Ceviche: Capability-Enhanced Secure Virtualization of Caches

Arnabjyoti Kalita
University of Virginia
mje6jj@virginia.edu

Yilong Yang
University of Virginia
yyilong@virginia.edu

Alenkruth Krishnan Murali
University of Virginia
alenkruth@virginia.edu

Ashish Venkat
University of Virginia
venkat@virginia.edu

***Abstract*—Modern systems make extensive use of resource virtualization to achieve high hardware utilization and minimize the total cost of ownership. However, sharing of physical resources invariably opens the door to side-channel exploitation where co-located attackers can covertly examine a victim's behavior and/or steal private information. Even though applications may not share data, they still compete for shared physical resources, notably for cache capacity. Since cache lookup is data/address-dependent, even the presence or absence of data in the cache can reveal sensitive information.**

This paper proposes Ceviche, a novel hardware virtualization strategy that allows for the secure allocation and use of physical cache resources among threads that belong to different trust domains. Ceviche enables a capability-based cache lookup by translating a given address-domain ID pair into a capability that encodes the access rights and the allowed set of operations on the physical cache line that it grants access to. By constraining cache lookup to occur based on a capability, Ceviche can achieve fine-grained partitioning of the cache at the granularity of a cache line, enforcing a wide set of confidentiality, availability, and fairness guarantees, while maximizing cache utilization. The paper presents detailed design mechanisms, policies, and optimizations along with extensive evaluation to demonstrate the feasibility of integrating the secure virtualization layer into modern multicore cache hierarchies.

Ceviche caches offer protections at all levels of the cache hierarchy and incur an average performance degradation of 2.4% when compared to an insecure baseline, while only imposing 1.8% additional performance degradation over state-of-the-art secure caches Mirage and ScatterCache.

1. Introduction

Resource virtualization and access control have formed the bedrock of modern computing systems. By abstracting physical resources into virtualized pools, operating systems and hypervisors enable secure, flexible, and on-demand resource allocation among the users of a system, while maintaining fairness and maximizing utilization. However, organizing shared microarchitectural resources such as cache memory into virtualized pools in a secure, efficient, and scalable manner continues to be a challenging endeavor.

Conventional virtualization solutions such as those that use Page Coloring [1] and Intel's Cache Allocation Technology (CAT) [2], [3] rely on providing exclusive access to all or specific partitions of the cache by pinning particular sets or ways, thereby minimizing interference among co-located programs or virtual machines in the system. However, due to their inherent inflexibility in organizing the cache into fine-grained partitions, they often suffer from low utilization and prohibitively high performance degradation when the number of trust domains is scaled beyond a certain point. Moreover, these solutions have been shown to be vulnerable to side-channel inference through indirect confused-deputy attacks [4] and cache occupancy attacks [5].

This paper proposes *Ceviche*, a novel hardware-based virtualization strategy that enables the secure allocation and use of cache resources at the cache line granularity, as governed by the principle of least privilege [6] and the principle of intentionality [7]. Ceviche enforces three key properties. First, each entity in the system is granted access to only those cache lines that have been specifically allocated to it and the set of operations that may be performed on such lines is restricted to the bare minimum it needs to accomplish its task. Second, no entity in the system is allowed to perform a cache operation without explicitly asserting their access rights. Third, to ensure fairness and availability, the allocation of cache resources is constrained by predefined hard and soft limits.

The key to our approach is the notion of a capability-based cache lookup, wherein a capability (a secure token granting access to a physical resource) is issued upon the successful allocation of a cache line during miss handling, and subsequent accesses to that line are granted only upon presenting the capability. To facilitate this, we introduce a novel cache design that augments a conventional direct-mapped SRAM data array with a content-addressable memory (CAM)-based capability register file (inspired by CAM-tagged cache designs employed in low-power embedded microprocessors [8], [9], [10], [11]), allowing for the seamless one-step translation of a given address-domain ID pair into its corresponding capability, if available (i.e., if data at that address has been allocated a cache line).

Each capability within the capability register file contains the cache line number it provides access to, along with the allowed set of operations (i.e., read, write, invalidate, and share) on that line. An important property of capabilities is

that they are unforgeable, which means the contents of a capability once burned, may not be modified, restricting an entity to the access rights and permissions encoded in it at the time of allocation. During its lifetime, a capability may be copied to allow secure and intentional sharing of cache lines with other entities in the system, as long as its permissions vector allows sharing. Since we implement capabilities as registers, entities sharing a cache line (e.g., threads running concurrently on different cores, but sharing the same last-level cache line, or SMT threads sharing the same private cache line) may simply map to the same physical register, greatly simplifying the effort required for tracking and managing copies of capabilities [6]. Further, a capability may also be revoked by invalidating the appropriate register, allowing entities to intentionally and gracefully give up access to the cache lines they are in possession of.

An important consequence of the capability-based cache lookup we introduce in this work is that it breaks the tight coupling of the address bits to the actual physical location (i.e., set number) of the cache line, allowing the allocation strategy to pick any available cache line to place data in (mimicking fully-associative caches), thereby limiting address-dependent (and by extension, data-dependent) contention, while simultaneously enabling a direct-mapped lookup as the capability already contains the line number. This has important security and performance implications.

First, by allowing the physical location of a cache line to be independent of its address, we can thwart conflict-based cache attacks [12], [13], [14], [15], [16], [17], [18] that rely on constructing eviction sets based on address-dependent contention. Second, since capability revocation needs to be voluntary, intentional, and explicit, invalidation of a cache line may only be triggered in the case of self-evictions (i.e., the eviction of an entity's own cache lines where an entity could include one or more threads within the same trust domain). In essence, this ensures that distrusting parties cannot force the eviction of each others' cache lines. Third, our ability to explicitly disallow the sharing of cache lines among distrusting parties (by specifying it in a capability's permission vector) prevents flush-based cache attacks [19], [20], [21], [22] that hinge on flushing shared lines. Fourth, owing to its CAM-tagged design, Ceviche reaps the performance benefits of a flexible conflict-averse fully associative allocation and a fast direct-mapped lookup.

The key contributions of this work are as follows:

- We introduce Ceviche, a novel capability-based hardware virtualization solution that allows for the secure allocation and use of cache resources among multiple entities in a system, enforcing key security properties that protect against conflict-based, flush-based, occupancy-based, denial-of-service, and confused deputy attacks.
- We present detailed design mechanisms and policies, including hit procedures, miss handling, replacement policies, and coherence logic, that together enable its integration in a modern multi-level cache hierarchy. Through CACTI [23] analysis, we show that the additional hardware imposed by our solution incurs a minor area and power overhead.
- By introducing an additional layer of virtualization, Ceviche is able to break the tight coupling between the address bits and the physical location of a cache line, while also enabling a direct-mapped organization with fully associative allocation. We observe that Ceviche incurs an average of 2.4% performance degradation over an insecure baseline. Ceviche also protects all levels of the cache while only imposing 1.8% additional performance degradation over state-of-the-art secure cache designs Mirage and Scatter-Cache that only protect the last-level cache.
- We show that Ceviche has the ability to seamlessly scale to multiple trust domains while maintaining high cache utilization and low miss rates through the mere reconfiguration of its soft limits that constrain the allocation of capabilities.

2. Background and Related Work

Cache Hierarchies. Modern processors typically employ a multi-level cache hierarchy with each level successively larger in size, albeit with higher access latency. Each core is equipped with *private* L1 and L2 caches. The L1 cache is closest to the core and thereby enables fast access. It is typically organized as a *split* cache, with separate memory arrays serving instructions and data. The L2 cache, on the other hand, employs a *unified* organization that stores both data and instructions. Further, all the cores in the processor *share* a large *last-level cache* (LLC) that is farthest from the cores and entails the highest access latency. Note that, while the L1 and L2 caches are private to a physical core, in designs that feature Simultaneous Multithreading [24], [25], they are shared among the hardware threads in each core.

Cache Organization and Addressing. Modern caches are typically organized as N-way *set-associative* caches, constraining data to be stored in one of N locations within the cache. For example, the 48 KB private data cache in Intel's Icelake is organized into 12 ways and 64 sets, allowing data being addressed at any given set to be stored in one of 12 cache lines within that set, with each line containing 64 bytes of contiguous data. Each line is tagged by a set of address bits uniquely identifying the data in it. Each cache access is then made in two steps – (a) *indexing*, which involves identifying the appropriate set using a subset of the address bits, and (b) *tag matching*, which involves identifying the appropriate cache line within the set (i.e., way of a set) by matching the higher order bits of the address with the tag. In a *fully-associative* cache, the indexing process does not exist as the cache is organized as one large set, and in a *direct-mapped* cache, each set contains only one line and thus requires only one tag matching operation. Consequently, fully-associative caches are conflict-averse while set-associative and direct-mapped caches are prone to conflicts where data at addresses with the same index bits but different tag bits contend for the same set.

Conventional caches are thus organized into separate data and tag arrays where a search across the appropriate set within the tag array is used to identify the data line to be read out from the corresponding set within the data array. The tag arrays may be implemented using static random access memory (SRAM) or content-addressable memory (CAM) [8], [9], [10], [11], [23], [26]. In SRAM-tagged caches, tag matching and data reads for all lines within a given set (identified by the index bits) are performed in parallel, with the result of the tag matching operation used to select one of the data lines to be serviced up the hierarchy. In CAM-tagged caches, an associative search of the tag array is used to identify exactly one line to be looked up from the data array in a direct-mapped fashion. While CAM-tagged caches are conflict-averse and enable greater utilization, they are also more expensive as standard CAM cells are known to consume more area than standard SRAM cells [10], [23]. Both SRAM-tagged and CAM-tagged caches (especially larger caches) are further divided into smaller banks to reduce access time and lower power consumption.

