metrics collected, owing to the complexity of a full-system multi-core simulation on gem5 which also includes unpredictable kernel runtime behavior into the statistics.

Additionally, to evaluate the impact of CHUNKED-CACHE on I/O-intensive workloads, we conduct experiments in which we run the `nginx` web server in one I-Domain and the HTTP benchmarking tool `wrk` in another I-Domain, whereas we keep the NI-Domain unmodified. We then use `wrk` to send HTTP requests to the web server using 12 threads and 400 open connections. In Figure 9, the miss rate impact of CHUNKED-CACHE on `nginx` and `wrk` is shown when increasing the number of sets from 128 to 2048. The results show, in line with our results on SPEC, how the increase of cache sets leads to a decrease in the overall miss rate. The decrease is already noticeable for a relatively small number of sets since the exclusive assignment of the cache sets prevents `nginx` and `wrk` from evicting the sets from one another.

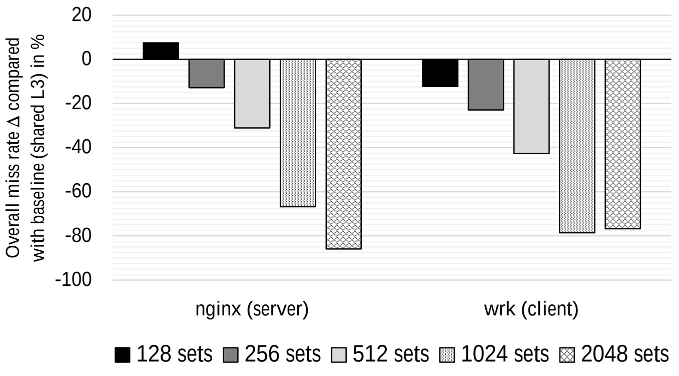


Fig. 9. Cache miss rate impact of CHUNKED-CACHE for `nginx` and `wrk` (increasing sets); compared to a shared L3 cache.

NI-Domain Performance Impact. In the second set of experiments, we focus on the performance impact of CHUNKED-CACHE on workloads executing in the NI-Domain. We again run mixed workloads from the SPEC benchmarks in I-Domains, while running Linux and the 2 memory-intensive benchmarks `600.perlbench_s` and `602.gcc_s` in the NI-Domain. In Figure 10, we vary the number of sets allocated to the NI-Domain from 2,084 to 8,192 while keeping the sets for the other domains unchanged. For these experiments, we show all 4 miss rate metrics over which we average in the other experiments, the data and instruction miss rates of the page table walker (DTB MR and ITB MR, respectively), and the data and instruction miss rates of the core (Data MR and Instr. MR, respectively). While in general, all miss rates and CPI metrics decrease compared to the baseline, we only observe a slight improvement when increasing the chunk size from 2,084 to 4,096 and 8,192 sets. This is because even when the number of statically allocated sets to the NI-Domain is rather small, the unallocated sets in the system (mainstream sets) remain available for the NI-Domain. Thus, performance is not significantly impacted for the NI-Domain and maximum utilization of the available resources (as in an unmodified insecure cache architecture) is preserved which was one of the key design goals of CHUNKED-CACHE.

To investigate this, we run experiments (same setup) in

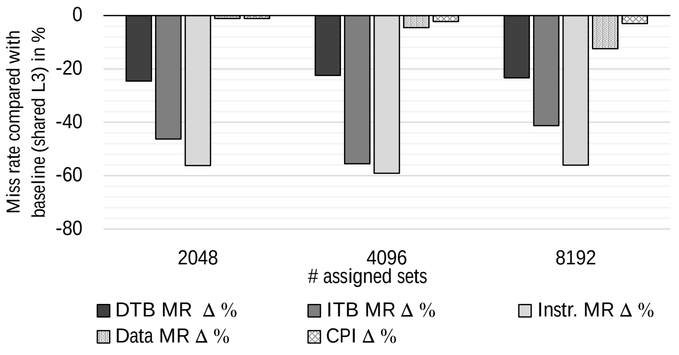


Fig. 10. Miss rate & CPI impact of CHUNKED-CACHE on the NI-Domain (increasing sets); compared to a shared L3 cache.

which we assign 1,024 sets to the NI-Domain and vary the number of unassigned sets. In the first run, all cache sets are allocated in our system, while in the second run, 4,096 sets remain unallocated and available for the NI-Domain. Figure 11 shows how the miss rates significantly decrease when 4,096 sets remain unallocated which demonstrates how CHUNKED-CACHE enables the NI-Domain to utilize unused cache sets.

