ElaCache: Fine-Grain Dynamic Partitioning of LLCs and Coherence Directories in Multiprocessors

Dingyuan Cao
University of Illinois, Urbana Champaign
Urbana, IL, USA
dc29@illinois.edu

Zirui Neil Zhao
University of Texas at Austin
Austin, TX, USA
neil.zhao@utexas.edu

Adam Morrison
Tel Aviv University
Tel Aviv, Israel
mad@cs.tau.ac.il

Josep Torrellas
University of Illinois, Urbana Champaign
Urbana, IL, USA
torrella@illinois.edu

Abstract

Partitioned last-level caches (LLCs) offer comprehensive security guarantees against LLC side-channel attacks. However, current designs have performance and scalability limitations. On the one hand, static partitioning schemes cannot adapt partition sizes to changing application demands; on the other hand, current dynamic partitioning designs support only coarse-grained partitions or support finer-grained partitioning only with software cooperation.

To address these limitations, this paper proposes the *Elastic Cache* or *ElaCache*, a new design for dynamic partitioning of the LLC and directory. ElaCache supports fine-grained partitions of both sets and associative ways without any software intervention. It uses a novel micro-architecture design that manages and resizes its partitions fully in hardware, leveraging on-line profiling. In addition, ElaCache provides a leakage bound that can be tuned to balance security and performance. We evaluate ElaCache with simulations of a 16-core processor running a mix of applications. Our results show that ElaCache delivers high performance while providing strong security guarantees. For example, when only one out of 8 running applications needs a large cache footprint, ElaCache improves the IPC of that application by 64.3% on average over static cache partitioning. Further, with 1 large-footprint application out of 16, the average IPC improvement is 82.2%. In both cases, the performance of the rest of the applications in the mix is not noticeably penalized. Finally, ElaCache bounds information leakage to a modest level.

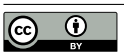
CCS Concepts

• **Security and privacy → Side-channel analysis and countermeasures**.

Keywords

Cache side-channel defense, dynamic cache partitioning

ACM Reference Format:
Dingyuan Cao, Zirui Neil Zhao, Adam Morrison, and Josep Torrellas. 2026. ElaCache: Fine-Grain Dynamic Partitioning of LLCs and Coherence Directories in Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19–22, 2026, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3838684.3846883



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846883

1 Introduction

Last-level cache (LLC) side-channel attacks (e.g., [16, 20, 22, 25, 27, 32]) are among the most powerful side-channel attacks due to their ease of deployment and stealthy operation. In a non-inclusive LLC setting, the same attacks can be mounted on the coherence directory [52], which contains entries for all the lines present in the cache hierarchy. These attacks can leak the access patterns of victim programs, revealing secrets such as encryption keys [16, 20, 22, 27] or fingerprinting application characteristics [44].

Side-channel-resistant LLC designs can be categorized as randomized [17, 33, 34, 39, 45, 51] or partitioned [1, 8, 14, 26, 38, 48, 50]. Randomized LLCs randomize the address-to-cache set mapping, blocking attackers from using set-conflict eviction techniques [27, 32] to identify cache sets and monitor access patterns. Partitioned LLCs isolate different security domains to distinct physical cache partitions, so attackers cannot monitor the cache effects of victim accesses.

Unfortunately, no existing approach simultaneously offers comprehensive side-channel protection, scalability to many security domains, and low performance overhead. Indeed, randomized LLCs lack comprehensive security guarantees, and are susceptible to newer attacks targeting them [7, 10, 11, 31, 45, 49]. Partitioned LLCs offer comprehensive security guarantees (i.e., non-interference across partitions), but have performance and scalability limitations. On the one hand, *static* partitioning cannot adapt the partition sizes to changing application demands, and thus delivers limited performance. On the other hand, while *dynamic* partitioning can adapt to the needs of the applications, existing designs have shortcomings: they support only coarse-grained partitions (e.g., a partition is one or more LLC associative ways) or support finer-grained partitioning only with software cooperation.

To address these limitations, this paper proposes the *Elastic Cache* or *ElaCache*, a new design for dynamic partitioning of the LLC and directory. ElaCache supports fine-grained partitions of both sets and associative ways (i.e., 2D partitioning) for performance and scalability. It manages its partitions in hardware, transparently from the software, using profiling-based automatic resizing. To our knowledge, ElaCache is the first general design that supports both software-free and fine-grained dynamic partitioning.

Partitions have highly-flexible sizes and are managed by a novel micro-architecture.

In ElaCache, the only possible source of information leakage is due to the resizing of the victim partition driven by the behavior of the victim program. Crucially, however, this leakage can be comprehensively analyzed and a bound for it can be quantified. Such a leakage bound can be used to trade-off security for performance, or to guarantee security. For example, ElaCache's parameters can be set to reduce potential leakage at some performance cost. Further, the OS can monitor when an application exhausts its leakage budget and then block additional leakage by disabling further dynamic resizing [2, 15, 55, 56].

We evaluate ElaCache with simulations of a 16-core processor running a mix of applications and with either 8 or 16 partitions per LLC slice. Our results show that ElaCache delivers high performance while providing strong security guarantees. For example, when only one out of 8 running applications needs a large cache footprint, ElaCache improves the IPC of that application by an average of 64.3% over a static cache partitioning scheme. Further, for an environment with one large-footprint application out of 16, the IPC improvement is 82.2%. In both cases, the performance of the rest of the applications in the mix is not noticeably penalized. Finally, ElaCache bounds information leakage to a modest level.

Overall, the contributions of this paper are:

- The ElaCache approach to dynamic, fine-grained, and software-transparent partitioning of LLC and directory.
- A detailed micro-architectural design of ElaCache, including mechanisms for partition resizing based on runtime profiling.
- A security analysis of ElaCache leakage.
- An evaluation of ElaCache.

2 Motivation

Side-channel-resistant LLC designs fall into two main categories: randomized LLCs [17, 33, 34, 39, 45, 51] and partitioned LLCs [1, 8, 14, 26, 38, 42, 48, 50]. Randomized LLCs randomize how memory addresses are mapped to cache sets. These designs reduce LLC side-channel risks by making it harder for the attacker to induce conflicts in targeted cache sets—disrupting the ability to monitor the victim's memory access patterns using attacks like Prime+Probe [27]. On the other hand, partitioned LLCs isolate different security domains to physically distinct cache partitions, restricting attackers to only impact and observe the cache state of their own partitions. Sadly, no existing approach simultaneously offers a comprehensive security guarantee that is applicable to any possible attack, a low performance overhead, and scalability to many security domains.

Lack of Comprehensive Protection. Randomized LLCs offer good performance, but their security guarantees are often tailored to specific attack strategies. As such, more advanced cache attacks can break their security guarantees. For example, CEASER's [33] security guarantee is based on a sub-optimal Prime+Probe attack strategy and was defeated by an improved strategy [34] after only a few months. CEASER-S [34] and ScatterCache [51] blocked the improved attack, but were later found susceptible to even more advanced attacks [7, 10, 31, 45]. This prompted the development of MIRAGE [39]. However, MIRAGE only blocks set-conflict attacks and is still vulnerable to cache occupancy attacks. SassCache [17] addresses the cache occupancy attack by adopting a hybrid design that combines randomization and partitioning. However, the security guarantee of SassCache is based on the assumption that different security domains do not collude. Unfortunately, an attack by colluding domains increases its success rate quickly as the number of colluding domains increases. For example, an attacker cannot build an eviction set, which is required by many cache attacks, for 99.99997% of victim cache lines in SassCache's default setting. But the protection quickly degrades to only 91.05% and 27.62% of victim cache lines when there are 4 and 8 colluding attackers, respectively. In a public cloud setting, multi-domain collusion can be achieved by co-locating multiple attacker VMs owned by distinct accounts, using known techniques [20, 36, 57].

Table 1: LLC partitioning with comprehensive security guarantees.

Partition Grain Size	Static	Dynamic with Annotation	Dynamic No-Annotation
Coarse Grain (Ways)	Intel CAT [21], CATalyst [26]		UMON [35], SecDCP [50], M-Party [24]
Medium Grain (Sets)	BCE [38], MI6 [8]	Chunked-Cache [14]	
Fine Grain (Sets+Ways)		CC [48], TEE-SHirT [1]	**ElaCache (This Work)**

Performance and Scalability Limitations. Partitioned LLCs can offer comprehensive security guarantees, but have performance and scalability limitations. Static partitions [8, 21, 26, 38] have zero leakage, as distinct security domains are fully isolated in physically-separated partitions. However, static partitioning often limits performance, because partition sizes cannot adapt to changing application demands.

Dynamic partitioning improves performance by resizing partitions based on application behavior [50]. However, it does not guarantee zero leakage and current designs have scalability limitations. To understand the scalability limitations, we describe the existing dynamic partitioning schemes in the next paragraph. Regarding the leakage, dynamic partitioning leakage can be comprehensively analyzed and quantified (Section 5). Despite the potential leakage, security can be guaranteed by monitoring when an application exhausts its designated "leakage budget" and then either (1) disable further dynamic resizing of its partition or (2) signal to refresh secrets such as ephemeral cryptographic keys [2, 15, 55, 56].