Note that L1 caches are typically *virtually addressed and physically tagged* (VIPT), which means the virtual address is used for indexing, but the physical address is used for tag matching, allowing address translation to overlap with the indexing process. L2 and last-level caches are typically physically addressed and physically tagged.

Inclusiveness. In an *inclusive* cache hierarchy, the upper-level caches are included in the lower-level caches. This means, for example, that all data contained in the L1 cache must also be present in the L2 cache. Conversely, if a line in the L2 cache is evicted, the corresponding line in the L1 cache must also be flushed out, if present. On the other hand, in an *exclusive* cache hierarchy, the upper and lower level caches contain data that are exclusive from each other. Caches may also be *non-inclusive* where neither of these restrictions apply.

Write Policies. Lower-level caches in an inclusive hierarchy act as backing stores for the data in upper-level caches that may implement either a *write-through* policy where all data written to the cache is also immediately written to the backing store, or a *write-back* policy where data is written to the backing store only upon eviction of a *dirty* cache line (i.e., whose data has been modified). Furthermore, in a *write-allocate* cache, a store miss would entail loading data from the backing store and then writing to it, whereas in a *write-around* cache, a store miss would result in the data being written directly to the cache line in the backing store.

Multicore and Coherence. Coherence issues may arise in modern cache hierarchies because multiple cores could end up maintaining copies of shared data/instructions in their own private L1 and L2 caches, and when a copy becomes stale, it needs to be *invalidated*. Coherence issues are typically resolved through cache coherence protocols that ensure the single writer multiple reader (SWMR) invariant [27]. Modern processors employ some variant of the MESI coherence protocol that stipulates the allowed set of states a cache line (typically tracked using a hardware directory structure) may be in to *modified* (M), *exclusive* (E), *shared* (S), and *invalidated* (I) (with some variants implementing an *owner* (O) or *forward* (F) state), and further specifies the appropriate set of transitions between these states.

Resource Virtualization. Resource virtualization enables physical resources to be aggregated into virtualized pools, allowing for a flexible allocation across various users of the system, while simultaneously maximizing utilization. Modern virtualization platforms such as VMWare's vSphere, Microsoft's Hyper-V, and OpenStack perform resource virtualization at various levels, for sharing available storage, network bandwidth, and even compute capacity. The academic literature includes multiple secure virtualization solutions geared towards flexible and efficient management of system-level resources [42], [43], [44], [45], [46], [47], [48], although the focus of this work is to enable secure virtualization of the processor cache resources.

Capability-Based Security. Capability-based security was introduced by Saltzer and Schroeder [6], and there have been several hardware and software approaches [7], [49], [50], [51], [52], [53], [54], [55], [56], [57] proposed since to enforce capability-based protection. These systems enforce the *principle of least privilege* providing every user with the least set of permissions and privileges required to accomplish its goal, enforced through capabilities that authorize specific operations on a given resource. They also enforce the *principle of intentionality* by stipulating that, for every access, users explicitly assert their access rights. Capabilities, by definition, are unforgeable, which means once the privileges in it are burned, they cannot be altered. However, copies of capabilities can be created for secure delegation of access rights.

Cache Attacks. Side-channel attacks that target the cache exploit the timing differences between hits and misses to observe and draw inferences about a victim's sensitive data-dependent cache access patterns. These attacks can be broadly classified into – (a) contention-based attacks [12], [13], [14], [15], [16], [17], [58], [59], [60], [61] that hinge on competing for cache lines that map to the same cache set in a conventional set-associative cache, and (b) reuse-based attacks [19], [20], [21], [22], [62] that rely on instructions exposed by the hardware to flush a particular cache line that contains data shared by the attacker and the victim (e.g., in case of shared libraries or memory de-duplication). Reuse-based attacks also include those that exploit coherence protocols [63], [64] to force invalidations or generate other observable timing signals. In either case, the hit/miss timing behavior for secret data-dependent accesses could be used to ultimately deduce the secret.

Secure Cache Designs. Multiple secure cache designs have been proposed in response to microarchitectural attacks that have targeted the cache as the dominant side channel. These designs fall into two major categories.

First, hardware cache partitioning solutions [1], [2], [3], [29], [33], [34], [35], [36], [38], [40], [41], [65], [66] aim at mitigating cache attacks by constraining distrusting threads to different partitions in the cache. CATalyst [3] supports two trust domains by assigning two LLC ways to a secure domain at boot time, thereby reserving the remaining eigh-

TABLE 1: Comparison against other secure caches

Secure Cache	Solution Type	Conflict-based Attacks	Cache Policy Attacks	Reuse-based Attacks	Cache Occupancy Attacks	Confused Deputy Attacks	Denial of Service Attacks	Caches Protected	Domains Supported
Random Fill [28]	Randomization	✘	✓	✘	✘	✘	✓	L1-D	2
Newcache [29]	Randomization	✓	✓	✘	✘	✘	✘	L1-I, L1-D	# RMT_IDs
CEASER [30]	Encryption	✓	✓	✘	✘	✘	✘	L3	≤ (# L3 sets)
SCATTER [31]	Randomization	✓	✓	✘	✘	✘	✘	L2	8
MIRAGE [32]	Encryption	✓	✓	✘	✘	✘	✘	L3	256
PL Cache [33]	Fine-Grained Partitioning	✓	✓	✘	✓	✘	✓	L1-D	≤ (# lines in L1-D)
SecDCP [34]	Way Partitioning	✓	✓	✘	✓	✘	✓	L3	≤ (# ways in L3)
RP Cache [33]	Randomization	✓	✓	✘	✓	✘	✓	L1-D	≤ (# L1-D sets)
NoMo Cache [35]	Way Partitioning	✓	✓	✘	✓	✘	✓	L1-D	≤ (# ways in L1-D)
SecSMT [36]	Set partitioning	✓	✓	✘	✓	✘	✓	L1-I/D, L2	2
CATalyst [3]	Way Partitioning	✓	✓	✓	✓	✘	✓	L3	2
MI6 [37]	Set Partitioning	✓	✓	✓	✘	✓	✘	L1, L2, L3	64
DAWG [38]	Way Partitioning	✓	✓	✓	✘	✘	✘	L1, L2, L3	≤ (sockets x ways in largest-way cache)
SHARP [39]	Random Replacement	✓	✓	✓	✘	✘	✘	L3	≤ (# lines in L3)
BCE [40]	Set partitioning	✓	✓	✓	✓	✘	✘	L3	512
Composable Cachelets [41]	Fine-Grained Partitioning	✓	✓	✓	✓	✘	✓	L3	# (cachelets per enclave) x # enclaves
Ceviche	Fine-Grained Partitioning	✓	✓	✓	✓	✓	✓	L1-I/D, L2, L3	≤ (# lines in L3)

teen ways for the insecure domain. PLCache [33] secures the L1 data cache by locking lines of interest for creating flexible partitions and disallowing cross-partition eviction. Similarly, in NoMo cache [35], defenses for L2 and L3 caches are out of scope but it allows for flexible partitioning of the L1 between two SMT threads. SecDCP [34] offers LLC protection by ensuring one-way information flow from public to confidential applications but it allows for dynamically changing the partition size using cache demand information. In contrast, DAWG [38] secures all levels of the cache through coarse-grained way partitioning that isolates the visibility of any cache state changes to a single protection domain. MI6 [37] uses page coloring-based set partitioning that maps pages from distrusting domains to disjoint cache sets. Similarly, Bespoke Cache Enclaves [40] perform scalable and flexible LLC set-partitioning wherein each domain owns up to 512 clusters. On the other hand, Composable Cachelets [41] employ both way and set-partitioning to allow dynamic reconfiguration of existing cachelet partitions while reserving some cache ways for non-enclave applications.

Second, randomization-based solutions [28], [29], [30], [33], [67], [68], [69] aim at randomizing the allocation and access procedures for the cache, thereby preventing deterministic conflict behavior. SCATTERCache [31] secures shared L2 caches in embedded ARM processors by randomizing the mapping of address to cache set using a key, tag-index pair, and domain ID. The random fill cache [28] replaces the demand L1 data cache fill by another random fill within a neighborhood window so as to not completely forgo the advantages due to locality. Similarly, Newcache [29] introduces a layer of indirection in the L1-I and L1-D caches where the address is first mapped to a logical direct-mapped (LDM) cache and then each LDM cache line is mapped in a fully associative and randomized way to a physical cache line. MIRAGE [32] leverages the V-way cache design to allow for global random LLC evictions at an extra tag storage cost. In contrast to the former randomization approaches, CEASER [30] leverages encryption to achieve randomization where a low-latency block cipher converts the physical line address into an encrypted line address in the shared LLC. SHARP [39], on the other hand, changes the replacement policy such that attacker-induced evictions do not generate inclusion victims in private caches. Most caches in Table 1 self-report vulnerability to the attacks. Randomization does not inherently protect against reuse-based attacks as cross-domain sharing or invalidation is still possible (except for SHARP [39] which disables flushing). Only the strictest partitioning-based solutions thwart occupancy-based and denial-of-service attacks. For confused-deputy attacks, we consider XLATE [4] style attacks, which only MI6 protects against. Deng [70] provides a more comprehensive survey of these class of attacks.

While our solution falls into the partitioning-based approach, we not only enforce a greater set of security properties, but we protect all cache levels, including the instruction cache, while being able to scale gracefully to several protection domains, as shown in Table 1.