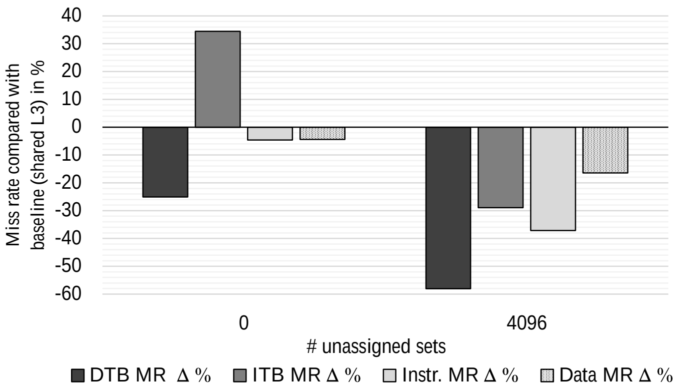


Fig. 11. Miss rate of CHUNKED-CACHE on the NI-Domain with varying number of unassigned sets; compared to a shared L3 cache.

Comparison with Partitioning-based Schemes. We compare CHUNKED-CACHE to a cache partitioning scheme which we implement on gem5, specifically way-based partitioning, being the only other strict cache partitioning approach. We run a number of experiments with a 5-domain setup where we assign the same cache capacity to the same benchmark in both, the CHUNKED-CACHE and way-partitioned cache – 1,024 or 2,048 sets in CHUNKED-CACHE and equivalently 1 way or 2 ways, respectively, in the way-partitioned setup. We show in Figure 12 how for the same cache capacity, CHUNKED-CACHE outperforms way-based partitioning for randomly selected benchmarks. In fact, for some benchmarks such as `625.x264_s` and `644.nab_s`, allocating 1,024 sets even outperforms 2 ways (double the cache capacity) on a way-partitioned cache. We calculate an average decrease of 43% in the miss rate for CHUNKED-CACHE vs. the way-partitioned cache for a 1 MB cache capacity (1024 sets) and a 39% decrease for 2 MB (2048 sets).

Scalability and Dynamic Cache Allocation. In Appendix A, we additionally evaluate CHUNKED-CACHE's ability to scale and support 32 I-Domains in parallel without degrading the performance of the NI-Domain (OS) and we also demonstrate how CHUNKED-CACHE supports the dynamic allocation of cache sets to an I-Domain during runtime.

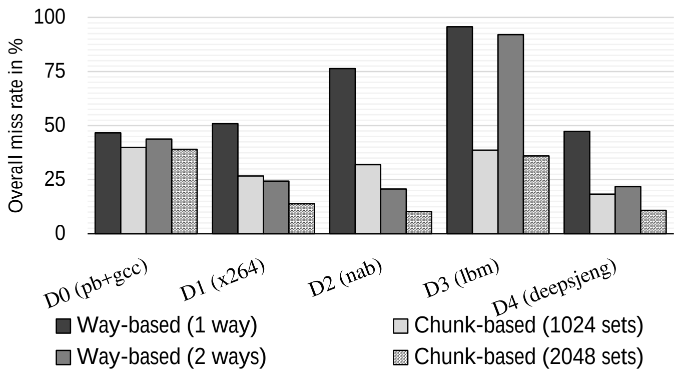


Fig. 12. Overall miss rate for SPEC CPU2017 benchmarks with CHUNKED-CACHE; compared to a way-partitioned cache.

C. Hardware Footprint and Power Consumption Evaluation

To evaluate the storage and logic overhead incurred by CHUNKED-CACHE, we synthesize our implementation targeting a single-issue single-core RISC-V processor [79] using Xilinx Vivado tools. While this processor does not provide an LLC, this is not necessary since we can still extend the existing simple cache controller to implement CHUNKED-CACHE, verify its functionality in cycle-accurate RTL-level simulations and evaluate its overheads.