Dynamic Partitioning Schemes. Table 1 classifies schemes for static or dynamic partitioning. We only consider designs that offer comprehensive security guarantees. For example, we exclude Vantage [40], which provides automatic dynamic partitioning but partitions only a portion of the LLC, falling short of comprehensive security.

The Y axis classifies the schemes based on the grain size of the partitions: *coarse* if the cache is partitioned only across ways, *medium* if it is partitioned only across groups of sets, and *fine* if it is partitioned across both groups of sets and ways. The X axis classifies the schemes based on whether the partitions are static or dynamic and, if the latter, whether they require software annotations to drive the partition changes.

For highest effectiveness, dynamic partitioning schemes must have three characteristics. First, they must scale to support many

partitions, to accommodate the large number of security domains in cloud servers. This suggests supporting fine-grained partitions across both cache sets and ways (i.e., 2D) [48].

Second, dynamic partitioning should operate in a software transparent manner, relying entirely on hardware actions rather than on annotations by the software. Indeed, adding software annotations is difficult and error-prone for programmers. Moreover, with annotations, each application requests resizings selfishly, based only on its own demand. Hence, total demand across applications can exceed the cache capacity and prevent a globally-optimal cache allocation. Currently, however, the designs that support fine-grained partitions [1, 48] rely on software-driven resizing decisions.

The third characteristic is that the partition sizes and the remapping mechanisms should be as flexible as possible. Some current designs are inflexible, as a partition must use a power-of-two number of sets or sets-ways chunks, and they do not detail the mechanism of remapping cache lines during a resizing [1, 14, 48].

To address these limitations, we propose *ElaCache*. ElaCache supports many fine-grained 2D partitions, which it resizes transparently from the software, using on-the-fly profiling hardware and a novel micro-architecture. Partitions have flexible sizes. Finally, ElaCache provides a leakage bound that can be tuned to balance security and performance.

3 Threat Model

We consider a scenario where multiple mutually distrusting domains run concurrently. Each domain executes its own processes on one or more cores, and owns one partition in each of the LLC and directory slices. For LLC partitioning to be meaningful, we assume that the system prevents side-channel attacks on core-private resources, such as private caches and TLBs, using orthogonal mechanisms. For example, AWS EC2 [3] assigns security domains to dedicated cores for their lifetime—except on burstable instance types where core-private microarchitectural state is flushed when the core context switches between security domains. The LLC slice hash function is fixed, and all the LLC slices are accessible to all the domains. In each slice, partitions grow or shrink dynamically and independently of what happens in other slices. Interference other than cache conflicts (e.g., network [13, 29] or port contention [42]) is out of scope. Frequency and power side channels [12, 19] are also out of scope. Each domain can only access its own partition in the multiple LLC and directory slices. For the security analysis in Section 5, we assume the worst case: the attacker is capable of using all the domains other than the victim one [53], and is thus capable of precisely observing the victim's partition size changes. Capacity attacks [43, 44] through partition size changes are within the scope of this work and are analyzed in Section 5.

4 Design

4.1 Overview

The goal of ElaCache is to support multiple security domains by dynamically dividing each slice of a distributed LLC and directory into multiple partitions in a scalable manner. Each partition includes a certain number of sets and ways, and changes size dynamically. The smallest unit of cache space re-assignment across partitions is a *Resizing Unit* (RSU). To visualize an RSU, assume that one way of an

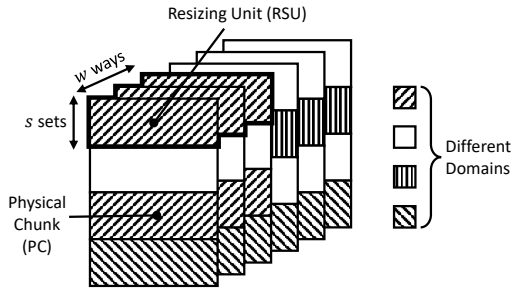


Figure 1: Partitioned LLC slice in ElaCache.

LLC slice is logically divided into groups of s contiguous sets called *Physical Chunks* (PCs). An RSU is a PC of w contiguous ways. A partition is composed of multiple RSUs that can be noncontiguous in the Y dimension (sets) and Z dimension (ways). Figure 1 shows an LLC slice with 4 domains.

For simplicity, most of our discussion is focused on partitioning the LLC. However, ElaCache also partitions the coherence directory. The directory has two array structures per slice: the Traditional Directory (TD) associated with the LLC and an Extended Directory (ED) [53] (also called snoop filter) that contains sharer information on lines cached only in private caches. The ED is needed in systems with non-inclusive caches. Figure 1 shows an LLC slice (implicitly including its TD) but can also represent an ED slice.

ElaCache uses a per-domain translation table that maps physical addresses to entries in the LLC partition owned by the domain. Further, to competitively partition the LLC among multiple domains dynamically, all domains periodically profile their LLC miss counts at their current partition sizes. They also estimate what their LLC miss counts would be at larger and smaller partition sizes *concurrently*. Then, the hypervisor uses this information to resize the partitions competitively. On a resize, remapping operations occur: the ElaCache translation tables are changed and some lines in the LLC are migrated to their correct new position. The same mechanisms are enabled for the ED.

4.2 Mapping Logical to Physical Chunks

To operate, ElaCache performs an extra translation of the index bits of the physical address (PA) as shown in Figure 3. Chart (a) shows a PA divided into tag, LLC index, and block offset. In ElaCache, as in any conventional system, on an L2 miss, the hardware uses a function of some of the Tag and Index bits to decide which LLC slice to access.

Once the request reaches the destination slice, ElaCache uses an address translation step to redirect the access to an entry in the domain's partition. For this, ElaCache logically divides the Index field as shown in Chart (b), into a Logical Chunk Number (LCN) and a Chunk Offset. The latter has $\log_2(s)$ bits, where s is the number of sets in a logical chunk (or an RSU). The LCN bits access a table that translates them into a *Physical Chunk Number* (PCN) in the domain's partition (Chart (c)). The resulting PA is used to access the LLC. In the following, we explain how this translation occurs and the additional support needed.

1. Translation Process. Figure 2 shows the chunk translation process in a slice. ElaCache uses two tables: the per-domain *Indirection Table* (IT) and the *Ways Selection Table* (WST) shared by all the

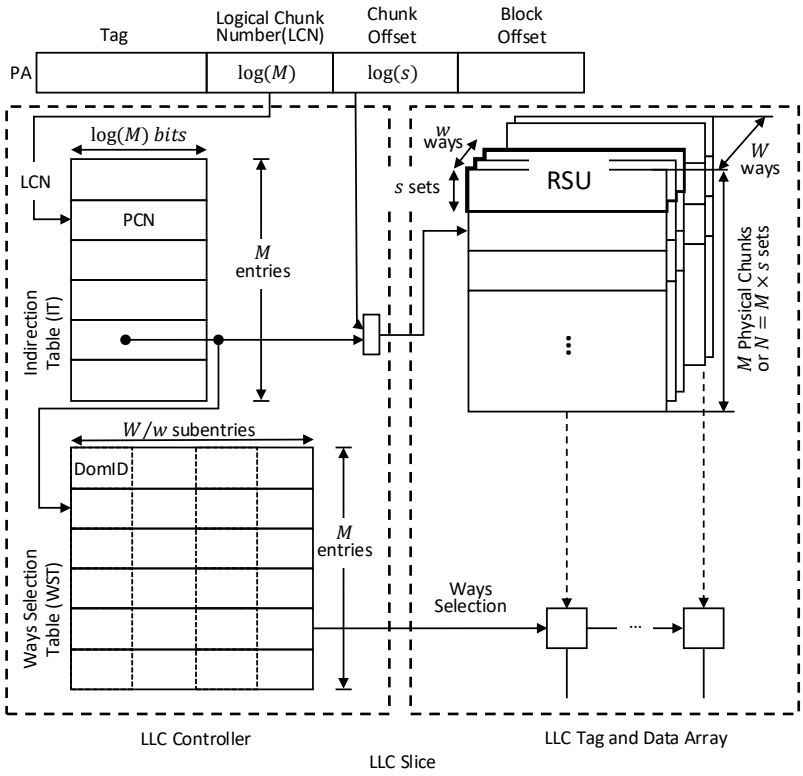


Figure 2: Chunk translation process in an LLC slice of ElaCache.

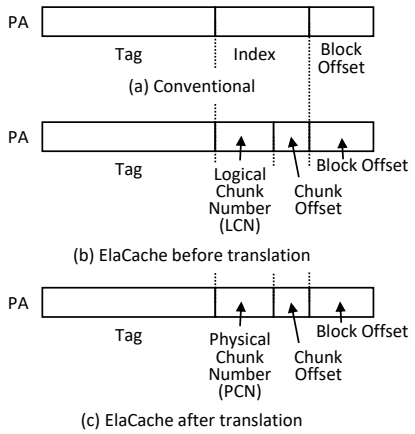


Figure 3: Physical address translation process.

domains in the slice. These tables are in the cache controller of the LLC slice, and are accessed every time that a core accesses the slice. The figure also shows an LLC slice with N sets of W ways. The slice is divided into RSUs of s sets of w ways. Hence, a way is logically partitioned into M=N/s Physical Chunks, and there are M * W/w RSUs in the slice.