3. Threat Model

Security Guarantees. The goal of Ceviche is to enable secure and scalable virtualization of cache resources among threads that belong to different trust domains, such that each thread owns exclusive access rights to the cache lines allocated to it (unless explicitly shared), and its cache timing

behavior may not be influenced by threads from a different trust domain. We enforce the following security properties and defend against attacks such as the ones enumerated in Table 1 that seek to violate these properties.

a) Access based on Least Privilege: Each thread is allowed to access only those lines that have been allocated to it and the set of operations (i.e., read, write, invalidate, share) that they can perform on the line are limited to the bare minimum needed. This not only allows us to strictly partition cache resources among mutually distrusting domains, but enables the flexible enforcement of fine-grained permissions. For example, this could allow a line to be shared as read-only among multiple threads, with the caveat that it may not be invalidated by a thread outside of its trust domain.

b) Unforgeability of Access Rights: The access rights and permissions of a line are baked into capabilities that are conferred upon a thread at the time of line allocation. We enforce that these capabilities are unforgeable, in that the contents of the register holding the access rights and permissions may not be altered, once burned. However, when sharing is explicitly allowed by the owner of a capability, sharer threads may map to the same capability register (see Section 5.3 for details of the exact mechanism for sharing of capabilities). Note that this prevents some Confused Deputy [71] scenarios as authorization is still based on the permissions encoded in the capability, regardless of what entity performs the access. In other words, capability-based protection inherently ensures that a privileged entity deputized by an attacker would still perform accesses using the access rights delegated to it by the attacker (achieved via sharing of capabilities), rather than its own access rights.

c) Intentionality: No cache operations are allowed unless access rights are explicitly exercised through capabilities. Even though a thread has access to multiple lines through different capabilities, each cache operation it undertakes needs to be explicitly tied to and authorized by the corresponding capability. By supplanting the global notion of access rights in favor of fine-grained capabilities, we are able to securely share resources and delegate access rights to other entities, preventing some confused deputy scenarios. Note that, in the interest of transparency, our cache line capability follows from the address that the thread is using since modifying loads and stores to explicitly present a capability instead of an address entails a major ISA overhaul.

d) Address-Independent Cache Access: The capability-based cache lookup allows us to enforce that virtual or physical address bits do not directly influence the physical location of a cache line, thereby thwarting the eviction set construction process used in conflict-based attacks such as PRIME+PROBE [58], [59] that relies on address-dependent cache accesses. This is because any register in the capability register file can point to any cache line in the data array and its CAM-based implementation ensures a fast associative search of the entire register file, independent of the address bits, rather than looking up a particular cache set based on the index bits of the address, as is typical of conventional SRAM-tagged set-associative caches.

e) Voluntary Revocation: We enforce that capabilities owned by a thread may not be revoked involuntarily outside of its trust domain. This allows us to implement strict policies where cache line invalidation and replacement operations are restricted to always occur within a trust domain, mitigating several contention-based, flush-based, and occupancy-based attacks that hinge on evicting lines outside of their domain. Once a capability is revoked, the corresponding register is invalidated and released into a free pool, preventing any *use-after-free*-style misuse of capabilities.

f) Availability: Finally, we enforce that any given entity in the system will neither be deprived of allocating cache lines up to a minimum guarantee limit, nor be allowed to exceed its maximum allocation limit. These limits can be specified per-thread or per-domain, so as to mitigate attacks that rely on exploiting occupancy behavior.

Scope. While we provide software with the flexibility to freely organize into trust domains, maintain domain IDs in a dedicated model-specific register and establish trust relationships as appropriate, we assume that hardware modules within the cache controllers responsible for creating and maintaining capabilities, verifying access rights, and performing cache operations are all trusted and tamper-resistant. For example, we consider voltage/frequency scaling, temperature, and electromagnetic attacks that attempt to induce faults in capability registers or other hardware metadata structures to circumvent our unforgeability properties to be out of scope. We also consider other side-channel attacks such as those that target functional units and execution port contention [72], [73], [74], branch predictors [75], [76], [77], and TLBs [78] to be out of scope, similar to other existing secure cache designs [3], [30], [32], [33], [34], [35], [39].

4. Overview of Hit and Miss Procedures

This section introduces key components of Ceviche that facilitate capability-based allocation and lookup. To simplify the discussion, we only consider a single-level cache and a single-core, single-threaded processor below, but in Section 5, we discuss detailed design mechanisms that enable their integration in modern multithreaded and multicore processors that feature a multi-level cache hierarchy.

Hit Procedure. Conventional private caches are typically virtually-indexed and physically-tagged, allowing them to index into the appropriate cache set using index bits derived from the virtual address, while performing address translation in parallel using the Translation-Lookaside Buffer (TLB). The tag bits from the physical address are then used to perform an associative lookup of the cache line within that set through parallel tag comparisons. In contrast, Ceviche L1 caches employ a CAM-based capability register file (CRF) that can be looked up by providing a virtual address (tag portion) and a domain ID pair.

Each capability register specifies the cache line it provides access to, along with a permissions vector that indicates the set of authorized operations (read, write, invalidate, and share) that may be performed on it. It also contains the physical address tag bits to assist in resolving aliases as described in Section 5.4. The CRF is provisioned to contain

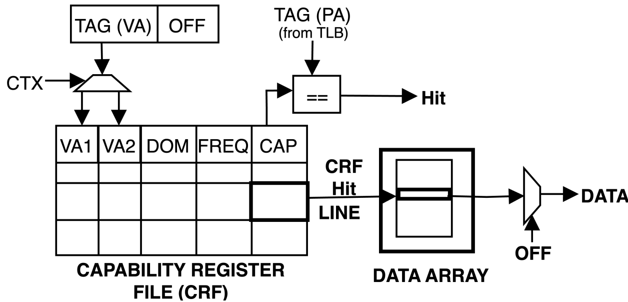

Figure 1: Ceviche L1 Cache Hit procedure

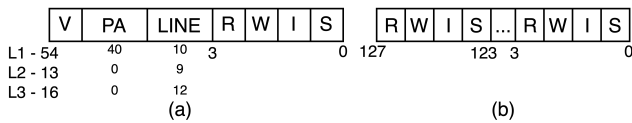

Figure 2: (a) Capability Register (b) Model-Specific Register encoding Cross-Domain Sharing Policies

as many registers as the number of lines in the L1 cache (512 entries for the instruction cache and 768 entries for the data cache). The access rights in the capability are verified to explicitly ensure that the desired operation is allowed before accessing the cache. Figure 1 illustrates this process.

Miss Handling. In conventional caches, a cache miss would entail line allocation and line fill operations, as governed by the cache's insertion and replacement policies. In Ceviche caches, a miss is detected when the CAM-based lookup of the CRF fails to find a match, in which case a new line would need to be allocated and a new capability would need to be issued with the appropriate access rights. To this end, we maintain a free pool of previously invalidated capability registers and cache lines, similar to the physical register allocation logic used in out-of-order processors [79]. If a free register is available, a new capability (shown in Figure 2(a)) is generated as follows.

First, a cache line is pulled from another free pool that maintains invalid lines, and that line number is recorded in the capability. Note that this mimics a fully associative allocation in that data is allowed to be placed in any available cache line upon a miss, without regard to its address. If the free pool is empty, a secure cache replacement procedure is triggered (described in Section 5.2).

Second, a permissions vector is burned into the register to indicate whether the allocated line can be read from, written to, invalidated, and/or shared. We employ the following rules to populate the permissions vector – (a) the *write* bit in the permissions vector is not set if the backing store in the lower-level cache (or the page table entry, when the backing store isn't available, i.e., if it is the last inclusive cache in the hierarchy) indicates that it contains read-only or execute-only content, (b) the *share* bit is set if the thread intends to share the line outside of its trust domain, and (c) the *invalidate* bit is set if the thread voluntarily permits the line to be invalidated by any thread outside of its trust domain. Note that the latter two policies are implemented per-thread but cache-wide, and can be configured by software through model-specific registers (as shown in Figure 2(b)) that maintain a bit vector indicating whether a shared cache line owned by the current domain is readable, writable, flushable or shareable outside of its domain. The rules together allow for significant flexibility. For example, although not desired, in systems that are less security-conscious, a highly permissive policy could be implemented by turning on all bits in the capability. A more restrictive policy could be implemented by turning off only the write and invalidate bits, allowing read-only sharing across trust domains. Finally, a strict no-sharing across trust domains policy could be implemented by turning off the *share* and *invalidate* bits in the capability. We implement this latter policy by default.

Note that the RWX permissions burned into each cache line capability typically follow from paging-based memory protections, although we provide the flexibility to further tune them at the cache level. For example, a read-only sharing (follows from paging) without invalidation (new in Ceviche) policy could prevent Flush+Reload attacks [19]. Moreover, paging-based protection checks are enforced later in the pipeline at the commit stage resulting in Meltdown-style attacks [80], [81]. However, Ceviche prevents the cache state from being updated when the right capabilities aren't presented (e.g., wrong physical address, R, W, X, S, I bits), providing a significant advantage as no information is disclosed without the need for inhibiting speculative execution.

Third, as soon as the TLB access is complete and the physical address is available, the physical tag bits are burned into the capability register. For all future accesses, the physical tag in the capability register is compared against that obtained from the TLB access. This ensures that a capability register does not contain a stale capability, preventing *use-after-free*-style misuse. Section 5.5 discusses this in detail.

Finally, the virtual address-domain ID pair (key) and the capability along with relevant metadata such as valid, dirty, and replacement bits initialized appropriately (value) are recorded in the appropriate entry within the CRF.

5. Design Mechanisms and Policies

This section delves into the design details of Ceviche and discusses its integration into multi-level cache hierarchies in modern multicore and multithreaded processors.

