SeqAss: Using Sequential Associative Caches to Mitigate Conflict-Based Cache Attacks with Reduced Cache Misses and Performance Overhead

Wei Song*†, Zhidong Wang*†, Jinchi Han*†, Da Xie*†, Hao Ma*†, Peng Liu‡
*State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, Beijing, China
†School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
‡The Pennsylvania State University, University Park, USA
{songwei, wangzhidong, hanjinchi, xieda, mahao}@iie.ac.cn, pxl20@psu.edu

***Abstract*—Cache randomization has been proposed as an effective defense against conflict-based cache attacks. Mirage and Chameleon are two of the state-of-the-art randomized last-level caches achieving a strong defense. However, they rely on techniques intrusive to the traditional cache structure, such as cache skews, over-provided metadata space, and separated data storage, and prohibit the use of the LRU replacement policy. Mirage incurs 22% extra area and 21% extra power. When running memory heavy applications, Chameleon consumes significant dynamic power due to its high relocation rate.**

This paper proposes to mitigate conflict-based cache at tacks using sequential associativity. The proposed SeqAss cache retains the set-associative structure and supports LRU. It achieves a defense as strong as Mirage. Instead of raising cache miss rate, SeqAss actually reduces it by 11.4%. Its area and power overhead is 28.8% and 22.1%, respectively, lower than Mirage. When running memory heavy applications, it incurs ∼50% lower dynamic power overhead compared to Mirage and Chameleon.

1. Introduction

Modern processors adopt a multi-level cache hierarchy to reduce the impact of long memory access latency. Each processing core contains one or two levels of small private caches while a large last-level cache (LLC) is shared between cores. To reduce the amount of on-chip communication required for maintaining cache coherence, an LLC usually keeps a copy for all the data stored in private caches [1], [2]; therefore, cache blocks belonging to different cores (processes) may be stored in the same LLC cache set. A cache conflict occurs when a newly fetched cache block is inserted to a fully occupied LLC cache set. One of the stored LLC cache blocks is chosen and evicted to make room. To maintain the inclusive relationship with private caches, the copies of the evicted block stored in all private caches, also called the inclusion victim [1], are forcefully purged in the eviction process. This type of backwards purging enables attackers to launch conflict-based cache attacks targeting the LLC, because they can precisely infer a victim's access to a specific data by making the data an inclusion victim and purging it using malicious LLC conflicts.

Cache randomization [3] has been proposed as an effective defense against such attacks. By randomizing the mapping between addresses and cache set indices [3], [4], [5] (random mapping), inserting newly fetched cache blocks at randomly selected cache sets [4], [6], [7] (random insertion), and evicting seemly random cache blocks when cache conflict occurs [8], [9] (indirect eviction), cache randomization endeavors to prevent attackers from collecting congruent addresses[1], deterministically occupying target cache sets, or precisely monitoring the victim's accesses on target cache sets. The arm race between attackers and cache defenses is constantly evolving. Only random mapping and random insertion were commonly applied on the early designs of randomized caches [4], [6], and attackers were used to be able to search congruent addresses using fast algorithms [10], [11], [12], [13] and launch persistent attacks [14]. Later, they failed to do so on the recently proposed Mirage [9] and Chameleon [8] caches. Both adopt indirect eviction to prevent attackers from collecting congruent addresses. Mirage further eliminates associative evictions by over-providing metadata space [9], making it almost impossible for attackers to deterministically occupy target cache sets.

As more advanced defense techniques are adopted, the cost of enforcing cache randomization gradually increases. Early randomized cache proposals rely on frequent cache remaps [3], [15] to reduce the time window available for attackers, but it leads to increased cache misses and reduced IPC (instruction per clock) [3], [16] as $10 \sim 50\%$ data cached in the LLC are unnecessarily evicted during each remap. Skewed cache [4] is adopted by most randomized caches for enforcing random insertion, but it reduces the efficiency of replacement policies and complicates the LLC control logic [15]. Chameleon achieves indirect eviction by swapping the conflicted cache blocks pushed into the victim cache (VC) [8] back to the main cache array, but this swap incurs significant power overhead. In addition, all existing caches implementing indirect eviction [8], [9]

1. Addresses mapped to the same cache set with a target address [10].

enforce the random replacement policy rather than the least recent used (LRU) replacement policy. Finally, the over-provided metadata space and separated data storage used in Mirage incur 22% extra area and 21% extra power [9], which make Mirage unlikely to be adopted by future processor designs [17].