The figure shows a PA that reaches the slice. The index bits are divided into $\log_2(M)$ bits for the LCN and $\log_2(s)$ bits for the offset in a chunk. The IT of a domain maps LCNs to PCNs owned by the domain. The IT is indexed by the LCN and has M entries of $\log_2(M)$ bits, where each entry is a PCN. The WST records which ways belong to the partitions of which security domain ID (*DomID*). The WST is indexed by the PCN and has M entries. Each entry has W/w subentries. Each subentry has the DomID of the domain that owns that RSU.

When a PA is presented to an LLC slice, the LCN bits are read and index the IT. The resulting PCN, concatenated with the chunk offset, is used to index the LLC. While the ways of the LLC are being read, the PCN indexes the WST, which provides which ways belong to the domain. As shown in the figure, only these ways are read to check for an LLC hit.

The IT of a domain with a small partition may have multiple entries with the same PCN. In this case, the corresponding physical chunk contains lines from multiple logical chunks. Hence, the tags of the LLC include bits for both the conventional tag and the LCN bits of a PA (which we call the *Index-Tag* bits). An LLC hit occurs only if the conventional tag and index-tag of the cache line match those of the requested line.

With the WST and ITs, accesses from different domains get mapped to different parts of the LLC slice.

2. Cache Replacement Logic. Multiple partitions can use the same cache sets (but different ways), and ElaCache enforces that line replacement decisions in a partition are unaffected by the activity in other partitions. Hence, upon a cache hit, ElaCache only updates replacement policy states of cache lines belonging to the same domain. In addition, on a miss, the eviction candidate can only be a line from the same domain. The domain IDs are obtained from the WST.

Our design may require adapting some existing cache replacement policies to ensure non-interference between ways belonging to different domains. However, it can easily support Pseudo-LRU (PLRU) [23], which is the policy we use. In our design, we place a PLRU sub-tree in each RSU (which is 3-way). To find a victim, each RSU owned by the domain that needs space yields one candidate, and then a final candidate is randomly chosen from one of them.

3. Operating System and Non-trusted Domain Virtual Machines (VMs). The architecture described can flexibly be used in different ways. Threads from the same domain running on different cores share the same ITs. Each VM can also have an additional domain for the guest operating system. Further, VMs that are not security-sensitive such as non-trusted domains can all share the same partition. The hypervisor has its own partition.

On a context switch between domains, the hypervisor saves the IT of the preempted domain to memory, and restores from memory the IT of the incoming domain. In our default design, the IT of a domain amounts to 128 bytes per slice, and saving/restoring this structure has minimal overhead. The data cached in the LLC partition of the preempted domain is not explicitly flushed. This allows a preempted domain to retain some of its RSUs when it is finally restored. The WST remains unmodified because each of its entries has the DomID of the owner domain, even when that domain is preempted. During execution, the hypervisor may reach to the ITs saved in memory to access the partitions of currently-preempted domains. This is necessary, for example, when the hypervisor wants to take some RSUs from a preempted domain and assign them to another domain; in this case, the hypervisor changes the IT and updates the corresponding data in the LLC, as we will see.

If the hypervisor is untrusted, modifications to preempted domains must be done through specialized hardware interfaces (e.g., SGX). Also, the hardware needs to add integrity and freshness guarantees to the IT and data stored in memory.

4. Sharing Code/Data across Domains. Because of security concerns [41], ElaCache disallows sharing among domains. If applications want to share code or data, they must be in the same domain. Even read-only pages cannot be shared across domains: they need to be replicated to eliminate any timing side channels. Not sharing pages is not a limitation of our design; it is considered a standard security measure in public cloud vendors such as AWS [3].

5. Cache Coherence Protocol Considerations. ElaCache does not affect the cache coherence protocol, and multiple cores running in the same security domain can share data. All L2 misses get tagged with the domain ID, which is used to pick the correct RSU in the LLC cache. However, the other L2 caches check only the physical address when receiving a snoop request. They do not check the domain ID. Partition changes do not affect the coherence either.

4.3 Profiling for Competitive Partitioning

During execution, a domain may need different LLC space at different times. Moreover, other domains may begin or terminate execution, requesting or freeing up space. For these reasons, ElaCache periodically enters a *Profiling* execution mode, where the hardware profiles the LLC misses of all the domains. After that, ElaCache potentially resizes the partitions of some or all the domains to divide the space more efficiently.

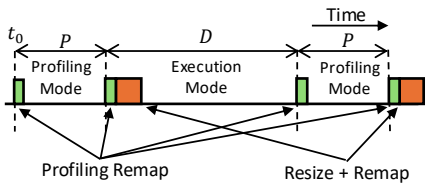


Figure 4: Execution timeline. *P* is duration of the profiling period and *D* is the time between profiling periods.

Figure 4 shows the timeline of a program's execution. The figure is not at scale. At regular *P+D* time intervals, execution enters the *Profiling Mode*, where performance counters in each of the LLC slices are enabled to count the number of LLC misses of each partition. After a *P* time period, profiling stops, and the hypervisor collects the miss counts and compares the values from all the partitions. Then, it makes a decision to potentially resize different partitions and hence remap the partitions' cached data. These operations are shown in Figure 4 with red rectangles labeled *Resize + Remap*, and are described later.

Unfortunately, the measured miss count for a partition does not tell by how much the miss count would change for a different-sized partition. To address this problem, ElaCache also estimates, during the profiling period, the miss count of larger and smaller partitions. It then generates a curve of the miss count vs partition size for all the domains. With this information, ElaCache reassigns RSUs from domains that are less sensitive to the partition size to those that are more sensitive. The result is a reduction in total misses.

To support these miss estimations for various partition sizes, ElaCache reserves a few logical chunks (LCs) and physical chunks (PCs) per domain. We call them *Profiling* LCs/PCs. Then, it maps these profiling LCs to the profiling PCs in a deliberate way, so that it artificially changes the probability of accessing each of these PCs for a range of addresses in the domain. Specifically, if we map many LCs to the same PC, we are emulating a small-sized partition for the domain (for this range of addresses); if we map few LCs (or just one LC) to the same PC, we are emulating a large-sized partition for the domain (for this range of addresses). If the profiling LCs are representative of the rest of the LCs, measuring the miss counts on the profiling PCs will allow us to estimate the miss counts of the domain in larger or smaller partitions than the current one.

Example. Assume that, in the LLC slice of Figure 2, M is 128 LCs, W is 12 ways, and an RSU has w=3 ways. Hence, there are $M \times W/w = 128 \times 12/3 = 512$ RSUs in the LLC. Further, assume that the slice has four domains. If each domain has the same size, each domain has 1/4 of the slice or 128 RSUs.

Given a domain, irrespective of the current size of its partition, we can estimate the domain's miss count if it had a partition size of 1/4 of the LLC slice by taking one profiling LC and mapping it, exclusively, to one profiling RSU. The misses observed multiplied by M (where M=128 LCs) would be the total estimated misses. Similarly, we can estimate the miss count if the domain had a partition size of 1/2 of the LLC slice by taking one LC and mapping it, exclusively, to two RSUs. The misses observed multiplied by M would be the estimated misses. Further, we can estimate the domain's miss count with a partition size of 1/8 of the LLC slice by taking two LCs and mapping them, exclusively, to one RSU. The misses observed on this

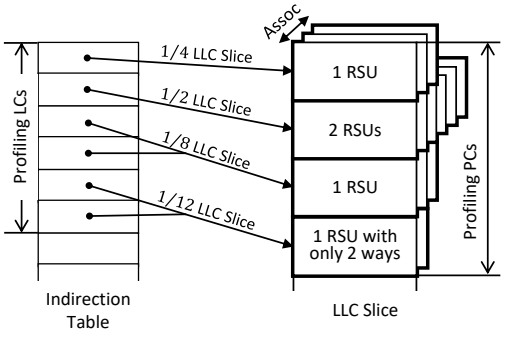


Figure 5: Example of profiling LC to profiling PC mapping that profiles four different partition sizes concurrently.

RSU multiplied by $M/2$=64 would be the estimated misses. Finally, we can estimate the miss count of a partition size of 1/12 of the LLC slice by taking two LCs and mapping them, exclusively, to one RSU with only 2 ways; the remaining way is left unused. The misses observed multiplied by $M/2$=64 would be the estimated misses.

The resulting mapping of profiling LCs to profiling PCs and RSUs is shown in Figure 5, where the four mappings described above are shown, in order, from top to bottom. With this mapping, we concurrently estimate the miss count for 4 different partition sizes by forfeiting 6 LCs and 4 PCs (with 5 RSUs). We also estimate the miss count for the current partition size by monitoring the non-profiling LCs.

Implementation. Before starting the profiling period, the hypervisor takes 6 LCs from each partition and maps them to 5 RSUs of the same partition as shown in the figure. Then, at the end of the profiling period, after collecting all the information, it returns the 6 LC mappings to their original values. The overhead of these operations is shown in Figure 4 as green rectangles labeled *Profiling Remap*. The operation is described in Section 4.5 in detail.