5.1. Integration with a Multi-Level Hierarchy

Capability Register File Organization. Figure 3 shows the organization of Ceviche caches in a multi-level hierarchy. Each cache in the hierarchy maintains its own set of capabilities to provide secure access to the lines contained in it and is equipped with a dedicated banked CRF maintaining as many registers as the number of lines in the cache it provides access to. Most modern cache designs (both SRAM-tagged and CAM-tagged) typically employ a banked design to reduce access latency and power consumption. During a

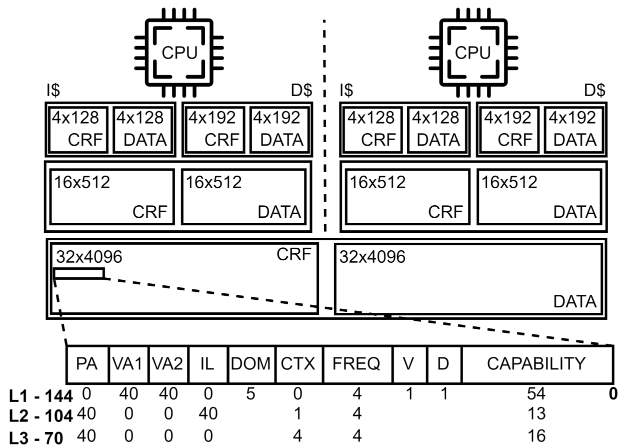

Figure 3: Integration with a Multi-Level Hierarchy

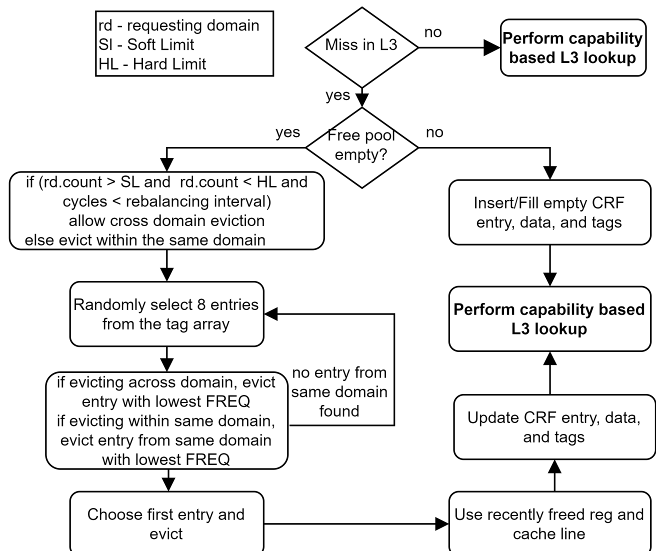

Figure 4: Flowchart for the LLC replacement policy

Rebalancing interval	50k	100k	200k	500k
Cross domain eviction count	167	95	56	34

TABLE 2: Sweep of Rebalancing Interval

cache lookup, all CRF banks are probed in parallel using the address-domain ID pair as the key. While we use the virtual tag bits to probe the L1 CRF to overlap the matching operation with the TLB access, we use the physical tag bits to probe the L2 and L3 CRFs, allowing us to forgo storing the physical tag bits in L2 and L3 capability registers. Note that we are expected to find a match in exactly one entry from one bank in case of a hit and none in case of miss. In Section 5.4, we discuss mechanisms to resolve special cases where more than one entry could produce a match due to the virtually-addressed L1 CRF.

Data and Tag Array Organization. The organization and lookup procedure of Ceviche caches mirrors that of CAM-tagged caches [8], [9], [10], [11], [23], [26], where the CRF is organized as a CAM and the data array is organized as a direct-mapped SRAM structure. The data array access is initiated only after the CRF access yields a valid capability that contains the line number to be accessed and its permissions vector affirms that the intended operation on the line is allowed. We make no additional changes to the cache parameters, including the number of banks, read, write, and search ports. We examine the latency, power, and area impact of Ceviche caches in detail in Section 9.3.

Inclusiveness. Since the L1 cache is virtually addressed (i.e., the virtual address tag-domain ID pair is used to obtain a capability), to maintain inclusiveness, for each included cache line in the L2, we provide a direct link to the corresponding line in L1, by maintaining a special set of *inclusion link* (IL) bits as part of L2's CRF. The IL bits essentially contain the virtual address tag of the included line in the upper-level cache, facilitating L1 CRF lookups during potential cache line invalidations. . The IL bits may be updated as part of handling an upper-level cache miss, which already entails copying data over from the lower-level line to the upper-level line, necessitating the lookup of both the L1 and L2 capability registers. This simultaneous lookup of both L1 and L2 capabilities due to back invalidations in the L2 cache is extremely rare. For the SPEC2017 benchmarks, an average of 2 and a maximum of 9 such simultaneous lookups were observed for every 1000 memory accesses.

5.2. Cache Replacement Policy

Since Ceviche caches employ a fully associative allocation strategy, implementing recency-based replacement would be inefficient due to the overhead of having to maintain and traverse large tree structures upon every access. We instead turn to a randomized frequency-based policy [60], [82] that maintains a 4-bit saturating counter per cache line. The counter is initialized to a small non-zero value (to prevent immediate eviction), which is incremented upon every access and decremented upon reaching a preset expiration interval. In our implementation, we start the counter at 5 and set the expiration interval to 64 cycles for L1 Icache, 128 cycles for L1 Dcache, 512 cycles for L2, and 4096 cycles for L3. During cache replacement, eight cache lines are chosen at random, forming the candidate set of victim lines for replacement. Among these candidates, the most infrequently used line that belongs to the same trust domain as the currently running thread is chosen as the victim. Figure 4 illustrates this.

Further, in accordance with our goals of maintaining fairness and availability, we extend our replacement algorithm as follows. First, we impose a hard and a soft limit on the number of capabilities granted for each trust domain. These limits may be configured dynamically by the system administrator or the runtime management engine (the software interface is described in Section 6). Second, we maintain counters to track the number of capabilities granted (and thus the number of lines allocated) for each domain. If the cache contains available lines, any new allocation

request is granted, as long as the counter has not reached its hard limit. As soon as the counter reaches its hard limit, all further cache line allocation requests are granted only upon the successful replacement of an already allocated line that belongs to the same trust domain, ensuring that no single domain is allowed to monopolize the available cache resources. On the other hand, if the cache is full and the counter hasn't yet reached its soft limit, a rebalancing procedure is initiated to improve fairness. A rebalancing operation involves evicting lines belonging to a domain that has exceeded its soft limit, with the constraint that only one eviction per rebalancing interval is allowed. To limit the amount of cross-domain occupancy information leaked, the rebalancing interval is configured to be a very long time (100,000 cycles in our implementation). We arrive at this rebalancing interval by performing a sweep of various intervals while running highly and moderately-intensive programs contending for the shared L3 cache. As can be seen from Table 2, with increase in the rebalancing interval, an exponential decay trend is observed in the number of cross-domain evictions. We select 100,000 cycles as a suitable threshold considering fairness and relatively fewer cross-domain evictions. Note that a single eviction every 100,000 cycles does not provide a fine enough granularity to create memorygrams [5], [36] to launch a cache occupancy attack and the eviction does not leak any spatial information thereby limiting contention-based attacks.

5.3. Sharing and Coherence

In a multicore processor, threads running on different cores may share data, requiring shared access to a last-level cache line. We consider two types of sharing – (a) location-aware sharing, where pages that map to the same physical location on the disk are shared among different threads (e.g., shared libraries), and (b) content-aware sharing, where the system coalesces unrelated pages with identical contents (a.k.a memory deduplication). While sharing happens at the granularity of a memory page, Ceviche offers flexibility to set or override sharing permissions at the granularity of a cache line by allowing software to configure per-domain sharing policies (described in Section 6).

Capability Sharing. If the threads intending to share a cache line belong to the same trust domain, we implicitly allow sharing as the two threads would use the same physical tag-domain ID pair to access the shared last-level CRF entry. If the threads belong to different trust domains, sharing would need to be intentionally permitted by and agreed upon by both threads. To check if cross-domain sharing is permitted, we extend the last-level CRF lookup as follows. First, the model-specific register encoding cross-domain sharing policies is looked up for the thread requesting to potentially share a line. If cross-domain sharing is allowed, the last-level CRF lookup is performed with the domain ID masked. If the lookup results in a hit, the capability is examined to see if the permissions vector has the *share* bit turned on, indicating whether cross-domain sharing has been allowed by the owner of the cache line. As misses are serviced up the hierarchy, the capabilities providing access to the line's copies in each core's private caches would inherit the same permissions as the ones in the LLC capability. On the other hand, if sharing is disallowed by either thread, a trap is issued to the operating system instructing it to remap the virtual page to a copy of the underlying physical page, similar to copy-on-write.

Note that this enforces the principle of intentionality as the decision to share a line across domains is made only after both threads explicitly declare their intent to share a capability, and all sharers are further bound by the permissions initially burned to and contained within the capability. This, for example, allows us to implement a restricted mode of sharing by disabling *write* and *invalidate* permissions.

Maintaining Coherence. Conventional processors maintain in-cache directories that keep track of the state of every line, alongside a list of cores that store a copy of the line in their respective private caches, and resolve coherence issues by issuing invalidation requests to all cores holding stale copies. In an inclusive hierarchy, Ceviche caches are also able to seamlessly take advantage of the same directory coherence mechanisms. This is because in-cache directories only extend the LLC tag array to maintain directory information, and the LLC CRF in Ceviche caches can be extended to include this information with other existing metadata such as valid, dirty, and replacement bits. When the LLC is non-inclusive, a separate physically tagged full directory may be employed to track shared L2 lines and their corresponding capabilities. In either case, when a coherence transaction fails due to misconfigured permissions in the capability (e.g., when reading, writing, and sharing are allowed, but invalidation is disallowed), a general protection fault (GP) is raised.