This paper would like to investigate the possibility of designing a randomized cache achieving a strong defense without most of the aforementioned overhead. To be specific, the proposed randomized cache is expected to achieve:

- A defense comparable to that provided by Mirage.
- Reducing cache misses rather than increasing it.
- Significantly reduced area and power overhead compared to Mirage and Chameleon.
- Preserving the LRU replacement policy and traditional set-associative structure without using cache skews.

Our idea is to apply sequential associativity [18] on a traditional set-associative cache and implement all the required randomization techniques on it. The intuition behind this idea is as follows: (a) Applying random mapping and sequential associativity on a set-associative cache would form a basic randomized cache which is inherently area-economical and cache-miss-reducing. (b) Although this design is still vulnerable to Evict+Time, it can be hardened without losing its inherent benefits. (c) Although this design consumes high dynamic power, it can be reduced in a practical way.

We name the new randomized cache as the *SeqAss* cache. It achieves a strong defense comparable to Mirage. Instead of degrading run-time performance, it reduces cache misses by 11.4%. Its area and power overhead is 28.8% and 22.1%, respectively, lower than Mirage. When running memory heavy applications, it incurs ∼50% lower dynamic power overhead compared to Mirage and Chameleon.

The artifact of this paper is available at https://doi.org/10.5281/zenodo.17248489.

2. Background

This section introduces the necessary background required for understanding this paper. Starting with a summary of the conflict-based cache side-channel attacks, we analyze the key techniques proposed and utilized by existing randomized caches, describe the extra attacks considered in the design of SeqAss, and finally introduce the concept of sequential associativity.

2.1. Conflict-Based Attacks

Conflict-based cache side-channel attacks occur when an attacker can deterministically occupy one or several cache sets shared with her victim and utilize this advantage to monitor the victim's accesses to these target cache sets. Depending on the way of monitoring such accesses, most attacks can be classified into two categories: Prime+Probe and Evict+Time attacks [19]. Prime+Probe infers victim's access by probing attacker's own cache blocks while Evict+Time does so by measuring victim's execution time.

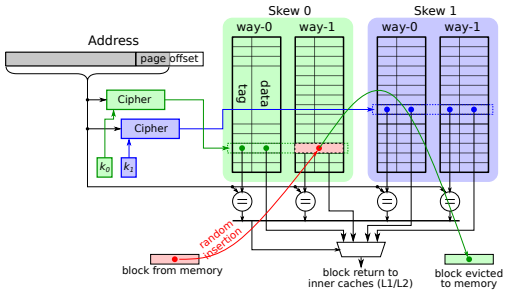

Figure 1. A randomized skewed cache with 2 skews over 4 ways.

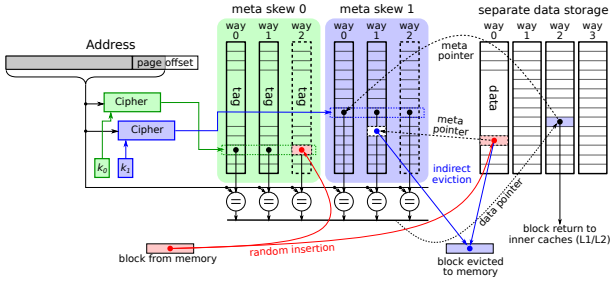

Figure 2. A Mirage cache with 2 skews over 6 ways of metadata (50% over-provided) and a separated 4-way data storage.

Generally speaking, the success of an attack depends on the effectiveness and the precision of the following four capabilities acquired by the attacker: *Sharing*: An attacker can access a cache set shared with her victim. *Controlling*: An attacker can control a cache set by either priming it with a sufficient number of congruent addresses or deterministically evict the victim's data from the cache set. *Monitoring*: An attacker can infer whether her victim has accessed the cache set by probing her own blocks or measuring the victim's access latency. *Implication*: An attacker can infer sensitive information by monitoring her victim's access.

2.2. Key Techniques Used in Cache Randomization

Table 1 summarizes the key techniques utilized in existing randomized caches, including *random mapping*, *random insertion*, *indirect eviction*, and *re-randomization*.