4.4 Resizing Algorithm

After the profiling step has concluded, the hypervisor runs the *Resizing* algorithm for each of the LLC slices on one or more cores. Based on the misses recorded for each of the current and estimated partition sizes during the profiling period, the algorithm may decide to change the sizes of some partitions. The goal is to minimize the total misses across all the partitions, to increase execution speed.

Algorithm 1 shows the algorithm for an LLC slice. First, it determines which domains are insensitive to the partition size. A domain is insensitive if either: (1) the number of misses is smaller than *FewMisses* for all the profiled partition sizes or (2) the number of misses changes negligibly with the partition size (i.e., the slope of the miss count vs partition size is flat). The latter case typically happens in streaming access patterns that do not reuse cache lines. In either case, the algorithm assigns the smallest possible partition size (*MinPartition*) to the domain.

The algorithm now decides how to repartition the remaining LLC space among the sensitive domains. The idea is to take space from domains whose miss count increases little with smaller partitions and re-assign it to domains whose miss count decreases substantially with larger partitions. To see how this is implemented, consider Figure 6(a). The profiling step has produced the miss counts

Require: Profile information for each domain that owns a partition in this LLC slice
for each domain **do**
  if miss count < *FewMisses* **or** it has a flat slope **then**
    Mark domain as insensitive
    Assign *MinPartition* to the domain
  else
    Mark domain as sensitive
  end if
end for
for each sensitive domain **do**
  Curve-fit the miss count into $k/x + b$
  Compute the domain's weight $\sqrt{k}$
end for
Compute the remaining cache space
Assign space to each sensitive domain based on relative weight
for each sensitive domain **do**
  Compute how many RSUs it gains or loses
end for

Algorithm 1: Resizing algorithm for an LLC slice.

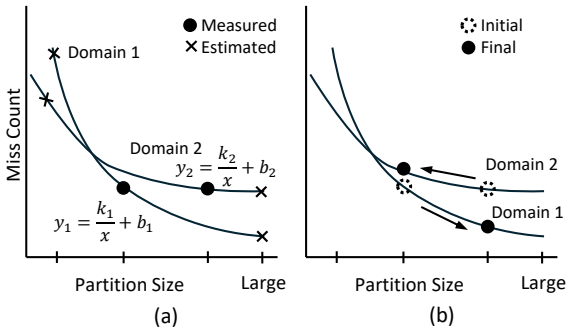


Figure 6: Reassigning LLC space among domains.

of each domain for different partition sizes. The figure shows the miss counts for two domains, with • for the misses at their current partition sizes (labeled *Measured*) and × for the misses at estimated partition sizes (labeled *Estimated*) at the extremes. Using simple computations, the hypervisor generates a curve $y = k/x + b$ that fits the points of a domain and, therefore, computes k_1, b_1, k_2, and b_2. Then, the slope of a curve at each partition size is given by its derivative function $y' = -k/x^2$.

We want to take cache space from domains that execute at a low-slope point and give it to domains that execute at a high-slope point. In the figure, this means taking space from domain 2 and giving it to 1, as shown in Figure 6(b). To accomplish this, the hypervisor finds the ratio of partition sizes that makes the slope to be the same. Specifically, the slope is the same when $k_1/x_1^2 = k_2/x_2^2$. Rearranging, $x_2/x_1 = \sqrt{k_2}/\sqrt{k_1}$. Hence, the optimal assignment is when each domain gets space proportional to its $\sqrt{k}$. We call $\sqrt{k}$ the domain's weight. The hypervisor assigns the cache space to each domain based on the relative weights of the domains. Then, given the difference between the current and desired space assignment, Algorithm 1 computes how many RSUs each sensitive domain gains or loses. To prevent thrashing, a partition is not allowed to lose more than *MaxLoss* RSUs at a time.

If a partition is profiled as insensitive for 3 consecutive profiling periods, to reduce overhead, it is put into a special *passive* profiling mode. In this mode, the partition size is shrunk even more to 4 RSUs, and profiling only involves measuring the misses with the current partition size. If the number of misses changes dramatically in one profiling period, the partition is moved out of passive profiling mode and the more advanced profiling resumes.

4.5 Remapping Algorithm

After the hypervisor has run the Resizing algorithm to decide how many RSUs to take from or give to each partition in the slice, it now runs the *Remapping* algorithm to perform three steps: 1) decide which specific RSUs are taken from each *donor* partition, 2) decide which specific RSUs are given to each *receiver* partition and how they are mapped, and 3) remap the chosen RSUs by updating both the LLC and the translation structures (i.e., the ITs and WST).

4.5.1 General Principles of RSU Reassignment. The process of reassigning an RSU from donor partition P_d of domain DM_d to receiver partition P_r of domain DM_r involves unmapping any number of logical chunks in DM_d that map to the RSU, and remapping one or more logical chunks in DM_r to the RSU. The choice of which RSU to reassign is given by two principles. First, the resulting ITs of DM_d and DM_r should distribute the LLC accesses as evenly as possible across the RSUs in their partitions, while minimizing the number of unmappings and remappings. Second, when multiple choices are equally good, we choose to reassign an RSU that does not change the LLC sets in P_r or P_d, just the associative ways.

To understand the first principle, consider first the algorithm in the donor domain DM_d. As an example, Figure 7a shows the IT and LLC partition of DM_d before (left) and after (right) donating one RSU. Initially, its 4 logical chunks point to 3 physical chunks of one RSU each. We define the *Density* (D) of a physical chunk as the number of logical chunks mapped to it. In the example, D is 2, 1, and 1 for physical chunks 0, 1 and 2, respectively. If a physical chunk had a double number of ways, then its D would be halved. If we assume that all logical chunks receive approximately the same number of accesses, physical chunks with lower D receive fewer requests. To keep physical chunk use as even as possible, our algorithm asks DM_d to donate an RSU from the physical chunk with the lowest density. In the example, DM_d donates the RSU from physical chunk 2, remapping logical chunk 3 to physical chunk 1. The right chart shows the result.

The algorithm used in the receiver domain DM_r is the opposite. Our algorithm asks DM_r to map the RSU so that it relieves the physical chunk with the highest density. As an example, Figure 7b shows the IT and LLC partition of DM_r before (left) and after (right) receiving one RSU. Initially, the 3 logical chunks point to 2 physical chunks of one RSU each. D is 2 and 1 for physical chunks 0 and 1, respectively. Hence, physical chunk 0 receives more requests. To keep physical chunk use as even as possible, ElaCache maps the incoming RSU to relieve physical chunk 0. Specifically, it remaps logical chunk 0 to the new physical chunk 2. The result is shown in the right chart.

In general, unmapping an RSU from DM_d may require changing the mapping of multiple logical chunks that were mapped to the RSU. Similarly, mapping an RSU to DM_r may require remapping multiple logical chunks that should now be mapped to the RSU.

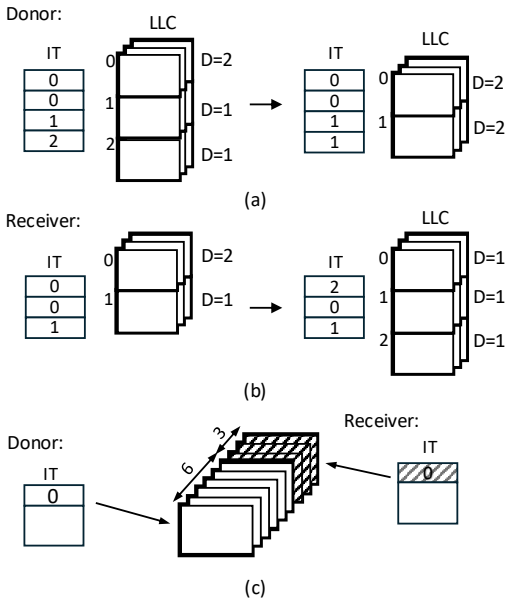


Figure 7: Choosing the appropriate RSU remapping.

Reassigning an RSU from DM_d to DM_r typically requires updating the ITs of the two partitions and physically moving data in the LLC (Section 4.5.2). However, some reassignments are cheap, which justifies the second principle above. Specifically, consider the case when the RSU reassignment does not change the LLC sets of P_r (or P_d). This is shown in Figure 7c. Initially, P_d has 1 physical chunk with 6 ways and P_r has 3 other ways of the same physical chunk. If ElaCache reassigns one RSU (i.e., 3 ways) of this physical chunk from DM_d to DM_r, the ITs remain unmodified—only the WST is modified. Importantly, after the donor has moved off the lines from the donated RSU, the receiver does not need to move its lines into the freed-up ways—the new ways will be filled on demand. This property saves data movement.

4.5.2 Algorithm for RSU Reassignment. Based on these principles, RSU reassignment works as follows. The hypervisor processes all the donor domains one at a time. For each domain, it sorts its physical chunks in increasing density. If multiple physical chunks have the same density, it sorts them in decreasing associativity. Then, starting at the top of the list, it unmaps as many RSUs as the Resizing algorithm needs. These RSUs are put in a free global pool.