Storage/Memory Overhead. The main contribution to the hardware area overhead of CHUNKED-CACHE is the extra storage required, rather than the logic itself, since that requires the fabrication of memory which consumes more gates than hardware logic. The extra storage is needed for the additional tag bits required per cache line (4-bit `DID` and 1-bit `SHARED` flag), the CST and DCAT. In our current prototype implementation targeting 16 domains, 16 MB LLC with 16-ways and 16,384 sets, and an allowed maximum of 8,192 sets per domain, the CST consumes 2 KB, the DCAT $\approx$ 224 KB, and the additional tag storage 160 KB, totaling 386 KB. This amounts to a negligible 2.3% storage overhead relative to a 16 MB LLC which would consume approximately an additional 2.7% area in fabrication.

The capacity of these tables and the consequent storage area overheads are directly impacted by how the various design/implementation trade-offs involved are configured in different implementations of CHUNKED-CACHE, namely 1) the number of active parallel domains supported (overhead increases only linearly) 2.) the total L3 cache capacity and its number of sets, and 3.) the maximum number of sets that can be allocated to a domain. For example, to support 32 domains, one more tag store bit is required costing an additional 0.25KB, relative to the overhead incurred for 16 domains as described above. The CST capacity is unaffected, while the DCAT capacity doubles to 448KB. The power consumption (evaluated below) would increase proportionally.

Logic Overhead. CHUNKED-CACHE requires extra hardware logic for the FSMs that handle the cache de-/allocation, and look up the tables prior to cache accesses (Section VI-A). We synthesize our hardware implementation using Xilinx Vivado targeting a ZedBoard Zynq-7000 FPGA board, and estimate a logic overhead of $\approx$ 1.6% relative to the single-core RISC-V processor that we extend. This would diminish relative to a significantly more complex multi-billion-transistor processor with a 3-level cache hierarchy which is the intended platform for CHUNKED-CACHE. Furthermore, this overhead does not increase as the number of domains supported by CHUNKED-CACHE increases.

Power Consumption Overhead. We focus here on the power consumption overheads incurred by the extended tag store and CST and DCAT tables, since the extra hardware added is dominated by them, and they contribute the most to the additional power consumption (static and dynamic) overheads. Besides, power consumption by cache memories is significantly more than logic, and is usually the largest contributor to the total power consumed by a chip. We estimate the power consumption overheads of CHUNKED-CACHE in 22nm technology using the CACTI-6.0 tool [70]. For a 16-way 16 MB cache with 64 B cache line size, the total leakage power increases from 5056.57 mW (baseline) to 5313.83 mW. The CST and DCAT incur an additional 365 mW, amounting to a total of 12.3% increase in the LLC power consumption. To support OS-specific chunk set indexing, the power consumption increases accordingly. If 2 sets are looked up in parallel (when 8,192 sets are allocated to the OS), the penalty on power consumption is negligibly minimal. When 4 or 8 sets are looked up in parallel, the power consumption overhead additionally increases by 5.5% and 27.1% relative to the baseline of 5056.57 mW, respectively. Relative to the overall chip power consumption of modern multi-core processors (90-150W), the LLC power consumption increases incurred by CHUNKED-CACHE remain reasonable.

VII. RELATED WORK

We categorize cache side-channel defenses which tackle the problem directly in the cache into two broad classes: partitioning-based and randomization-based. We focus in this section only on the most relevant works to CHUNKED-CACHE, which all propose hardware changes at the cache architecture.

A. *Partitioning-based Microarchitectures*

The partitioning-based defenses most related to CHUNKED-CACHE propose new cache architectures that assign cache resources (cache lines or ways) exclusively to protected domains. The TEE architectures Keystone [57] and CURE [6] implement way-based partitioning to assign cache ways exclusively to enclaves. SecDCP [94] forms security classes of applications with similar security requirements and assigns cache ways to them. DAWG [51] provides way-based cache partitioning in the context of speculative execution attacks. The main limitation of way-based partitioning is its inability to support a large number of protected domains in parallel since even large LLCs only comprise a small number of cache ways (up to 16). Moreover, these defenses lead to cache underutilization when assigned cache ways are not evenly utilized by a protected domain since the unused cache lines are blocked for all other domains on the system.