5.4. Simultaneous Multithreading

In SMT designs, private caches are shared between the threads running on the same physical core. Since our L1 CRFs are virtually tagged, we run the risk of incurring synonym aliases (different virtual addresses point to the same physical addresses) and homonym aliases (same virtual addresses from different threads point to data at different physical addresses). In order to ensure that these aliases are resolved, the L1 CRF is extended to be keyed using two virtual tag-domain ID pairs, albeit only one virtual address-domain ID pair is supplied at the time of lookup. When a line is not shared by the two hardware threads, its CRF entry is keyed by only one virtual tag-domain ID pair. However, when a line is shared, its CRF entry is keyed by two virtual tag-domain ID pairs, one for each thread, thereby resolving potential synonym aliases. This is possible because, upon an L1 miss and a subsequent L2 hit, it can be detected that the two threads already share a line in the L2 cache and the corresponding line is included in the L1, at which point, the L1 CRF entry is updated to also be keyed using the virtual tag-domain ID pair of the sharer thread.

Further, during an L1 CRF lookup, a match with any one of the two keys is considered to be a hit, provided

the physical tag maintained in the corresponding capability register also matches with the translation obtained using the TLB, thereby resolving potential homonym aliases. Thus, even though an L1 CRF probe could result in two conflicting matches across all banks, the physical tag matching step ensures that the conflict is resolved.

5.5. Temporal Sharing

In addition to being spatially shared by threads running concurrently on different logical/physical cores, cache resources may be temporally shared by threads transitioning in and out of the CPU through context switches. Upon a context switch, the incoming thread could potentially incur homonym aliases with outgoing thread. However, the physical tag matching step described above is expected to seamlessly resolve such aliases. Further, the incoming thread could potentially evict the cache lines of the outgoing thread if they belong to the same domain. Note that an eviction operation triggers the revocation of a capability, resulting in and entailing two key operations. First, the valid metadata bit of the capability is cleared. Second, the corresponding CRF entry is rekeyed to its initial invalid state, so that when the owner thread switches back into execution, future accesses to the line being evicted will be detected as misses.

5.6. Interplay with Hardware Prefetchers

A hardware prefetcher typically resides between two cache levels and learns patterns in the misses it observes. Based on the learned pattern, it issues prefetch requests that instruct the cache controller to speculatively bring missing lines into the cache, with the expectation that they will be used in the near future. The key distinction in Ceviche is that the miss patterns are made up of a sequence of physical address and domain ID pairs rather than physical addresses, and just like regular cache misses (i.e., those coming from the CPU), prefetch requests also go through the same capability-based allocation and lookup procedures, ensuring that prefetch requests pertaining to one domain do not result in the eviction of a line from another domain, preventing cross-domain leakage. We note that this does not necessarily prevent an attacker from mistraining speculative structures within the prefetcher to influence the prefetching behavior (and thus the cache timing behavior) of a victim. Similar to related work on secure cache designs [3], [28], [29], [30], [32], [33], [34], [35], [39], we consider the problem of designing secure prefetchers that are resilient against mistraining speculative structures within the prefetcher to be orthogonal, as our primary goal is to enable the secure virtualization of cache resources in a scalable manner among multiple domains.

6. Software Interface

Our flexible software interface augments the already well-established Intel's Cache Allocation Technology (CAT) [3], [83], which ensures priority-based shared hardware resource allocation. Similar to CAT, the processor in Ceviche exposes a predefined number of domains into which applications (or individual threads) can be assigned, where each domain is represented by a *domain_id* and for each logical processor, an MSR is exposed to allow the operating system to specify a domain when an application is scheduled for execution. The *domain_id* may change when a different thread is scheduled for execution after a context switch.

For each cache, we maintain two MSRs for storing the hard and soft limits respectively. These MSRs may be configured dynamically during runtime using privileged WRMSR instructions [84], [85], [86], [87], [88]. Furthermore, as mentioned in Section 4, sharing and invalidation policies are encoded in an MSR that maintains a bitmap of per-domain policies. While any given thread is allowed to modify this MSR through an unprivileged instruction exposed through the ISA, writes are restricted through a bitmask based on the domain ID, allowing updates to only those bits in the MSR pertaining to the thread's domain.

We also leverage Intel's Resource Director Technology feature that allows monitoring of shared hardware resources by tagging each core by a Resource Monitoring ID (RMID) [89], to monitor cache occupancy at all levels. In addition, similar to capacity bitmasks (CBM) exposed by Intel's CAT to enforce fixed size partitioning of the LLC across the different classes of service, we maintain programmer-invisible per-domain counter registers which maintain L1, L2, and LLC cache occupancy information, allowing us to enforce our fairness and availability guarantees through hard and soft limits as described in Section 5.2.

System software for Ceviche can leverage these existing interfaces, thereby ensuring that no additional developer effort is required. Moreover, the programmer-invisible counter registers can also be virtualized as appropriate. We recommend that the soft limit be set to $num_cache_lines/num_active_domains$ and the hard limit to slightly larger than the soft limit, but adjusted based on application priority and other constraints. Modern cloud-based virtualization solutions already use these strategies to improve utilization and quality-of-service metrics.

Finally, we expose instructions to the OS crash handler for gang revocation of capabilities that belong to a crashing domain. These instructions may be enabled or disabled through BIOS configuration.

7. Security Discussion

The rest of this section discusses how the security properties enforced by Ceviche protect against several known classes of cache attacks.

Conflict-Based Attacks. These attacks [12], [13], [14], [15], [16], [17], [18] rely on contending for shared cache resources with a co-resident victim to glean sensitive information. This involves constructing an eviction set of addresses that map to a particular cache set that the attacker and the victim contend for, thereby allowing the attacker to execute and time its own set of accesses, with the difference

in the hit/miss timing revealing information about a victim's data-dependent cache access behavior. Ceviche mitigates these attacks in two key ways. First, it thwarts eviction set construction by decoupling the address from the physical location (i.e., cache set). In particular, Ceviche mimics a fully-associative allocation strategy where any capability register may point to any line in the cache with no regard to the address, due to which the attacker has no control over the mapping of addresses to a particular location in the cache (like the limit case described by Vila, et al. [90]). Second, it prevents cross-domain conflicts by restricting a thread from evicting lines outside of its domain.

Attacks exploiting Cache Policies. Stealthier variants of cache attacks have been proposed that exploit replacement and write policies [91], [92], [93]. These attacks hinge on influencing the replacement states of a victim's cache line that shares a set with the attacker. In Ceviche caches, these attacks are not possible as cache replacement can occur only within a trust domain. The only exception to this is when a thread meets its hard allocation limit, but even in that case, the rate at which information is leaked is extremely small since we impose a strict limit that only one cross-domain eviction may occur within a 100,000 cycle interval, and the only information revealed that way was that a thread from a different domain was using the cache beyond the allowed soft limit, which is too coarse-grained to carry out practical attacks that exploit secret data-dependent behavior [13].

Cache Occupancy Attacks. These attacks seek to observe the overall cache occupancy of a victim without directly contending for particular cache sets, and further correlate cache occupancy information with certain sensitive attributes of the victim. For example, Shusterman et al. [5] describe a website fingerprinting attack that exploits the cache occupancy channel. Ceviche caches significantly mitigate cache occupancy attacks as follows. First, it imposes a hard limit on the number of capabilities that can be issued to (and thus, the number of lines that can be allocated to) any given thread. This prevents an attacker thread from learning about the occupancy in the rest of the cache outside of its hard limit. Second, once a thread reaches its soft allocation limit and the cache is full, it is further constrained to operate only within its soft limit, thereby not allowing it to glean any more information than the fact that the cache is full. Third, when the attacker thread hasn't reached its soft limit and the cache is full, to ensure fairness, we allow evictions to occur across domains, but as noted above, this process is deliberately slow, significantly limiting the attacker's ability to make accurate correlations regarding the victim's activity.

Reuse-Based Attacks. These attacks [19], [20], [21], [22], [94] leverage shared virtual memory (such as shared libraries or page deduplication), and the ability to flush a shared cache line by the virtual address. In this case, the attacker may observe victim accesses to the shared cache line by either timing flushes or reload operations to that line. Since the default policy in Ceviche is to disallow sharing of cache lines between two distrusting parties, these attacks are completely mitigated. Note that this does not necessarily mean that sharing is entirely disallowed. In fact, the default policy allows sharing within a domain, but disallows sharing of cache line across domains. However, this policy is configurable such that cross-domain sharing be allowed with the caveat that the lines are to be shared in a read-only fashion, with write and invalidate permissions turned off, as discussed in Section 4.

Confused-Deputy Attacks. Van Schaik, et al. [4] describe an indirect cache attack on systems employing way- or set-based cache partitioning, where an attacker abuses hardware modules such as the page table walker as confused deputies to access trusted cache partitions. These attacks are possible because the privileged entities acting as confused deputies access trusted partitions based on their own access rights on behalf of the attacker, even though the system's access control policy disallows the attacker from explicitly accessing trusted partitions. Ceviche thwarts some of these attacks through safe delegation of access rights via the per-domain capabilities contained in the CRF. This, for example, prevents an attacker from obtaining elevated access rights by abusing a trusted hardware or software module, as the cache lines that that module can access on behalf of the attacker are still limited by the capabilities granted to the attacker, rather than the capabilities granted to the trusted module. Note that, as mentioned in Section 3, loads and stores do not directly present a cache line capability, and instead the capability is derived from the effective address. While this ensures transparency, in the absence of memory capabilities (e.g., as enforced in CHERI [7]), one could deputize a privileged entity to flush shared lines within its domain by providing a rogue address. However, we ensure that cache lines are not arbitrarily shared or flushed across different domains unless explicitly allowed at allocation.

Denial-Of-Service Attacks. Finally, we note that Ceviche caches are resilient against denial-of-service attacks as no thread is allowed to allocate beyond its hard limit, and when the cache is full, it is further restricted to operate within its soft limit, which forms the basis of our minimum guarantee resource allocation policy.

New Side-Channels in Ceviche. The CAM-based CRFs are inherently fully-associative, so conflict-based attacks are not possible. Reuse-based attacks aren't possible either because there is no mechanism to flush individual CRF entries. There is also no contention or timing-dependent behavior for structures such as the free list due to the 1:1 mapping between the CRF and the number of cache lines.