Then, the hypervisor processes all the receiver domains one at a time. For each receiver domain DM_r, it searches the free pool for RSUs that: 1) simply add more associativity to physical chunks already in DM_r, and 2) if taken, would not reduce the D of those physical chunks in DM_r too much relatively to the other physical chunks. Then, DM_r takes from the list as many RSUs with these properties as it needs. If it needs more, it takes RSUs randomly from the global pool. All these RSUs are remapped according to the two principles discussed above.

4.5.3 Updating the LLC and the Metadata. Consider an RSU_0 that is to be reassigned from DM_d to DM_r. The hypervisor has identified

in DM_d: 1) the set of logical chunks that currently map data to RSU_0 and 2) for each of them, the destination RSU_d that the logical chunk must remap the data to in the future. Similarly, the hypervisor has identified in DM_r: 1) the set of logical chunks that must remap data to RSU_0 in the future and 2) for each of them, the source RSU_s that the logical chunk currently maps the data to. It is now necessary to move LLC data from RSU_0 to one (or more) RSU_d in DM_d, and then to move LLC data from one (or more) RSU_s in DM_r to RSU_0. Also, the WST and the ITs of DM_d and DM_r must be updated. These operations are done in hardware by the cache controller of the LLC slice.

We propose an API that allows the hypervisor to pass the necessary information to the cache controller so it performs these operations. The cache controller supports a single "data transfer" operation. An RSU reassignment is composed of multiple invocations of this basic operation with different arguments. Next, we explain this basic operation and how multiple invocations of it in sequence combine to complete an RSU reassignment.

Basic Operation. The basic operation is called *Transfer*. Within a given partition, *Transfer* moves the data of an LC from an RSU_s to another RSU_d. To do so, *Transfer* traverses RSU_s to find lines that belong to the LC. (Note that RSU_s may contain lines from other LCs that should not be moved now.) Then, for each line found, the algorithm checks if the data plus its Index-Tag bits (Section 4.2) can be moved to RSU_d. They can be moved if RSU_d has an empty spot at the corresponding index. If it does not, the line is discarded. Note that RSU_d may have some valid data from another LC, and it should not be overwritten. Finally, the line in RSU_s is marked as invalid.

Performing Multiple Basic Operations to Reassign an RSU. To reassign an RSU (e.g., RSU_0), the hypervisor must invoke *Transfer* multiple times. First, it invokes it for the donor DM_d, in order to clear RSU_0. Specifically, it invokes *Transfer* for each of the LCs that currently map data to RSU_0. Each invocation unmaps one LC, moving its data to another RSU. Next, the hypervisor invokes *Transfer* for the receiver DM_r, to move new data into RSU_0. Again, it invokes *Transfer* for each of the LCs that should now map data to RSU_0. Each invocation remaps one LC, moving its data to RSU_0.

4.5.4 Overall Operation. The remapping algorithm for an LLC slice is executed by a core running the hypervisor. As part of the algorithm, the core invokes all the *Transfer* operations needed. In response to a *Transfer* call, the LLC controller hardware moves the data from the source to the destination RSUs in the LLC, and then updates the WST and IT. During the operation, the controller blocks access to the two RSUs and to the relevant IT and WST entries. Any requests by any core that attempt to access these structures are postponed. Accesses to other RSUs and IT/WST entries proceed as usual.

A single core can execute the Remapping algorithm for all the LLC slices sequentially. Alternatively, multiple cores can operate on different slices in parallel.

4.6 ElaCache Components and Overheads

The hardware components of ElaCache in each LLC slice are the per-domain IT, the global WST, the Index-Tag bits, and an enhanced LLC controller. The software components are the profiling, resizing, and remapping algorithms. The IT access adds some non-overlapped time to the LLC access. Using CACTI [4], we estimate such time to be 2 cycles at 4 GHz which, in our simulations, adds 0.4% execution overhead. The access to the WST is hidden by the LLC access.

4.7 Partitioning the Extended Directory (ED)

The partitioning described for the LLC slices (with their associated Traditional Directory slices) can also be applied to the ED slices. Increasing the ED partition size of a domain above the size of the L2 can improve performance if a parallel application experiences footprint imbalance across slices—i.e., the application's threads running on different cores tend to access more data from some ED slices than from others. Decreasing the ED partition size of a domain below the size of the L2 is tolerable if the application in the domain has a small working set. In general, the values of the thresholds and parameters in the ElaCache algorithms may be different for LLC and ED.

5 Security Analysis

ElaCache offers strict isolation between security domains in a distributed LLC/directory. Each domain can only access its partition in these structures. Conflicts, evictions, and replacement algorithm state changes occur only within a partition.

The resizing events can reveal information about the victim's behavior, by exposing the change in the partition size. To ensure that no other information is leaked, on a *Transfer* operation, the controller blocks access to both source and destination RSUs for a constant time. Such time is equal to the worst case overhead of a *Transfer*, which involves the move of every single line in the RSUs. If the number of lines that need to be moved is lower than this maximum number, the controller pads its latency. Hence, the total latency of a resize depends only on the number of RSUs involved in the resize, and not on the data in these RSUs.

The leakage of a victim performing a set of resizes can be analyzed as follows. An information-theoretical upper bound on the leakage bit rate is $\log_2(D_o) \cdot D_f$, where D_o is the number of possible outcomes in one resizing decision, and D_f is the frequency of resizing. This is a very conservative bound, as it assumes that every victim resizing uniquely leaks secret information that the attacker can instantly observe. In practice, (1) differences in secret-dependent victim behaviors may not change resizing decisions; (2) a victim's partition resizing may not impact an attacker partition; and (3) even if it does, the attacker can only indirectly detect the victim's resizing outcomes by measuring their own partitions' sizes through cache occupancy, which takes time and may thus miss some resizing events.

ElaCache supports setting the resizing granularity and frequency on a per-domain basis. A domain can tune these values to reduce potential leakage based on the above analysis. For example, the default ElaCache design considers resizing every 4 ms, and the possible resizing decision outcomes are: no change, adding from 1 to 192 RSUs, or losing from 1 to 64 RSUs. Thus, one resize decision leaks at most $\log_2(1 + 192 + 64) = 8.01$ bits of information, which results in an overall leakage rate bound of 250 B/sec. But a more strict ElaCache design can choose to resize partitions in units of 16

Table 2: Leakage rates of existing LLC covert channels.

Attack	Rate	Uses Shared Memory?
Streamline [37]	1801 KB/s	Yes
Flush+Flush [18]	588 KB/s	Yes
Flush+Reload [18]	298 KB/s	Yes
Prime+Probe [27]	75 KB/s	No
Occup. Attack [49]	10 KB/s	No
Defense		
ElaCache-Default	**< 250 B/s**	**No**
ElaCache-Strict	**< 25.5 B/s**	**No**

RSUs every 20 ms, reducing the leakage rate bound to 25.5 B/sec. We call this design *ElaCache-Strict*.

ElaCache's leakage rate *bounds* are orders of magnitude smaller than *achievable* leakage rates in non-partitioned LLCs. This is shown in Table 2 for some existing LLC covert channels. In addition, ElaCache's leakage bounds can be used to guarantee security despite leakage [2, 15, 55, 56]. Using the leakage bound, the operating system can determine when an ElaCache domain exhausts its pre-specified "leakage budget" and then either (1) disable further dynamic resizing of the domain's partition or (2) issue a signal to refresh secrets, such as ephemeral cryptographic keys.

This leakage budget enables a security-performance tradeoff. A larger budget permits more runtime resizings, and hence higher performance. The right budget depends on the nature of the secrets. Specifically, if the victim computes on short-lived secrets containing a lot of information, such as ephemeral cryptographic keys or images, they can tolerate a larger leakage budget in exchange for higher performance. Conversely, if the victim computes on long-term secrets containing little information like social security numbers, they prefer a lower leakage budget, which hurts performance but not security.

6 Evaluation Methodology

Architecture. To evaluate ElaCache, we implement our design in the Sniper [9] simulator. We simulate a 16-core chip multiprocessor roughly following the Intel Skylake [21] with a sliced LLC and directory. Each core is co-located with its private caches and one LLC and directory slice, and is connected to a 4x4 mesh NoC. Table 3 shows the architecture and the ElaCache parameters. In the simulator, we set constant overhead values to operations that the hypervisor would do. Specifically, we assume that a core takes 20,000 cycles per slice to execute the Resizing and Remapping algorithms until it produces the final list of RSUs to be remapped. Then, a core that uses this list and invokes the *Transfer* commands to the cache controller takes an additional 50 cycles plus 10 cycles per *Transfer* invocation. The controller performs the *Transfer* operation in hardware in the background. We estimated these overheads based on an analysis of the instructions to be executed to complete these algorithms, and of the hardware coherence message latencies and controller overhead. Note that these overheads have a minimal impact on performance, since resizing occurs every 4ms, which is every 16M cycles.

We compare 6 different ways to partition an LLC and ED slice:
• *Baseline.* It uses static partitioning along sets, with as many partitions as domains. All partitions have the same number of contiguous sets.

Table 3: Architecture and ElaCache parameters.