CHUNKED-CACHE, besides other approaches [95], [23], is more flexible since it partitions the cache on a cache-line basis. PLcache [95] assigns cache lines exclusively to processes which allows for a strict and fine-grained partitioning of cache resources. However, PLcache's strict isolation does not allow for caching data shared between processes and strongly impacts the overall system performance and fairness of the cache utilization. Moreover, PLcache does not protect against occupancy-based attacks since the adversary can still infer the victim's memory accesses by observing that the victim is unable to access/evict cache lines.

HybCache [23] assigns cache ways to protected domains (or enclaves) by providing a fully-associative mapping with random replacement for the ways to overcome the cache underutilization problem of way-based partitioning schemes. In contrast to PLcache, HybCache assigns only a subset of the cache resources to the protected domains which can be reclaimed by non-sensitive domains and thus, a fairer cache utilization is achieved which does not heavily degrade the overall system performance. However, HybCache does not scale practically with large LLCs since it would incur high power consumption overheads. Moreover, HybCache does not provide strong security guarantees against occupancy-based attacks since it does not enforce a strict partitioning.

In memory page-coloring schemes [29], [50], [101], [61], [22], the mapping from physical memory addresses to cache lines is utilized to ensure that the cache lines used from sensitive applications do not overlap. One problem with page-coloring is its high impact on the software memory layout. It cannot fully support DMA and requires modifying the memory management (OS or hypervisor). Moreover, the assignment of cache lines is static, i.e., modifying the number of assigned cache lines during runtime would require to alter the physical memory layout of the software which is highly impractical.

CHUNKED-CACHE, however, provides flexible cache-line partitioning that can scale to support a larger number of protection domains than the number of cache ways. It additionally overcomes the limitations of other cache-line partitioning techniques by providing support for shared memory and by scaling to large LLCs while still providing strict isolation. In contrast to page coloring schemes, CHUNKED-CACHE does not influence the memory layout, is compatible with commodity memory management software, and allows dynamic modification of the chunk sizes during runtime.

B. *Cryptographic Randomization Defenses*

These randomization techniques attempt to avoid the storage overhead of large randomized mapping tables that are deployed by earlier defenses [95], [63], [62] by relying on cryptographic primitives to reproducibly generate the randomized mapping. Time-Secure Cache [89] uses a set-associative cache indexed with a keyed function using the cache line address and process ID as its input. However, a weak low-entropy indexing function is used, thus, frequent re-keying and cache flushing must be performed which increases complexity and performance impact.

CEASER [77] also uses a keyed indexing function but without process ID. It also requires frequent re-keying of its index derivation function and re-mapping to limit the time interval available for an attacker to reconstruct the eviction set. Under a minimal eviction set construction algorithm of $\mathcal{O}(E^2)$ complexity, CEASER has been shown able to withstand attacks with a re-keying rate of 1%. However, under eviction set construction techniques with $\mathcal{O}(E)$ complexity [78], the re-keying rate needs to increase to 35%-100%, which incurs prohibitively high performance overheads. To resist these improved attacks, a skewed variant of CEASER, CEASER-S [78] was proposed that divides the cache ways into multiple partitions (skews), with different encryption keys used for each partition. A cache line maps to a different set in each partition, where one of the partitions is chosen randomly for the line placement, making the minimal eviction set construction more difficult.

ScatterCache [96] also uses keyed cryptographic indexing where cache set indexing is different and pseudo-random for every protected domain but consistent for any given key. Thus, re-keying is still required at time intervals to hinder the profiling and minimal eviction set construction efforts.

Phantom-Cache [87] relies on a set of hardware-efficient hash function and XOR operations to map a cache line to 1 of 8 randomly chosen sets in the cache, each with 16 ways, thus, increasing the associativity to 128. This requires accessing 128 locations on each cache access to check if an address is cached, resulting in a high power overhead of 67%.