Random mapping is the basis for all randomized cache proposals. As shown in Figure 1, this is normally done by calculating the cache set for an incoming cache block using a cryptographic cipher [3]. This reduces attackers' capability to control and monitor cache sets by forcing them to search congruent addresses at runtime. It also limits attacker's implication capability as cache set indices no longer leak address bits. However, attackers still can collect congruent addresses using fast search algorithms, such as Group-Elimination [4], [10], [20], Prime-Prune-and-Probe (PPP) [11], [12], Conflict-Testing (CT) [4], [13], CT with Probe+Prune (CTPP) [21], Prune+PlumTree [22], and Write+Write [23].

Random insertion further reduces attackers' capability of controlling and monitoring cache sets [4], [6]. Also shown in

TABLE 1. SUMMARY OF THE KEY TECHNIQUES USED IN EXISTING RANDOMIZED CACHES.

Technique	Benefit	Implementation	Overhead
Random Mapping	Force attackers to search congruent addresses at runtime.	Index Randomization	Extra cache access latency of $1 \sim 5$ cycles. Extra bits for storing the full address tag in metadate [3].
Random Insertion	Reduced efficiency and increased noise in priming a cache set. Attackers either need extra congruent addresses to control a cache set or suffer from reduced attack precision.	Skewed caches (CEASER-S [4], ScatterCache [6], Chameleon [8])	Reduced efficiency for the replacement policies and complicated cache control logic.
		Parallel Index Mappings (PhantomCache [7])	Area overhead due to extra LLC banks. Increased power and control complexity due to simultaneous accesses to all banks.
		Load balance (Mirage [9])	Large storage overhead due to the over-provided metadata.
Indirect Eviction	Prevent attackers from identifying congruent addresses through cache evictions.	Global eviction using separate data storage (Mirage)	Extra storage for bidirectional pointers between metadata and data. Increased cache misses due to random replacement.
		VC and cache block relocation (Chameleon)	Increased power due to extra block relocation per each cache eviction. Increased cache misses due to random replacement.
Re-Randomization	Limit attack window by nullifying all congruent addresses already collected by the attacker.	Single-step remap [4]	Unnecessarily evict $\sim$50% cache blocks in each remap.
		Multi-step relocation [15]	Unnecessarily evict $\sim$10% cache blocks in each remap.

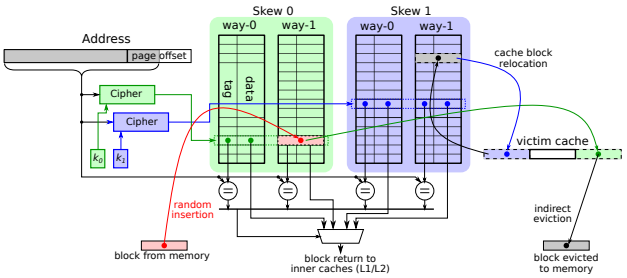

Figure 3. A Chameleon cache with 2 skews and a 3-way VC.

Figure 1, incoming cache blocks are inserted into randomly selected cache skews. Since it is extremely difficult to find fully congruent addresses,[2] attackers instead use a large number of partially congruent addresses to prime cache sets or evict victim's data, which leads to low success rate and undesirable disturbance to other unrelated cache sets. Mirage makes it more difficult by introducing load balance. As shown in Figure 2, extra (unused) ways are added to the metadata, which allows Mirage to estimate the level of conflict by counting occupied ways in each cache set and insert cache blocks into the less conflicted ones. This makes priming a cache set extremely difficult to achieve.

Although random insertion is effective in preventing cache sets to be primed, it cannot stop attackers from collecting congruent addresses from cache evictions, because the address evicting, or being evicted by, the target address is always congruent with the target address.[3] To prevent attackers from doing so, *indirect eviction* is adopted in Mirage and Chameleon by either enforcing a globally random replacement with the help of a separate data storage (Figure 2) or relocating the cache blocks in the VC back to the main cache array through a swap, as shown in Figure 3. Either way, the cache block evicted by a cache conflict is no longer congruent with the address causing the conflict. Currently, enforcing indirect eviction on all cache evictions is the only defense technique capable of fully thwarting CT.