8. Experimental Methodology

In this section, we describe our modeling assumptions and workload generation process used to evaluate the performance, power, and area overheads of Ceviche in single-core, multi-core, and SMT environments.

Performance Modeling. We use the Gem5 v20 [95] architectural simulator with the Ruby cache model for a detailed microarchitectural performance evaluation. We use a modern out-of-order superscalar processor with a multi-level cache hierarchy as our baseline, modeled after Intel's Icelake microarchitecture [96], [97] (shown in Table 3).

Frequency	3.3 GHz	Number of Cores:	1, 2, 4, 8, 12
ICache	32 KB, 8-way	DCache	48 KB, 12-way
Superscalar width	10 μops	ROB	352 entries
Register file	256 INT/FP	LQ/SQ	128/72 entries
L2 Cache	512 KB 8-way	L3 Cache	8 MB 16-way

TABLE 3: Baseline Configuration

	Latency Overhead	
	Baseline (ns/cycles)	Ceviche (ns/cycles)
L1-ICache	0.696/3	0.944/4
L1-DCache	1.218/5	1.551/6
L2Cache	2.184/8	2.617/9
LLC	4.606/16	5.105/17

TABLE 4: Latency Estimation

We use C and C++ benchmarks from SPEC CPU 2017 suite [98] to construct workloads based on the simpoint methodology [99]. Using PinPlay [100] and Gem5's check-pointing feature, we generate multiple simulation points that each span 100 million dynamic x86 instructions. While we use these simpoints directly for evaluating single-threaded workloads, for our scalability analysis on SMT and multi-core designs, we construct joint simpoints as follows. First, we classify the applications into CPU-intensive, memory-intensive, and balanced, based on the instruction-level parallelism (ILP) and memory behavior of each application. In particular, we group *wrf, exchange, nab* and *povray* into the CPU-intensive cluster; *mcf, fotonik, xalancbmk* and *xz* into the memory-intensive cluster, and *gcc, deepsjeng, leela* and *perl* into balanced cluster, per the characterization study by Panda, et al [101]. Second, we create joint checkpoints by considering simpoint combinations within a cluster. To make simulations tractable, we limit ourselves to simpoints that have the highest weight (i.e., the most representative region of the benchmark). We perform both multithreaded (2-way SMT) and multicore (2, 4, 8, and 12-core) simulations. For all the multi-core experiments, the LLC has a 75% hard limit and a 2-core design has 50% soft limit, 4-core has 25% soft limit and so on. For SMT experiments, all of the caches have a 75% and 50% hard and soft limit respectively.

For multicore and SMT experiments, we evaluate the performance of Ceviche against an insecure and a secure baseline in terms of normalized throughput (i.e., instructions committed per cycle across all cores), LLC cache miss rates, and LLC utilization. While the insecure baseline is modeled after the cache hierarchy in Intel's Icelake microarchitecture (Table 3), the secure baseline is implemented as follows. In a multi-core environment, the secure baseline is a statically partitioned LLC cache with N way-partitions, where N is the number of trust domains in the system. In our experiments, we assume the number of unique trust domains to be equal to the number of cores, and threads belong to different domains are scheduled on different cores. In an SMT environment, static way-partitioning is introduced at all levels of the cache wherein the number of way partitions is set to two, the number of hardware threads within each physical core. We ensure that all threads always perform the same amount of work by limiting the number of instructions executed by each thread to 10 million dynamic instructions.

Note that the default cache sizes in Ceviche are consistent with commercial XeonE-21X4/22X4 processors, although we conservatively choose a fixed 8MB LLC for all configurations to highlight the impact of LLC contention among memory-intensive applications in our multithreaded workloads. In our scalability experiments, we find that Ceviche performs significantly better on larger LLCs.

Power, Area, and Latency Estimation. We use CACTI-P [26] to estimate the area, power, latency overheads associated with Ceviche. We use the 32nm technology node to model our memory structures owing to its maturity and reliability [32], [102], [103]. We perform RTL analysis using Synopsys DC compiler [104] to estimate the latency, power, and area overheads entailed by the victim search logic and the free list. CACTI can model memory structures as a CAM, SRAM (cache), or DRAM (main memory) at various process nodes with specified ports and banks. The CRF is modeled as a CAM structure in CACTI and the virtual address tag and domain ID is used as the key for the look up. We bank all of the structures similar to the baseline's SRAM-tagged caches (i.e., 4 banks for L1I and L1D, 16 banks for L2, and 32 banks for L3). Ceviche's data array is modeled as a direct-mapped multi-banked data array. We also use the standard low power (LOP) cells from the International Technology Roadmap for Semiconductors (ITRS) for the CAM-tagged cache implementation as assumed in prior works [8], [9], [10], [11]. Finally, since CACTI does not allow modeling of caches with an associativity that is not a power of two, we estimate the latency, power, and area overheads of our baseline 48KB 12-way set-associative L1 data cache from Intel's Icelake using a average of 32KB 8-way and 64KB 16-way set-associative caches, albeit skewed slightly towards the larger cache, accounting for the complexity entailed by the non-power of two associativity. Table 4 shows the latencies computed using CACTI for both the baseline and Ceviche. Overall, due to our CAM-tagged implementation we incur one extra cycle latency across all levels of the cache hierarchy.

Furthermore, modern LLCs are sliced in that they are typically divided into as many slices as there are number of cores sharing the LLC, and each slice is identified by means of a hash calculation. Our CACTI analysis shows that thirty-two L3 CRF banks (each holding 4096 16-bit L3-capabilities) may be looked up in parallel within the extra cycle added to the hit latency (Table 4), providing access to (and enabling flexible partitioning among) a cluster of four 2MB slices. For larger caches, more banks are needed; thus, a hash of the address/domain-ID pair would have to determine the slice cluster and its corresponding CRF. For e.g., Xeon-Platinum-83XX (32-40 cores/48-60MB LLC; beefier than Intel-12900k) would feature eight multi-banked L3 CRFs, each serving a cluster of four 1.5MB slices, but only one CRF is looked up for any given L3 access. The hash computation to identify the slice cluster would not involve ciphers, and hence no additional latency is incurred to access a slice cluster.

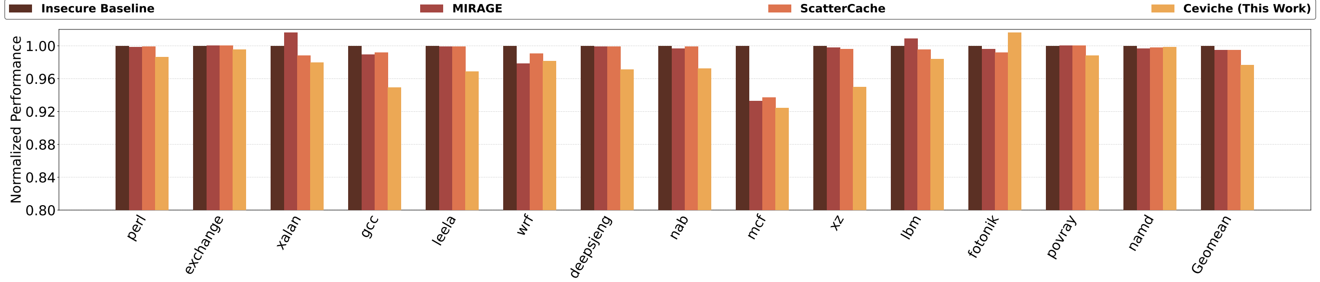

Figure 5: Single-Threaded Performance Comparison

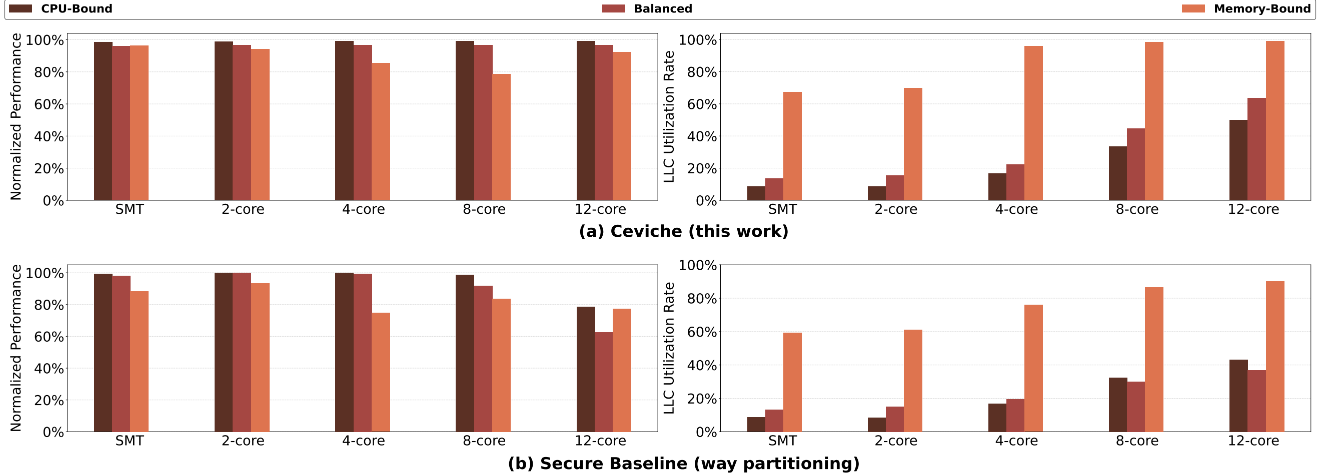

Figure 6: Performance Comparison for SMT/Multicore

9. Results

In this section, we first present results from our experimental evaluation in a single-core environment and then present our scalability analysis on SMT and multicore designs, in addition to discussing the storage, area, and power overheads of Ceviche.