Parameter	Value
Architecture	16 cores, 4.0 GHz, MESI protocol
Out-of-order core	6 issue, no SMT, 224 ROB, 128 LSQ
Cache Line	64 B
Private L1-D, Private L1-I	32 KB, 8-way, 4 cycle latency
Private L2	1 MB, 16-way, 10 cycle latency
Shared L3 (per slice)	2048 sets, 1.5MB, 12-way, 20cyc local lat.
Extended Direc. (per slice)	2048 sets, 12-way
Network on Chip	4x4 mesh, 2 cycle lat. / hop, 128b width
DRAM	50 ns roundtrip latency after L3
ElaCache parameters	
RSU Size	16 sets with 3 associative ways
Indirection Table (IT) Size	128 entries per domain
Indirection Overhead	Visible: 2 cycles at every LLC access
Profiling Region	6 LCs and 5 RSUs
Interval Duration	2ms profiling and 2ms execution
Resizing Algorithm	*MinPartition* = 16 RSUs, *FewMisses* = 10, *MaxLoss* = 64 RSUs
Basic Profiling	$N_1 = 8$, $N_2 = 4$
Resizing + Remapping Algo	20,000 cycles
Invoke *Transfer* for RSU List	50cycles + 10cycles per *Transfer* invoked
ElaCache Strict	2ms prof; 18ms exec; 16RSUs minimum reassign

• *Unsafe.* It has no partitioning.
• *ElaCache* and *ElaCache Strict.* The latter uses the more conservative resizing of Section 5.
• *UMON.* It is the UMON [35] scheme, a dynamic no-annotation partitioning scheme that offers strict isolation, and also serves as the basis for several other designs [24, 50]. However, UMON only supports way-partitioning, which limits its scalability: since our LLC has 12 ways, UMON can at most support 12 domains.
• *ElaCache BP.* It is ElaCache using a simple form of profiling that we call Basic Profiling (BP). In a profiling period, BP only collects the misses for the current partition sizes. With BP, the Resizing algorithm assigns *MinPartition* to domains with few misses and competitively repartitions the rest of the cache space between the other domains. The repartitioning algorithm tries to take space from the domains that have fewer misses and give it to the domains with more misses. Specifically, it ranks the sensitive domains in decreasing order of miss counts in the current profiling period. Then, it moves N_1 RSUs from the last domain to the first one, and N_2 RSUs from the one before the last to the second one.

Applications. To evaluate ElaCache, we use applications from PARSEC 3.0 [6]. In most experiments, we run 16 applications at a time, each with 1 thread (hence, 16 domains); in some experiments, we run 8 two-threaded applications at a time (hence, 8 domains). In all cases, each thread is pinned on a different core. In our simulations, we fast-forward until the application reaches the region of interest. Then, each thread warms-up for 200 M instructions, and then we measure the next 1 B instructions per thread.

We group the applications into 2 categories based on their LLC demand. Given an application, we measure the performance of its multithreaded execution with a quarter of each LLC slice (i.e., 128 RSUs) and with the *MinPartition* size (i.e, 16 RSUs). If the performance decreases by about 5% or less, we call the application *No Cache Pressure (NCP)*; otherwise we call it *Cache Pressure (CP)*. The classification is shown in Table 4.

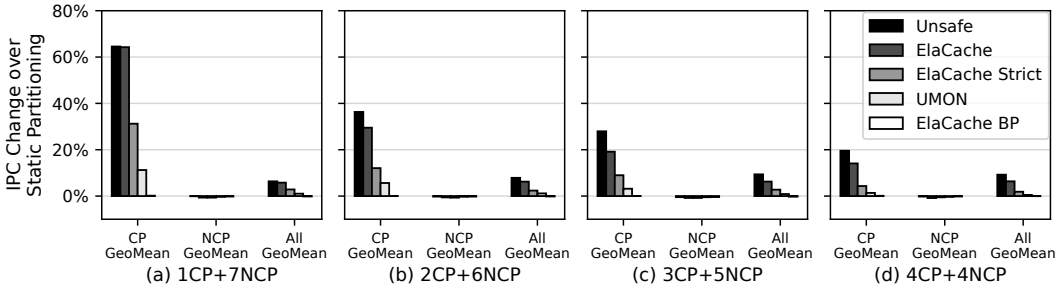


Figure 8: Impact of different partitioning schemes on the IPC of mixes of 8 applications with 8 domains per LLC slice.

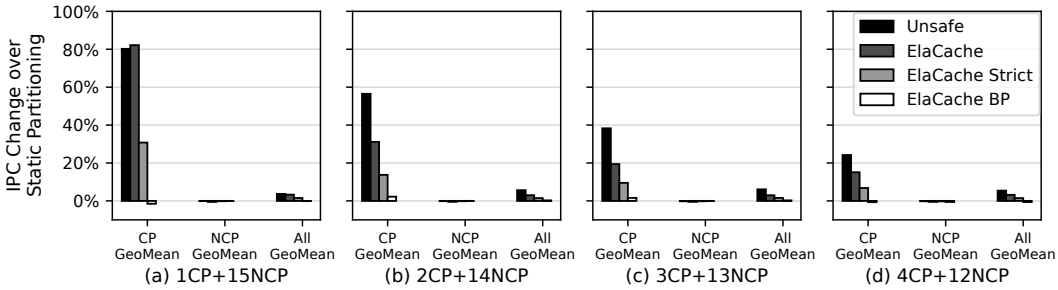


Figure 9: Impact of different partitioning schemes on the IPC of mixes of 16 applications with 16 domains per LLC slice.

Table 4: Categorization of the applications.

Category	Applications
No Cache Pressure (NCP)	radiosity, bodytrack, facesim, fluidanimate, barnes, swaptions, raytrace, blackscholes, water.nsq, water.sp
Cache Pressure (CP)	canneal, streamcluster, ocean.ncont, dedup, lu.cont, lu.ncont, ferret, ocean.cont

7 Evaluation

In this section, we evaluate *ElaCache*. In our experiments, we find that dynamically partitioning the ED only delivers marginal gains because of its large size compared to L2. Consequently, for the ED, we always use same-size static partitioning along sets (i.e., *Baseline*).

7.1 Performance Impact of ElaCache

Figures 8 and 9 show the impact of different partitioning schemes on the IPC of mixes of 8 and 16 applications, respectively. In each figure, we show 4 graphs with different types of mixes, which differ in the number of CP applications in the mix: 1, 2, 3, or 4. For example, Chart (a) is 1CP+7NCP, which means it has 1 CP and 7 NCP applications. As we move to Chart (b), we replace one NPC by one CP application, and similarly for the other charts. We repeat the experiments 10 times, in each case starting with a different randomly-generated application mix. The graphs show the geometric mean of: the IPC change of the CP application(s), the IPC change of the NCP applications, and the IPC change of all the applications. The IPC changes are shown over *Baseline* (i.e., static partitioning) for the different architectures. *UMON* is not in Figure 9 because it does not support 16 domains,

In the 8-application mix, under 1CP+7NCP (Figure 8a), *ElaCache* provides a large IPC gain (64.3%) over *Baseline*. This gain is comparable to *Unsafe*, which suffers from the noisy neighbor effect. As the number of CP applications increases, *ElaCache* still provides considerable IPC improvements for them: 29.5% for 2CP+6NCP, 19.1% for 3CP+5NCP, and 14.1% for 4CP+4NCP. *ElaCache* performs even better for the 16-application mix, where the IPC gains are 82.2% for 1CP+15NCP, 31.1% for 2CP+14NCP, 19.4% for 3CP+13NCP, and 15.1% for 4CP+12NCP. Just as important, *ElaCache* does not noticeably hurt the IPC of the NCP applications in any of the cases in the figures.

ElaCache Strict struggles to offer even half of the gains of *ElaCache* for 8 and 16 applications. This is due to its larger resize granularity (i.e., 16 RSUs) and that each NCP application must keep a minimum of 16 RSUs, even if it does not need them. Still, *ElaCache Strict* sees substantial IPC gains—e.g., 31.2% for 1CP+7NCP and 30.8% for 1CP+15NCP.

UMON fails to perform well mostly due to its way-partitioning nature: each domain occupies at least 1 way in the LLC. Given that our LLC has 12 ways, for the 8-domain experiments, 4 ways out of 12 can be redistributed to the domain(s) that need LLC space. For 1CP+7NCP (Figure 8a), the CP application in *UMON* can use at most 5/12 (41.7%) of the LLC, while the same application can use more than 90% of the LLC with *ElaCache*. *UMON*'s limitation of way partitioning exacerbates when more security domains are needed in the system. For example, it cannot be applied to the 16-domain experiment (Figure 9).

ElaCache BP shows no performance gain. This shows the need for the advanced profiling proposed by *ElaCache*.

Many CP Applications. To gain further insight, we also evaluate *ElaCache*'s performance with many CP applications in the 16-application setting. We evaluate 1 to 8 CP applications, with the rest being NCP applications. To ensure stable results, we only run *lu.ncont* as the CP application. Hence, the results differ from

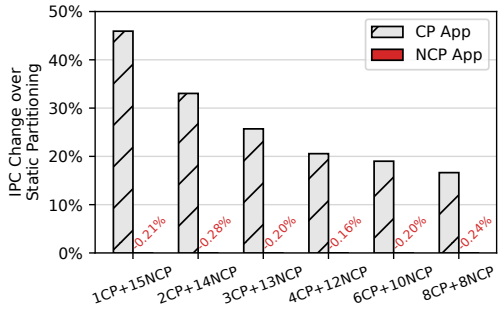


Figure 10: ElaCache gains for increased number of CP applications.