In random caches without enforcing indirect eviction, attackers may gradually collect a large number of congruent addresses. Therefore, these caches must periodically *re-randomize* the random mapping by re-keying the ciphers, which effectively nullify the congruent addresses collected previously. Each re-key is immediately followed with a remap, where all cache blocks are relocated to their new cache sets based on the re-randomized mapping. However, $10 \sim 50\%$ cache blocks might be unnecessarily evicted due to the conflicts incurred by the remap [15].

Some of the key techniques cause structural disruption to the traditional set-associative cache structure. A majority of existing randomized caches [4], [6], [8], [9] adopt skewed cache structure to enforce random insertion, although skewed cache is not utilized in the caches of any modern processor, to our best knowledge. The performance benefit of skewed cache comes from its increased cache associativity [24]. However, such benefit diminishes in caches already with high associativity, such as LLCs [4]. In addition, skewed cache reduces the efficiency of the performance-oriented replacement policies, such as LRU and re-reference interval prediction (RRIP) [25], and complicates the metadata access control logic as all cache skews are looked up simultaneously for each cache access. It becomes even more disruptive in Mirage due to its over-provided metadata space and separated data storage, which effectively dismantle and restructure a traditional cache into two affiliated caches connected by pointers.

All the aforementioned key techniques incur performance overhead as well. The cipher used for the ran-

2. A fully congruent address shares the same cache set with the target in all cache skews, while a partially congruent one shares the same cache set in at least one but not all skews [11], [15]. The probability that a random address is fully congruent with the target is $1/S^P$ [4], where S, P are the numbers of cache sets and skew partitions. It becomes extremely difficult to find fully congruent address in caches with a large number of cache sets and multiple cache skews.

3. Since attackers rarely collect fully congruent addresses in randomized caches adopting random insertion, we no longer differentiate fully and partially congruent addresses in this paper. All congruent addresses collected on caches adopting random insertion are assumed partially congruent.

dom mapping typically prolongs the cache latency for 3∼5 cycles, although it can be reduced to just 1 cycle if a non-cryptographic hash algorithm is used instead [16]. Chameleon relies on cache block relocation for implementing indirect eviction, but the extra cache operations required to relocate cache blocks incur power overhead and consume extra cache bandwidth. The over-provided metadata space and separated data storage in Mirage incur ∼20% storage and power overhead [9]. Cache re-randomization leads to unnecessary data loss during each remap.

The increased cache miss rate due to the use of random replacement policy deserves a special notice. According to our estimation, replacing the LRU policy with a random one on a traditional set-associative cache leads to $5 \sim 20\%$ extra cache misses for memory heavy applications.[4] However, a number of randomized caches, including both Chameleon and Mirage, enforce the use of random replacement policy.

2.3. Extra Attacks Considered in This Paper

We consider two extra types of attack in the design of SeqAss. One is prefetch related attacks [27]. When a cache block is prefetched into the LLC, it is placed on the eviction candidate position to avoid cache pollution [27]. The block would be promoted only if it is later re-accessed. This feature is recently found exploitable for speeding up the CT algorithm, by prefetching rather than accessing the target address chosen by the attacker (Algorithm 2 in [27]). As reported in [27], CT-prefetch successfully reduces 80% search time on Intel Skylake and Kaby Lake processors.

Prefetch can be used to significantly speed up Prime+Probe as well [27]. Instead of priming the whole cache set, the attacker simply prefetches a single congruent address as it would be put at the eviction candidate position, exactly what Prime+Scope [13] achieves using thousands of cycles. The probe is also simplified as only one address needs to be probed. On randomized skewed caches, the attacker still needs to prime by prefetching multiple congruent addresses to ensure one of them is inserted into the target cache set, but this number is significantly lower than priming the whole cache set.

As described in the introduction, SeqAss supports the LRU replacement policy along with the described feature implemented on Intel processors and must resolve the extra vulnerability brought by LRU. Chameleon and Mirage are immune as they enforce random replacement policy.

The other one is the Write+Write (W+W) algorithm for searching congruent addresses [23]. W+W is special in its way of identifying congruent addresses. It is discovered that the write access latency is slightly prolonged when the write access operates simultaneously with another write accessing a congruent address. The probable cause is a hardware conflict inside the LLC as the two parallel write accesses compete for the same cache set. Since the timing different cause by this conflict is small (∼10 cycles), W+W suffers from a low level of accuracy. It needs to repeatedly test an address 30 times before deciding whether it is congruent.