Defenses based on cryptographic primitives have multiple weaknesses: 1.) These defenses remain only as secure as the best/fastest known attack strategy/minimal set eviction construction algorithm [12], [75] with no solid future-proof security guarantees. In fact, a recent work [86] has further shown that other attack techniques and workarounds can be used to exploit certain flaws in ScatterCache and CEASER-S to completely undermine them and their security guarantees. 2.) Their promised security guarantees often rely on the alleged, yet not thoroughly investigated unpredictability of low-latency cryptographic primitives. The primitives deployed by CEASER, CEASER-S and ScatterCache have been shown vulnerable to cryptanalysis which enables the construction of eviction sets without even accessing memory [74], [10]. Deploying primitives that resist formal cryptanalysis is also not practical since it would incur increased latency, thus, further degrading performance in the cache's critical path. 3.) If the re-keying rate is increased to mitigate novel attacks, the induced performance overhead renders these defenses impractical.

Mirage, a concurrent work, attempts to overcome the vulnerability to newer faster eviction-set construction algorithms, by eliminating set-associative eviction altogether [80]. However, besides still being vulnerable to occupancy-based attacks, Mirage does not support selectively enabling side-channel resilience only for execution domains that require it, thus, incurring a performance slowdown on the entire workload.

CHUNKED-CACHE, in contrast, eliminates the described unreliability and inflexibility fundamentally by providing strict, yet perfectly configurable and selective, partitioning across the execution domains. This enables each domain to allocate the cache capacity it requires and thus, experience the performance that it has opted to tolerate accordingly. This different paradigm provides well-grounded security assurances that stand the test of advances in cache side-channel attacks and different attack methodologies and complexities, without sacrificing performance. Instead, it provides by-design the possibility to tune the security-performance trade-off for each domain as desired, without overtaxing the OS either.

VIII. CONCLUSION

In this paper, we presented a novel side-channel-resilient cache microarchitecture, CHUNKED-CACHE, for TEE architectures, that enables each execution domain to flexibly and selectively configure its exclusive cache sets only when cache isolation and side-channel resilience is required. Unlike randomization-based cache microarchitectures recently proposed, CHUNKED-CACHE fundamentally mitigates side-channel attacks by enforcing strict cache partitioning, thus providing future-proof and solid security guarantees. It also outperforms way-based partitioning and scales to support a larger number of execution domains, without degrading the performance of the OS. In this work, we show how CHUNKED-CACHE incorporates this configurable performance-security trade-off by design in the cache microarchitecture to cater most optimally for TEE architectures. Through our security analysis and evaluation, we also show how on-demand sophisticated side-channel security, as well as performance, functionality and usability requirements are preserved in CHUNKED-CACHE, with small hardware and memory costs.

ACKNOWLEDGEMENT

We thank our anonymous reviewers for their valuable and constructive feedback. This work was funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – CROSSING/236615297.

APPENDIX

A. *I-Domain Scalability*

We also demonstrate how CHUNKED-CACHE scales for a larger number of parallel domains. As described in Section IV, the design of CHUNKED-CACHE allows to support more domains in parallel than the 16 domains we choose for our hardware implementation. Thus, we conduct scaling experiments where we run the `619.lbm_s` benchmark on every I-Domain and we increase the number of I-Domains from 4 to 8, 16 and up to 32. Running more I-Domains in parallel is not possible on our evaluation platform since the gem5 full-system simulation with 32 I-Domains already consumes the complete 186 GB of available RAM which unavoidably imposes certain limitations on our experiments. Given these constraints, we selected `619.lbm_s` as a benchmark because of its relatively small working set. Throughout these experiments, the NI-Domain (which runs the Linux kernel and one instance of `619.lbm_s`) gets 8,192 sets assigned. The overall miss rates for the NI-Domain, when scaling from 4 to 32 I-Domains, are stable, reaching 71.45%, 71.64%, 72.06% and 71.75%, respectively. Thus, with CHUNKED-CACHE, also a high number of I-Domains can be supported without degrading the performance of the NI-Domain (OS). Running even more domains was only limited by the memory constraints of our evaluation platform.