9.1. Single Core Environment

Figure 5 shows the performance (IPC) of Ceviche and two state-of-the-art defenses, Mirage [32] and ScatterCache [31], normalized to the insecure baseline. We make several key observations. First, the benchmarks that suffer the most are the memory-intensive ones such as *mcf* (maximum degradation of 7.8%) that spends most of its time in a pointer-chasing loop with irregular memory accesses. These benchmarks get penalized primarily due to the increased hit latency of CAM-tagged caches in Ceviche and cipher-based set-index computation for the last-level caches in Mirage and ScatterCache. Second, not all memory-intensive benchmarks observe a degradation. For example, we observe a speedup in *fotonik* due to a substantial uptick in the hit rates at the L2 and L3 caches, owing to the conflict-averse fully-associative allocation policy in Ceviche. Third, the memory accesses from applications exhibiting high instruction-level parallelism such as *perlbench*, *exchange*, *namd*, and *povray* are mostly L1-bound and even those accesses typically tend to get resolved in the processor through store-to-load forwarding, due to which they hardly incur any overhead. Fourth, benchmarks such as *gcc*, *xz*, and *leela* suffer a slight performance degradation due to pollution in the instruction cache, caused as a side effect of the fully-associative allocation strategy in Ceviche, coupled with the already high branch misprediction rate in these benchmarks forcing them to fetch unnecessary cache lines along mispredicted paths. Mirage and ScatterCache don't incur these overheads as they don't protect the private data and instruction caches.

On average, Ceviche incurs only 2.4% slowdown over the insecure baseline that does not implement any protections against cache attacks, while being able to provide substantial security guarantees. In comparison to Mirage and ScatterCache, we impose only an additional 1.8% degradation in performance, while protecting all levels of caches rather than just the last-level cache.

9.2. Multicore and SMT Environment

To evaluate the scalability of Ceviche to multiple trust domains, we construct multiprogrammed mixed workloads as described in Section 8, such that each program within the workload belongs to a different trust domain and further schedule them to different hardware threads (in the SMT design) or cores (in the multicore design). Figure 6 shows

		Storage Overhead (in KB)		Area Overhead (in mm^2)		Power Overhead (in W)	
		Baseline	**Ceviche**	**Baseline**	**Ceviche**	**Baseline**	**Ceviche**
L1 ICache	Tag	2.25	0	0.019	0	0.006	0
	CRF	0	9.063	0	0.079	0	0.030
	Free List	0	1.125	0	0.011	0	0.03
	Data Array	32		0.173		0.046	
L1 DCache	Tag	3.375	0	0.109	0	0.023	0
	CRF	0	13.594	0	0.221	0	0.085
	Free List	0	1.688	0	0.015	0	0.036
	Data Array	48		0.663		0.117	
L2 Cache	Tag	32	0	0.124	0	0.010	0
	CRF	0	106	0	0.406	0	0.074
	Free List	0	20	0	0.13	0	0.164
	Data Array	512		1.997		0.360	
LLC	Tag	464	1152	1.519	0	0.058	0
	CRF	0	9.063	0	3.577	0	0.278
	Free List	0	416	0	0.251	0	0.3
	Data Array	8192		13.998		0.125	
Total Overhead		**1227 KB**		**2.919 mm^2**		**0.9 W**	

TABLE 5: Raw Storage, Area and Power Estimates

Cores /LLC Size	Baseline Storage (KB)	Ceviche Storage (KB)	Storage Overhead	Baseline Area (mm^2)	Ceviche Area (mm^2)	Area Overhead	Baseline Power (W)	Ceviche Power (W)	Power Overhead
4C/8MB	8807	10366	17.70%	130.5	135.0	4%	685.0	686.8	0.20%
8C/16MB	17613	20668	17.30%	258.2	267.7	3.60%	1366.3	1369.9	0.30%
16C/32MB	35226	41336	17.30%	513.1	531.6	3.60%	2933.6	2938.5	0.20%
32C/64MB	70452	82761	17.30%	1026.3	1063.1	3.60%	5867.2	5877.0	0.20%
64C/256MB	279400	320990	14.80%	4105.1	4252.6	3.60%	23468.8	23507.9	0.20%

TABLE 6: Raw Storage, Area and Power Impact with Core/Cache Scaling

Domains	32	256	1024	4096
L1-I	9.06 KB	9.25 KB	9.38 KB	9.5 KB
L1-D	13.59 KB	13.8 KB	14.06 KB	14.25 KB
L2	106 KB	109 KB	111 KB	113 KB
L3	1152 KB	1200 KB	1232 KB	1264 KB
Total	1280.65 KB	1332.05 KB	1366.44 KB	1400.75 KB

TABLE 7: Storage Impact with Domain Scaling

normalized throughput and utilization of the LLC of these workloads on a single-core SMT, 2-core, 4-core, 8-core, and 12-core designs for Ceviche (top) and our secure baseline that performs way partitioning (bottom). The utilization of the LLC is computed as the average number of valid cache lines in the LLC over the duration of the experiment.

We observe that both Ceviche and our secure baselines with way partitioning perform only slightly worse than the insecure baseline with no protections for the CPU-bound and balanced workloads, and for the memory-bound workloads in SMT and 2-core scenarios. However, as we scale to 4, 8, and 12 cores, we see that the performance of the way partitioning design starts to incur prohibitively high degradation. This is because partitioning at the coarse granularity of cache ways is inherently prone to high LLC miss rates and low overall utilization as the number of trust domains is scaled beyond a certain point. For instance, the LLC is utilized by 25% more in Ceviche as opposed to way-partitioning in the 8-core design, and even when the LLC is underutilized in the 2-core and SMT scenarios due to the other cores being idle, we still observe lower utilization and degradation in the secure baseline with way partitioning. Further, while Ceviche is able to continue to maintain a low performance degradation for the CPU-bound and balanced workloads in 8-core and 12-core scenarios, the way partitioning baseline starts to degrade drastically due to fewer cache ways available for allocation per-thread. Finally, way partitioning also incurs greater performance degradation in the SMT scenario for the memory-intensive workload as not just the LLC is partitioned equally, but even the private caches get partitioned into two to cater to the two SMT threads, effectively halving cache capacity across all levels.

On the other hand, Ceviche degrades much gracefully due to its fine-grained partitioning solution and competitive allocation policy that together not only removes the restriction on partitions having to include contiguous lines in a way, but also allow partitions to grow beyond their configured soft limits when space is available, thereby achieving low miss rates and high utilization rates that are comparable to the insecure baseline. Also note that, although the overall performance increases with more cores, per-core IPC significantly decreases due to LLC contention among memory-intensive workloads. Hence, Ceviche experiences a smaller slowdown as the cache size increases.

9.3. Power, Area, and Storage Overheads

The storage overheads in Ceviche caches arise due to the additional free list and the extra bits in the CRF – address tags, domain ID, metadata bits, and capabilities containing the cache line number and access rights (shown in Figure 3. Table 5 provides a detailed breakdown incurred due to these additional tagging and metadata across the 3-level cache hierarchy. It should be noted that the storage

size of the data arrays remain constant for both baseline and Ceviche. Overall, across all levels, Ceviche consumes 13.2% more storage than the insecure baseline, in comparison to a recent secure LLC design, Mirage [32], that adds 17-20% extra tag storage. Further, we measure the area overhead imposed by the combinational logic used for capability-based lookup and replacement to be 2.3 mm^2 with negligible impact on clock latency. The CAM-tagged design further adds 2.9 mm^2 overhead and 0.9 W in power, which is negligible in comparison to the overall chip area (tens of square millimeters) and power (tens of Watts) in most modern processors [32]. Furthermore, we would observe some savings in power and area due to our decision to use a direct-mapped data array in place of a set-associative array. This savings would come from the reduction in the multiplexing logic and conflict resolution logic that could be found in a set-associative data array. Table 5 also provides the raw storage, power, and area overheads incurred due to the additional structures we add.

Table 6 shows the area, storage, and power overhead with increase in the number of cores and the size of the LLC. Note that the area and power analysis includes the cumulative area and power consumed by the processor, memory, peripherals, and the cache subsystem. The area and the power overheads stay consistent even after scaling the number of cores and the LLC size. Similarly, Table 7 shows the storage overhead while scaling the number of domains supported by Ceviche. We find that the storage overhead scales linearly with the number of domains.

10. Conclusion

This work introduces Ceviche, a novel virtualization solution for caches that offers protection against conflict-based, reuse-based, occupancy-based, confused deputy, and denial-of-service attacks. The key to Ceviche is a secure virtualization layer that translates an address-domain ID pair into a capability that encodes the access rights and the permitted set of operations on a cache line, enforcing the principle of least privilege. By decoupling the address from the physical location of a cache line, Ceviche enables fully associative allocation and fine-grained cache partitioning, maximizing utilization and amortizing the virtualization cost. Ceviche incurs an average slowdown of 2.4% over an insecure baseline, while gracefully scaling to multiple domains, outperforming way partitioning-based solutions.

Acknowledgments

The authors would like to thank the anonymous reviewers and the Shepherd for their insightful suggestions and comments. This research was supported by NSF grants CCF-2238548 and CNS-2213700, and a DARPA I2O cooperative agreement FA8750-24-2-0002.

References

[1] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: a dynamic cache partitioning system using page coloring," in *PACT*, 2014.

[2] K. T. Nguyen, "Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family," 2016, https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html.

[3] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing," in *HPCA*, 2016.

[4] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious management unit: Why stopping cache attacks in software is harder than you think," in *USENIX Security*, 2018.

[5] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 639–656.

[6] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *IEEE*, 1975.

[7] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *ISCA*, 2014.

[8] P. J. Wilson, "Cache organization and method," 2018, uS Patent 10,025,720.