Despite the low accuracy, it is important to notice that the timing difference is not caused by an LLC miss but a hardware conflict, which widely exists in hardware in various forms. This type of timing differences was considered too small to be detected until the discovery of W+W. More importantly, enforcing indirect eviction has no impact on search algorithms relying on such conflicts. Without a thorough timing analysis on the hardware implementation, no cache can be ruled out from W+W or a similar search algorithm yet to be discovered. However, almost all of the existing randomized caches are only proposals without full hardware implementation. For this reason, this paper takes a conservative approach by assuming that all randomized caches, including SeqAss, are potentially vulnerable to algorithms similar to W+W, although the exact search algorithm is unknown, and it is likely to be extremely slow and inaccurate. Therefore, attackers can slowly collect congruent address on all randomized caches, and thwarting existing search algorithms is no longer sufficient. SeqAss is designed to diminish the usefulness of congruent addresses.

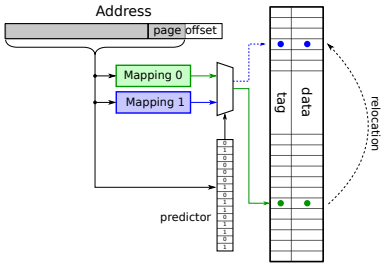

Figure 4. A predicative sequential associative cache.

2.4. Sequential Associativity

Sequential associativity [18], [28], [29] was originally proposed to reduce the miss rate of direct-mapped caches and make them behave like 2-way associative caches. Figure 4 illustrates a predicative sequential associative cache [18], which optimizes sequential associativity with a predictor. The cache is directly mapped using two mapping functions. The predictor is a table recording the mapping function used by cache blocks stored in the cache. For an access request, the cache produces two indices using both mapping functions and checks the cache block pointed by the predictor in its first trial. If the first check fails, the other cache block is checked subsequently. The requested cache block misses only when both checks fail. After this missing cache block is fetched from memory, it is stored using mapping 0. The cache block occupying the place is relocated using mapping 1, which consequently evicts the cache block originally stored at the relocated location. Since a cache block can be stored in two locations and the

4. Executing the SPEC CPU 2017 benchmark [26] on a baseline cache described in Table 3 of Section 5, we find that the average LLC miss rate increases 2.83% when the LRU replacement policy is replaced with a random one. However, this overhead increases to 6.5% when counting only those benchmark cases asserting higher than 1 access per 1K instructions to the DRAM, with 519.lbm incurs the highest MPKI overhead of 20.3%.

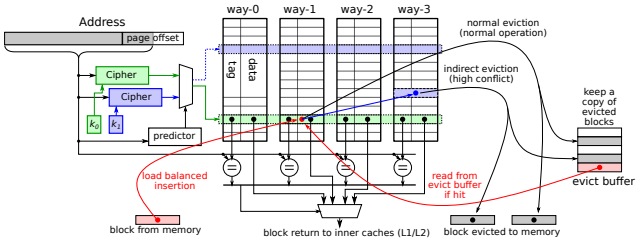

Figure 5. A SeqAss cache with four ways.

relocation process helps retain the cache blocks with good temporal locality, the directly mapped predicative sequential associative cache achieves a similar performance with a 2-way associative cache. [18]

3. Threat Model

We assume the attacker and her victim run simultaneously on separate address spaces. The attacker launches conflict-based cache side-channel attacks targeting the inclusive LLC. The attacker has deciphered the virtual-to-physical address mapping and has the full design of the hardware. Since no data is shared between the attacker and her victim, the attacker cannot access data belonging to her victim. The attacker can access an unlimited number of random addresses and measure time spans using high-precision timers [30]. We also assume the attacker may have successfully collected a certain number of congruent addresses in a persistent attack using an unknown search algorithm similar to W+W. Conflict-based attacks targeting inner cache levels, such as L1 [30], reuse-based attacks, such as Flush+Reload [31], and cache occupation attacks [32] are out of the scope of this paper. Conflict-based LLC side-channels have recently been utilzied in attacks on confidential computings, but existing attacks [33] still rely on a deterministic mapping of cache index and fail on a randomized LLC.