B. Dynamic Set Allocation

In another experiment, we analyze how the dynamic set allocation capabilities of CHUNKED-CACHE impact the NI-Domain and I-Domains during runtime. For this, we select a SPEC benchmark (`631.deepsjeng_s`) which achieves a relatively small average cache miss rate, when enough cache sets are available, in order to better demonstrate the behavior of the dynamic set allocation. We run the benchmark in 4 distinct I-Domains and as part of the NI-Domain. We simulate 24 billion cycles on our evaluation platform which corresponds to 12s worth of computing (given that we simulate processors with a clock frequency of 2 GHz). At the beginning of the experiment, the NI-Domain (D0) gets 8,192 sets assigned, the I-Domains D1-D3 512 sets each and the I-Domain D4 only 1 set. Then, during runtime, the size of D4's chunk is modified. After 3s, the chunk size is increased to 512 sets, after 6s to 2048 sets and after 9s decreased to 1 set. The chunk sizes of the domains D0, D1, D2 and D3 are kept constant throughout the duration of the experiment. We collect miss rate statistics for all domains every 75ms (150,000,000 cycles) and compute the arithmetic mean over the instruction and data miss rates of the page table walker and core.

The results of the experiment are shown in Figure 13, whereby we only show the miss rates for D0, D1 and D4 since the results of D2 and D3 are very similar to those of D1. The plot clearly shows how the increase and decrease of the chunks size affects the miss rate of D4. At the beginning, when only 1 set is assigned to D4, the miss rate fluctuates heavily around a value of 80%. At the time point 3s, when 511 additional sets are assigned to D4, the miss rate almost immediately drops to around 60%, thereby catching up with the miss rates achieved by D1. After another 3s, when D4's chunk size is increased to 2048, a low and stable miss rate of 20% is achieved. The fact that D0 experiences the same miss rate with 8,192 sets shows that applications are not always benefiting from an increased chunk size and thus, available sets are better redistributed to other benefiting domains to improve the overall system performance. After 9s, the chunk size is decreased to 1 set which again leads to a heavily fluctuating miss rate of around 80%.

Another interesting take-way from Figure 13 is that the flushing of all chunk sets, which happens after 6s, does not negatively influence the miss rate of D4, at least not when collecting the miss rate statistics at intervals of 75ms.

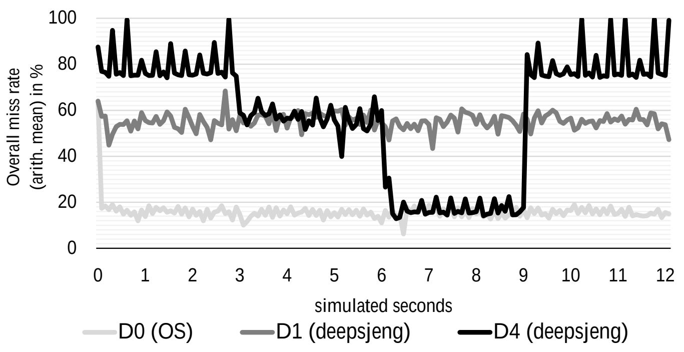


Fig. 13. Cache miss rate impact of CHUNKED-CACHE on the NI-Domain (D0) and I-Domains (D1, D4) when dynamically modifying the size of the cache chunk assigned to D1.

REFERENCES

[1] Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018.

[2] Intel Skylake X. https://www.7-cpu.com/cpu/Skylake_X.html, 2020.

[3] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. *ACM Symposium on Information, computer and communications security*, pages 312–320, 2007.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. *Cryptographers' Track at the RSA Conference*, pages 225–242, 2007.

[5] ARM Limited. ARM Security Technology – Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[6] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *USENIX Security Symposium*, 2021.

[7] Daniel J Bernstein. Cache-timing attacks on AES. 2005.

[8] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*, pages 159–176. Springer, 2012.

[9] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystem. In *CRYPTO*, 2000.

[10] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro. Brutus: Refuting the Security Claims of the Cache Timing Randomization Countermeasure Proposed in CEASER. *IEEE Computer Architecture Letters*, 2020.

[11] Joseph Bonneau and Ilya Mironov. Cache-collision Timing Attacks Against AES. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2006.

[12] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In *Micro*, 2020.

[13] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

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

[15] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE Symposium on Security and Privacy*, 2020.

[16] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *IEEE European Symposium on Security and Privacy*, 2019.

[18] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[19] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. In *CCS*, 2014.

[20] Standard Performance Evaluation Corporation. SPEC CPU 2017. https://www.spec.org/cpu2017, 2017.