[9] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanovic, "Direct addressed caches for reduced power consumption," in *MICRO*, 2001.

[10] M. Zhang and K. Asanovic, "Highly-associative caches for low-power processors," in *Kool Chips Workshop, MICRO*, 2000.

[11] M. Zhang and K. Asanović, "Fine-grain cam-tag cache resizing using miss tags," in *ISLPED*, 2002.

[12] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *CCS*, 2015.

[13] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks Are Practical," in *S&P*, 2015.

[14] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: A timing attack on openssl constant time rsa," in *International Conference on Cryptographic Hardware and Embedded Systems*, 2016.

[15] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX," in *USENIX Security*, 2017.

[16] O. Aciiçmez, "Yet Another Microarchitectural Attack:: Exploiting I-cache," in *ACM Workshop on Computer Security Architecture (CSAW)*, 2007.

[17] C. Percival, "Cache missing for fun and profit," 2005.

[18] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[19] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack." in *USENIX Security*, 2014.

[20] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *S&P*, 2011.

[21] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *CCS*, 2014.

[22] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive {Last-Level} caches," in *USENIX Security*, 2015.

[23] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, 2009.

[24] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," *International Symposium on Computer Architecture (ISCA)*, 1995.

[25] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *International Symposium on Computer Architecture (ISCA)*, 1996.

[26] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *ICCAD: International Conference on Computer-Aided Design*, 2011, pp. 694–701.

[27] A. González, F. Latorre, and G. Magklis, *Processor Microarchitecture: An Implementation Perspective*, 12 2010, vol. 5.

[28] F. Liu and R. B. Lee, "Random Fill Cache Architecture," in *ISCA*, 2014.

[29] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," in *MICRO*, 2016.

[30] M. K. Qureshi, "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping," in *MICRO*, 2018.

[31] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: thwarting cache attacks via cache set randomization," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 675–692.

[32] G. Saileshwar and M. K. Qureshi, "Mirage: Mitigating conflict-based cache attacks with a practical fully-associative design," in *USENIX Security Symposium*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:221819421

[33] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks," in *ISCA*, 2007.

[34] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: secure dynamic cache partitioning for efficient timing channel protection," in *DAC*, 2016.

[35] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks," *TACO*, 2012.

[36] "SecSMT: Securing SMT processors against Contention-Based covert channels," in *USENIX Security*, 2022.

[37] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 42–56. [Online]. Available: https://doi.org/10.1145/3352460.3358310

[38] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," in *MICRO*, 2018.

[39] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 347–360.

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

[41] D. Townley, K. Arıkan, Y. D. Liu, D. Ponomarev, and O. Ergin, "Composable cachelets: Protecting enclaves from cache Side-Channel attacks," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2839–2856. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/townley

[42] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," 2012.

[43] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," in *MICRO*, 2011.

[44] F. Lombardi and R. Di Pietro, "Secure virtualization for cloud computing," *Journal of network and computer applications*, 2011.

[45] J. Criswell, B. Monroe, and V. Adve, "A virtual instruction set interface for operating system kernels," in *Workshop on the Interaction between Operating Systems and Computer Architecture*, 2006.

[46] J. Criswell, N. Geoffray, and V. S. Adve, "Memory safety for low-level software/hardware interactions." in *USENIX Security*, 2009.

[47] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *ACM SIGPLAN Notices*, 2014.

[48] M. Pearce, S. Zeadally, and R. Hunt, "Virtualization: Issues, security threats, and solutions," *ACM Computing Surveys (CSUR)*, 2013.

[49] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[50] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[51] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *ASPLOS*, 1994.

[52] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *CCS*, 2013.

[53] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *Security and Privacy (SP), 2015 IEEE Symposium on*, 2015.

[54] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. Son *et al.*, "Cheri jni: Sinking the java security model into the c," in *ASPLOS*, 2017.

[55] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. Watson, D. Chisnall, M. Roe, B. Davis *et al.*, "Efficient tagged memory," in *ICCD*, 2017.

[56] B. Davis, R. N. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka *et al.*, "Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the posix c run-time environment," in *ASPLOS*, 2019.

[57] R. Sharifi and A. Venkat, "Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *ISCA*, 2020.

[58] D. A. Osvik, A. Shamir, and E. Tromer, *Cache Attacks and Countermeasures: The Case of AES*, 2006. [Online]. Available: https://doi.org/10.1007/11605805_1

[59] D. J. Bernstein, "Cache-timing attacks on aes," 2005. [Online]. Available: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[60] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," in *ISCA*, 2021.

[61] M. Taram, A. Venkat, and D. Tullsen, "Packet Chasing: Spying on Network Packets over a Cache Side-Channel," in *ISCA*, 2020.

[62] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," Apr. 2016, arXiv:1511.04594 [cs]. [Online]. Available: http://arxiv.org/abs/1511.04594

[63] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 168–179.

[64] F. Yao, M. Doroslovaki, and G. Venkataramani, "Covert timing channels exploiting cache coherence hardware: Characterization and defense," *International Journal of Parallel Programming*, vol. 47, pp. 595–620, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:53873984

[65] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *ASPLOS*, 2015.

[66] F. Yao, H. Fang, M. Doroslovački, and G. Venkataramani, "Cotsknight: Practical defense against cache timing channel attacks using cache monitoring and partitioning technologies," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 121–130.

[67] M. K. Qureshi, "New Attacks and Defense for Encrypted-Address Cache," in *ISCA*, 2019.

[68] F. Liu, H. Wu, and R. B. Lee, "Can randomized mapping secure instruction caches from side-channel attacks?" in *HASP*, 2015.

[69] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "Phantomcache: Obfuscating cache conflicts with localized randomization." in *NDSS*, 2020.

[70] S. Deng, W. Xiong, and J. Szefer, "Analysis of secure caches using a three-step model for timing-based attacks," Cryptology ePrint Archive, Paper 2019/167, 2019. [Online]. Available: https://eprint.iacr.org/2019/167

[71] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, 1988.

[72] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port Contention for Fun and Profit," in *S&P*, 2019.

[73] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," in *CCS*, 2019.

[74] Z. Wang and R. B. Lee, "Covert and Side Channels Due to Processor Architecture," in *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[75] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," in *ASPLOS*, 2018.

[76] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking Branch Predictors to Bypass ASLR," in *MICRO*, 2016.

[77] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference*, 2007.

[78] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security*, 2018.

[79] K. C. Yeager, "The mips r10000 superscalar microprocessor," *IEEE micro*, 1996.

[80] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security*, 2018.

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

[82] M. Gupta, V. Sridharan, D. Roberts, A. Prodromou, A. Venkat, D. Tullsen, and R. Gupta, "Reliability-aware data placement for heterogeneous memory architecture," in *HPCA*, 2018.

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

[84] M. Taram, A. Venkat, and D. Tullsen, "Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency," in *ISCA*, 2018.

[85] Taram, Mohammadkazem and Venkat, Ashish and Tullsen, Dean M, "Context-sensitive decoding: On-demand microcode customization for security and energy management," *IEEE Micro*, 2019.

[86] Taram, Mohammadkazem and Venkat, Ashish and Tullsen, Dean M., "Context-sensitive fencing: Securing speculative execution via microcode customization," in *ASPLOS*, 2019.

[87] M. Taram, A. Venkat, and D. Tullsen, "Mitigating speculative execution attacks via context-sensitive fencing," *IEEE Design & Test*, 2022.

[88] M. Taram, D. Tullsen, A. Venkat, H. Sayadi, H. Wang, and H. Homayoun, "Fast and efficient deployment of security defenses via context sensitive decoding," in *GOMACTech*, 2019.

[89] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, and O. Krieger, "A closer look at intel resource director technology (rdt)," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, ser. RTNS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 127–139. [Online]. Available: https://doi.org/10.1145/3534879.3534882

[90] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 39–54, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52916873

[91] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "{RELOAD+ REFRESH}: Abusing cache replacement policies to perform stealthy cache attacks," in *USENIX Security 20*, 2020.

[92] W. Xiong and J. Szefer, "Leaking information through cache lru states," in *HPCA*, 2020.

[93] Y. Cui, C. Yang, and X. Cheng, "Abusing cache line dirty states to leak information in commercial processors," in *HPCA*, 2022.

[94] M. C. W. K. Gruss, Daniel and S. Mangard, *Flush+Flush: A Fast and Stealthy Cache Attack*, 2016. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_14

[95] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[96] I. Corporation, "Intel icelake - products formerly ice lake." [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/codename/74979/products-formerly-ice-lake.html

[97] L. Moody, W. Qi, A. Sharifi, L. Berry, J. Rudek, J. Gaur, J. Parkhurst, S. Subramoney, K. Skadron, and A. Venkat, "Speculative code compaction: Eliminating dead code via speculative microcode transformations," in *MICRO*, 2022.

[98] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, 2006.

[99] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS*, 2002.

[100] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *CGO*, 2010.

[101] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a decade: Did spec cpu 2017 broaden the performance horizon?" in *HPCA*. IEEE, 2018.

[102] A. Venkat and D. M. Tullsen, "Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor," in *ISCA*, 2014.

[103] A. Venkat, H. Basavaraj, and D. M. Tullsen, "Composite-isa cores: enabling multi-isa heterogeneity using a single isa," in *HPCA*, 2019.

[104] Synopsys, "Design compiler." [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper proposes a novel cache organization that thwarts a wide-range of attacks.

A.2. Scientific Contributions

- Provides a valuable step forward in an established field.
- Addresses a long-known issue.

A.3. Reasons for Acceptance

1) The proposal of Ceviche is quite novel and different from existing partition and randomization schemes.
2) Protect against a type of confused deputy attacks.
3) The performance overhead is reasonable compared to non-secure baseline.
4) The writing is clear and easy to read.