4. Design of the *SeqAss* Randomized Cache

This section applies sequential associativity to set-associative caches and optimizes it for both security and performance. The proposed SeqAss cache is depicted in Figure 5, and the techniques utilized in SeqAss are summarized by Table 2. The original structure of the predicative sequential associative cache is repurposed to a randomized set-associative cache by replacing the two mapping functions with two non-linear ciphers. In addition, several design features are either newly proposed or retrofitted to enhance defense, and reduce cache misses and power overhead: a *load balanced insertion* scheme to enhance random insertion, enforcing *partial relocation* rather than full relocation to save power, using *conflict counters* to estimate conflict levels with a tiny area footprint, an *evict buffer* to detect active search algorithms, and a *on-demand remap* scheme to reduce data loss caused by remaps. All the randomization techniques described in Table 1 have been implemented. Without incurring significant performance overhead, SeqAss preserves the set-associative structure and the LRU replacement policy, while reduces both cache misses and memory accesses.

4.1. Load Balanced Insertion

As depicted in Figure 5, two ciphers[5] are used to calculate two random cache sets (colored in blue and green) in parallel for each incoming cache block (colored in red). Instead of statically selecting one cache set as defined in the original sequential associativity [18], SeqAss implements a load balanced insertion scheme which selects the cache set with the lower level of conflict. Each cache set is attached with a conflict counter, which estimates the conflict level of the cache set by counting the number of recent conflicts. Assuming the conflict level of the green cache set is lower than that of the red one, the incoming cache block is inserted into the green cache set, as shown in Figure 5. When too many cache blocks are squeezed into a single cache set, the rising number of cache conflicts pushes up its conflict level, and following cache blocks are diverted to other cache sets.

The value of a conflict counter increases by one when a conflict occurs on its monitored cache set. It is designed as a 3-bit saturated counter, as later shown in Algorithm 2, and gradually decays by factors untouchable by attackers to avoid being manipulated. The width of the counter affects the defense strength. Wider counters result in more balanced insertion and stronger defense, but incur higher area overhead. Our evaluation in Section 5.1 demonstrates that using 3-bit counters provides a defense comparable to Mirage while widening the counters to 4-bit would make SeqAss stronger than Mirage.

As shown in Algorithm 1, all counters decay sequentially by a hardware-controlled index d (line 2, 3). When a relocation occurs (described soon in Section 4.2), the counter monitoring the relocation destination cache set also decays by one (line 4). If an attacker can manipulate the decay process, e.g. speeding up the decay for certain cache sets, she may get favorable advantage in triggering conflicts on these sets. Since the hardware-controlled index d is inherently ignorant to access pattern, and the relocation destination cache set is chosen by the random mapping and evenly distributed to all cache sets, Algorithm 1 is effective in fixing the relocation rate, i.e. the ratio of relocations to evictions, ignorant to access patterns. As later shown in Figure 15d, the relocation rates of running different SPEC 2017 benchmark cases concentrate around 15.6% with a small standard deviation of 0.77%, while the various SPEC 2017 benchmark cases present widely different memory access patterns. This is a good indication that the relocation process, under the control of the decay process, maintains a good randomness and is difficult to be affected by specific access patterns. Without

5. This paper does not discuss the implementation of these ciphers or their impact on IPC as this has been carefully discussed in [9], [16] Either a multi-cycle cipher [36] or a single-cycle hasher [16] would be OK.

TABLE 2. TECHNIQUES UTILIZED IN THE SEQASS CACHE.

Technique	Initial Idea	Security Benefits	Performance Benefits	Other Note
Sequential Associativity	[28]	Implement random mapping and random insertion.	Reduce cache misses due to increased cache associativity.	Used for defense for the first time.
Load Balanced Insertion	Mirage [9]	Weaken Evict+Time.	Reduce cache misses by balancing the distribution of cache blocks.	Significantly lower area overhead than Mirage.
Partial Relocation	[28]	Defeat Prime+Probe and further weaken Evict+Time.	Reduce cache misses by re-balancing the distribution of cache blocks.	Use partial relocation to reduce power overhead.
Conflict Counter	This work	Enable load balanced insertion and partial relocation.	Identify cache sets with a high conflict level.	Key technique to enable load balance with low area overhead.
Evict Buffer	[34]	Trigger on-demand remaps and slow down CT-prefetch.	Reduce memory accesses.	A directly mapped buffer significantly smaller than that in [34].
On-Demand Remap	SP2021 [15]	Limit the congruent addresses collected by attackers.	Reduce cache misses by re-balancing the distribution of cache blocks.	Use postponed remap [35] to further reduce data loss.