[21] Victor Costan and Srinivas Devadas. Intel SGX Explained. Technical report, Cryptology ePrint Archive. Report 2016/086, 2016. https://eprint.iacr.org/2016/086.pdf.

[22] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, 2016.

[23] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *USENIX Security Symposium*, 2020.

[24] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, 2017.

[25] Goran Doychev and Boris Köpf. Rigorous Analysis of Software Countermeasures Against Cache Attacks. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2017.

[26] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*. ACM, 2013.

[27] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[28] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 693–707, 2018.

[29] Michael Godfrey. On The Prevention of Cache-Based Side-Channel Attacks in a Cloud Environment. Master's thesis, Queen's University, Ontario, Canada, 2013.

[30] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*, 2017.

[31] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[32] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[33] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-channel Protection Using Hardware Transactional Memory. In *USENIX Security Symposium*. USENIX Association, 2017.

[34] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[35] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer-Verlag, 2016.

[36] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*, 2015.

[37] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2016.

[38] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2011.

[39] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1991.

[40] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[41] Intel. Intel Software Guard Extensions. Tutorial slides. https://software.intel.com/sites/default/files/332680-002.pdf. Reference Number: 332680-002, revision 1.1.

[42] Intel. Intel Integrated Performance Primitives Cryptography Developer Reference. 2019.

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

[44] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2016.

[45] Kaplan et al. AMD memory encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[46] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A High-resolution Side-channel Attack on Last-level Cache. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2016.

[47] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. RIC: Relaxed Inclusion Caches for mitigating LLC side-channel attacks. In *IEEE/ACM Design Automation Conference (DAC)*, 2017.

[48] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[49] Khang T Nguyen. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/articles/introduction-to-cache-allocation-technology, 2016.

[50] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*. USENIX Association, 2012.

[51] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[52] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[53] Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, 1996.

[54] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[55] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic Quantification of Cache Side-channels. In *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2012.

[56] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Security Symposium*, 2018.

[57] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, 2020.

[58] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *USENIX Security Symposium*, pages 16–18, 2017.

[59] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

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

[61] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating Last-Level

Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[62] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014.

[63] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[64] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2015.

[65] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[66] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129. IEEE, 2012.

[67] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2012.

[68] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. *Cryptographers' Track at the RSA Conference*, pages 21–44, 2018. 10.1007/978-3-319-76953-0_2.

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

[70] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. https://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf, 2009.

[71] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[72] Mathias Payer. Hexpads: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*, pages 138–154. Springer, 2016.

[73] Colin Percival. Cache missing for fun and profit, 2005.

[74] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In *IEEE Symposium on Security and Privacy*, 2021.

[75] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache. *arXiv preprint arXiv:1908.03383*, 2019.

[76] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2019.

[77] Moinuddin K. Qureshi. Ceaser: Mitigating Conflict-based Cache Attacks via Encrypted-Address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[78] Moinuddin K. Qureshi. New Attacks and Defense for Encrypted-Address Cache. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2019.

[79] Roa Logic BV. RV12. https://github.com/RoaLogic/RV12, 2020.

[80] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *USENIX Security Symposium*, 2021.

[81] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, 2019.

[82] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[83] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[84] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 639–656, 2019.

[85] Sarabjeet Singh and Manu Awasthi. Memory Centric Characterization and Analysis of SPEC CPU2017 Suite. *arXiv preprint arXiv:1910.00651*, 2019.

[86] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. Randomized Last-Level Caches Are Still Vulnerable to Cache Side-Channel Attacks! But We Can Fix It. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2021.

[87] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In *NDSS*, 2020.

[88] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, 2017.

[89] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2018.

[90] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *USENIX Security Symposium*, 2018.

[91] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium*, 2018.

[92] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *IEEE Symposium on Security and Privacy*, 2019.

[93] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 41–46, 2011.

[94] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2016.

[95] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2007.

[96] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[97] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Atacks. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2017.

[98] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. To appear in the *Proceedings of the IEEE Symposium on Security & Privacy (IEEE S&P)*, May 2019.

[99] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*, 2014.

[100] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. volume 7, pages 99–112. Springer, 2017.

[101] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. Cryptology ePrint Archive, Report 2016/980, 2016. https://eprint.iacr.org/2016/980.

[102] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2012.

[103] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1723–1740, 2019.