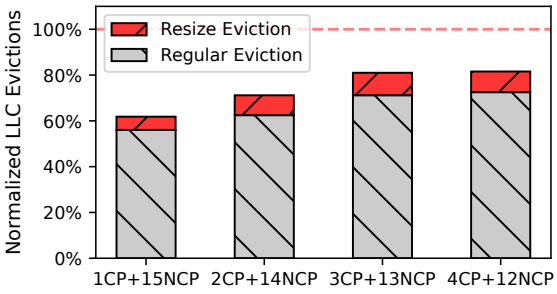


Figure 11: LLC evictions in *ElaCache* normalized to *Baseline*.

Figure 9. Figure 10 shows the IPC change over *Baseline* for CP and NCP applications. The NCP bars are so small that we show their height as numbers. As the number of CP applications increases, the gains delivered by *ElaCache* decrease, but they are still substantial: from 33.0% for 2CP+14NCP to 16.6% for 8CP+8NCP. In all cases, the NCP applications change their IPC negligibly.

7.2 Comparing the Number of LLC Evictions

ElaCache improves performance over *Baseline* by utilizing the LLC space better, reducing the number of line evictions. Figure 11 shows the number of LLC evictions with *ElaCache* normalized to *Baseline*. The figure shows bars for 1CP+15NCP to 4CP+12NCP. Evictions are divided into those that occur while *ElaCache* is resizing partitions (*Resize*) and the rest (*Regular*). Thanks to *ElaCache*'s effective use of the LLC, evictions in *ElaCache* are 61.8%, 71.2%, 81.0%, and 81.6% of those in *Baseline*. Also, *Resize* evictions are relatively few, accounting for 10% or fewer. This explains *ElaCache*'s better performance across different settings.

7.3 Characterizing Profiling

We tested Profiling Mode durations of 0.5ms–5.0ms, and fraction of execution time devoted to Profiling Mode of 50%–20%. We see no more than 1% of performance variation across configurations. The reason is that profiling only adds a small overhead to overall performance. In addition, some NCP applications are put into passive profiling mode and do not affect resizing decisions. To validate the first insight, we rerun *ElaCache* except that, after each profiling period, we discard the profile information and continue with the *Baseline* partition sizes. Figure 12 shows the resulting IPC change over *Baseline*. We see that the degradation is always less than 0.2%.

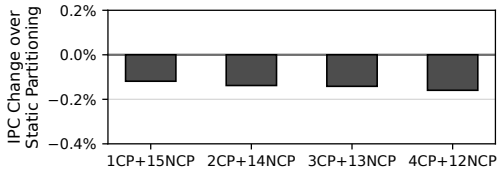


Figure 12: IPC degradation due to *ElaCache* profiling.

Table 5: Average values of *ElaCache* measures for 16 domains.

Measure		Value
# of Resizings per Thread		35.1
# of RSUs Moved per Resize (per slice and app)		8.5
# of Transfer Ops per Resize (per slice and app)		11.8
Diff between max & min physical chunk density		0.13
CP Partition Size; NCP Partition Size (in # of RSUs)	1CP+15NCP	408.2; 6.9
	2CP+14NCP	208.1; 6.8
	3CP+13NCP	145.9; 5.7
	4CP+12NCP	112.1; 5.3

Table 6: *ElaCache* storage overhead in an LLC slice.

Num of Security Domains in Slice	Storage Overhead	
	(KB)	% of LLC Size
4	21.6	1.32
16	23.0	1.41
64	28.4	1.74
256	49.5	3.04

7.4 ElaCache Performance Statistics

Table 5 shows the average values of some *ElaCache* measures for 16 domains. *ElaCache* performs 35.1 resizings per thread—recall that all the threads of all the applications perform the resizing at the same time. Per slice and application, 8.5 RSUs are moved in a resize with 11.8 transfer operations. Recall that an RSU move may require 0, 1, or more transfers. The difference between the maximum and minimum density of physical chunks (PCs) is only 0.13, which means that *ElaCache*'s remapping algorithm balances the load between PCs. The table also shows the average partition size in RSUs of CP and NCP applications for different counts of CP and NCP applications. We see that a CP application uses many RSUs (408.2 to 112.1), while an NCP one uses few (6.9 to 5.3). The average CP partition size decreases as more CP applications co-execute.

7.5 Storage Area and Power Analysis

We compute the storage overhead of ElaCache in one LLC slice with N domains. The 128-entry IT is 128x7 bits = 112B per domain. The WST is 128x4x$\log_2(N)$ bits = 64$\log_2(N)$ bytes, and is shared by all the domains. Each LLC tag increases by 7 bits, which results in 21KB over the whole slice. Table 6 shows, for different numbers of domains, the ElaCache storage overhead in absolute value and as a percentage of the original LLC slice size. We see that the overhead is small (1.32–3.04%). ElaCache also has a statically-partitioned ED, which increases the ED tags, for a total of 3x$\log_2(N)$ KB per slice. This overhead is also small.

We use CACTI 7 [4] to estimate the area and power of the storage for the ElaCache design with 256 security domains in Table 6. We

Table 7: Dynamic energy of a read and a write access.

Operation	IT	WST	LLC + TD
Read	0.1 pJ	0.8 pJ	14.2 nJ
Write	0.2 pJ	1.2 pJ	15.5 nJ

scale the numbers given by CACTI down to 7nm technology using [46]. The ElaCache storage hardware (i.e., ITs, WST, and extended tag arrays) replicated 16 times for the 16 LLC slices takes 1.60 mm^2 and consumes 151.15 mW of static power.

To estimate the dynamic energy consumed by accessing these ElaCache structures, we note that every time that a core accesses the LLC, the IT and WST are also accessed. Hence, we compare the dynamic energy of an access to the IT and WST, to an access to a slice of the LLC plus the traditional (TD) directory. We neglect here the very small impact of the *Index-Tag* bits added to the LLC tag. The numbers are shown in Table 7. We see that the accesses to the IT and WST are much cheaper than the access to the LLC plus TD. Overall, the dynamic energy of IT and WST is negligible.

A similar analysis shows that the storage area and power of the extra *Index-Tag* bits added to the ED tag are also negligible.

Since the remapping and resizing operations (including those when setting-up/terminating profiling) are much less frequent than LLC accesses, their total dynamic energy consumption spent accessing the IT and WST tables is negligible. Overall, compared to a server-class CPU that takes ≈500mm^2, has more than 250 W TDP, and has large LLCs, the area, static power, and dynamic energy overheads of ElaCache's storage are negligible.

7.6 Setting an Information Leakage Limit

To see the tradeoff between information leakage budget and performance, we set the leakage limit for a victim application to be 256 bits. This allows the application to perform no more than 31 resizes (Section 5). We consider the victims to be the CP applications, and run different mixes of CP+15NCP, 2CP+14NCP, 3CP+13NCP, and 4CP+12NCP under ElaCache without and with this limit for the CP applications. Figure 13 compares the IPC change of the CP applications without and with this limit (*ElaCache 256b*). The figure shows that, when applying this leakage limit, the overall performance degradation of the CP applications is only 0.7%. This is a very small impact. The reason is that ElaCache is able to set quasi-optimal domain sizes relatively early on in the execution of the applications. Therefore, stopping resizing toward the later part of the execution has little impact.

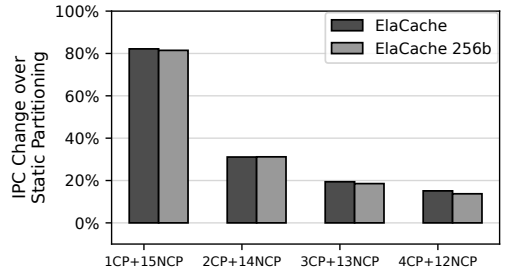


Figure 13: Performance impact of limiting leakage to 256b.

8 Other Related Work

In this section, we discuss related work beyond the one introduced in Section 2. We compared ElaCache to UMON because both are hardware-based partition schemes. UMON modifies the eviction and insertion policies of the cache. Other techniques, like CPA [30], provide sophisticated software algorithms to leverage existing way-partitioning support in current processors. Other way-partitioning designs based on UMON like SecDCP [50] and M-party [24] are not comparable to ElaCache. The reason is that, while ElaCache and UMON use the popular peer-to-peer model (i.e., all domains have the same security level), SecDCP and M-party only support special cases. Specifically, SecDCP assumes that domains have different security levels; lower-security domains can grow into the space used by higher-security ones but not vice-versa. Further, M-party assumes a 2-party scenario and secures it by adding a dummy partition for a 3-party that adds noise to hide the victim accesses.

Existing medium- and fine-grain dynamic resizing schemes such as Chunked-Cache [14] and CC [48], in addition to requiring software annotations, invalidate every cache line in a partition when resizing. This can significantly hurt performance if resizings are frequent. In contrast, ElaCache's *Transfer* operation migrates data between chunks without invalidations.

In modern processors, a core accesses the local LLC slice with lower latency than remote LLC slices—an effect called Non-Uniform Cache Access (NUCA). To reduce the average LLC access latency, there are NUCA-aware cache partitioning schemes, which place cache lines in slices that are close to the cores that use such lines. Specifically, Jigsaw [5] allows the software to define where cache lines are stored. Jumanji [42] improves Jigsaw in both security and tail latency. These NUCA-aware schemes operate on top of an LLC partitioning scheme, which can potentially be ElaCache.

Secure partitioning of hardware structures to defeat side-channel leakage is a popular research topic. Recent works [28, 47, 54] apply partitioning across microarchitecture components to guarantee non-interference. Untangle [56] provides rigorous leakage quantification for dynamically-partitioned structures such as caches.

9 Conclusion

This paper proposed *ElaCache*, a new design for dynamic partitioning of the LLC and directory. ElaCache supports fine-grained partitions of both sets and associative ways without any software annotation. It uses a novel micro-architecture that dynamically manages and resizes its partitions in hardware.

We evaluated ElaCache with simulations of a 16-core processor. Our results showed that ElaCache delivers high performance while providing strong security guarantees. When only one out of 8 running applications needs a large cache footprint, ElaCache improves the IPC of that application by 64.3% on average over static cache partitioning. Further, with 1 large-footprint application out of 16, the average IPC improvement is 82.2%. In both cases, the rest of the applications in the mix are not noticeably penalized.

Acknowledgments

This work was supported in part by NSF under grants CCF 2107470 and CCF 2316233; and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Kerem Arıkan, Abraham Farrell, Williams Zhang Cen, Jack McMahon, Barry Williams, Yu David Liu, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2024. TEE-SHirT: Scalable Leakage-Free Cache Hierarchies for TEEs. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024*.

[2] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. 2010. Predictive Black-Box Mitigation of Timing Channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*.

[3] Amazon AWS. 2022. The Security Design of the AWS Nitro System - AWS Whitepaper. https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html.

[4] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (June 2017), 25 pages. doi:10.1145/3085572

[5] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 213–224. doi:10.1109/PACT.2013.6618818

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81. doi:10.1145/1454115.1454128

[7] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. 2020. CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[8] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 42–56. doi:10.1145/3352460.3358310

[9] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)*, Article 5 (2014), 23 pages. doi:10.1145/2629677

[10] Anirban Chakraborty, Sarani Bhattacharya, Sayandeep Saha, and Debdeep Mukhopadhyay. 2023. Are Randomized Caches Truly Random? Formal Analysis of Randomized-Partitioned Caches. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[11] Anirban Chakraborty, Nimish Mishra, Sayandeep Saha, Sarani Bhattacharya, and Debdeep Mukhopadhyay. 2023. On the Amplification of Cache Occupancy Attacks in Randomized Cache Architectures. *arXiv e-prints* 2310.05172 (2023).

[12] Paizhuo Chen, Lei Li, and Zhice Yang. 2021. Cross-VM and Cross-Processor Covert Channels Exploiting Processor Idle Power Management. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 733–750. https://www.usenix.org/conference/usenixsecurity21/presentation/chen-paizhuo

[13] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. 2022. Don't Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2857–2874. https://www.usenix.org/conference/usenixsecurity22/presentation/dai

[14] Ghada Dessouky, Emmanuel Stapf, Pouya Mahmoody, Alexander Gruler, and Ahmad-Reza Sadeghi. 2022. Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-225/

[15] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. 2014. Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*.

[16] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. 2018. Drive-By Key-Extraction Cache Attacks from Portable Code. In *16th International Conference on Applied Cryptography and Network Security (ACNS)*.

[17] Lukas Giner, Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss. 2023. Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks. In *2023 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Los Alamitos, CA, USA, 2273–2287. doi:10.1109/SP46215.2023.10179440

[18] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.

[19] Yanan Guo, Dingyuan Cao, Xin Xin, Youtao Zhang, and Jun Yang. 2023. Uncore Encore: Covert Channels Exploiting Uncore Frequency Scaling. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[20] Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *Cryptographic Hardware and Embedded Systems (CHES)*.

[21] Intel Corporation 2024. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

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

[23] Kamil Kędzierski, Miquel Moreto, Francisco J. Cazorla, and Mateo Valero. 2010. Adapting cache partitioning algorithms to pseudo-LRU replacement policies. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. 1–12. doi:10.1109/IPDPS.2010.5470352

[24] Yuan-Tai Lin, Chin-Yu Sun, and TingTing Hwang. 2023. M-Party: A Secure Dynamic Cache Partitioning by More Than Two Parties. In *2023 IEEE 36th International System-on-Chip Conference (SOCC)*. 1–6. doi:10.1109/SOCC58585.2023.10256845

[25] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium*.

[26] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 406–418. doi:10.1109/HPCA.2016.7446082

[27] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622. doi:10.1109/SP.2015.43

[28] Jarrett Minton and Rajeev Balasubramonian. 2025. FLEXPROF: Flexible, Side-Channel-Free Memory Access. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 407–420. doi:10.1145/3676641.3715997

[29] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. 2021. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 645–662. https://www.usenix.org/conference/usenixsecurity21/presentation/paccagnella

[30] Lucia Pons, Julio Sahuquillo, Vicent Selfa, Salvador Petit, and Julio Pons. 2020. Phase-Aware Cache Partitioning to Target Both Turnaround Time and System Performance. *IEEE Transactions on Parallel and Distributed Systems* 31, 11 (2020), 2556–2568. doi:10.1109/TPDS.2020.2996031

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

[32] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*.

[33] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 775–787. doi:10.1109/MICRO.2018.00068

[34] Moinuddin K. Qureshi. 2019. New Attacks and Defense for Encrypted-Address Cache. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 360–371.

[35] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA*. IEEE Computer Society, 423–432. doi:10.1109/MICRO.2006.49

[36] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. Association for Computing Machinery, New York, NY, USA, 199–212. doi:10.1145/1653662.1653687

[37] Gururaj Saileshwar, Christopher W. Fletcher, and Moinuddin Qureshi. 2021. Streamline: A fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1077–1090. doi:10.1145/3445814.3446742

[38] Gururaj Saileshwar, Sanjay Kariyappa, and Moinuddin Qureshi. 2021. 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)*. 37–49. doi:10.1109/SEED51797.2021.00015

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

//www.usenix.org/conference/usenixsecurity21/presentation/saileshwar

[40] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *38th Annual International Symposium on Computer Architecture (ISCA)*. 57–68.

[41] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. 2021. Remote Memory-Deduplication Attacks. *ArXiv* abs/2111.08553 (2021). https://api.semanticscholar.org/CorpusID:244130204

[42] Brian C. Schwedock and Nathan Beckmann. 2020. Jumanji: The Case for Dynamic NUCA in the Datacenter. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 665–680. doi:10.1109/MICRO50266.2020.00061

[43] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2863–2880. https://www.usenix.org/conference/usenixsecurity21/presentation/shusterman

[44] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 639–656. https://www.usenix.org/conference/usenixsecurity19/presentation/shusterman

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

[46] Aaron Stillmaker and Bevan M. Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integr.* 58 (2017), 74–81. https://api.semanticscholar.org/CorpusID:205666078

[47] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. 2022. SecSMT: Securing SMT Processors against Contention-Based Covert Channels. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3165–3182. https://www.usenix.org/conference/usenixsecurity22/presentation/taram

[48] Daniel Townley, Kerem Arıkan, Yu David Liu, Dmitry Ponomarev, and Oğuz Ergin. 2022. Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2839–2856. https://www.usenix.org/conference/usenixsecurity22/presentation/townley

[49] Tarunesh Verma, Achilleas Anastasopoulos, and Todd Austin. 2022. These Aren't The Caches You're Looking For: Stochastic Channels on Randomized Caches. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. 37–48. doi:10.1109/SEED55351.2022.00011

[50] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2016. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *Proceedings of the 53rd Annual Design Automation Conference* (Austin, Texas) *(DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article 74, 6 pages. doi:10.1145/2897937.2898086

[51] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 675–692. https://www.usenix.org/conference/usenixsecurity19/presentation/werner

[52] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *2019 IEEE Symposium on Security and Privacy (SP)*. 888–904. doi:10.1109/SP.2019.00004

[53] Mengjia Yan, Jen-Yang Wen, Christopher W. Fletcher, and Josep Torrellas. 2019. SecDir: A Secure Directory to Defeat Directory Side-Channel Attacks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 332–345.

[54] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. 2023. Half&Half: Demystifying Intel's Directional Branch Predictors for Fast, Secure Partitioned Execution. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1220–1237. doi:10.1109/SP46215.2023.10179415

[55] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2011. Predictive Mitigation of Timing Channels in Interactive Systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*.

[56] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2023. Untangle: A Principled Framework to Design Low-Leakage, High-Performance Dynamic Partitioning Schemes. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 771–788. doi:10.1145/3582016.3582033

[57] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2024. Everywhere All at Once: Co-Location Attacks on Public Cloud FaaS. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 133–149. doi:10.1145/3617232.3624867