Algorithm 1: Decay of the Conflict Counters

```
Input: r, whether to relocate a block;
Input: t, the relocation destination cache set.
Input: CL[], the conflict counters for all cache sets.
function decay(r, t, CL)      // triggered by cache conflicts
    d = (d + 1) % S   // d is a hardware-controlled index
    if CL[d] > 0      then CL[d]-- // decay cache set d
    if r and CL[t] > 0 then
        CL[t]-- // decay relocated cache sets
    end
end
```

a method to maliciously affect the relocation rate, the only way left to manipulate the decay process is to introduce random cache evictions, causing unfavorable noise to all attacks.

Load balanced insertion is a strong defense technique. It significantly increases the difficulty in occupying a target cache set or precisely evicting a victim's cache block, where the latter is a key requirement for Evict+Time attacks. Without load balanced insertion, such as in CEASER-S, a congruent address has a 50% chance to be inserted to the correct cache set sharing with the target address. When load balanced is enforced, even if an attacker has collected a large number of congruent addresses (cache blocks), accessing them would unavoidably raise the conflict level of the target cache set, which diverts following addresses to other cache sets and in turn reduces the probability of successfully inserting attacker's cache blocks into the target cache set to almost 0%. As shown in Figure 6, this load balanced insertion increases the number of congruent addresses required to evict a victim's cache block by ∼31 times comparing to a 2-skew CEASER-S. In addition, we can consider that randomizing the original sequential associative cache results in a cache equivalent to a CEASER-S cache supporting the full relocation. This added full relocation indeed pushes up the number of congruent addresses to ∼1000 according to Figure 6. However, it is not strong enough to thwart Evict+Time and it is weaker than load balanced insertion even without relocation.

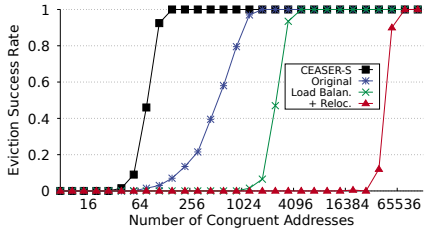

Figure 6. The success rate of evicting a randomly selected cache block by accessing various numbers of congruent addresses. All caches are 16384-set and 16-way. All results are averaged from 200 repeated tests. *Original*: randomizing the original sequential associative cache, *Load Balan.*: SeqAss cache with load balance but no relocation, + *Reloc.*: SeqAss cache with load balance and partial relocation.

Takeaway 1: *As a strong defense, load balanced insertion significantly increases the number of congruent addresses required for launching Evict+Time attacks.*

***Performance benefit*:** Both Mirage and SeqAss implement load balanced insertion. However, Mirage incurs over 20% area overhead while only 3.7% extra area is required by SeqAss. Such an enormous difference is caused by the different ways in estimating conflict levels. Mirage estimates the conflict level of a metadata cache set by counting the occupied ways, which requires the metadata cache set to be 50∼75% over-provided and the data array to be separated from the metadata array [9]. The over-provided metadata space and the pointers used to connect metadata and data lead to the heavy area overhead. In SeqAss, conflict level is estimated by a small (3-bit) conflict counter attached to each cache set, which causes a negligible area overhead.

In addition, the load balanced insertion doubles the available cache ways for each cache block and actively reduces the conflict level of all cache sets, both of which help reduce cache misses. Our evaluation in Section 5.4 shows that applying load balanced insertion on a traditional set-associative cache reduces 4.29% cache misses.

Unlike cache skews, the two cache sets in Figure 5 are checked sequentially for each access request. If the first one results in a hit, checking the second one can be avoided. The predictor utilized in the predicative sequential associative

cache [18] is also applied to predict the correct cache set for cache hits. It is formed as a hashed bitmap, where bits are indexed by hashing addresses. Whenever a cache block is inserted into a cache set or relocated to another one, the correct cipher id (1-bit) is stored in the bitmap. Therefore, the correct cipher can be chosen by reading this bitmap in parallel with calculating the two cache sets for each access request. Our experiment shows that a bitmap large enough to hold only 1-bit per block in the cache achieves a 95% prediction rate. As a result, only 5% of the hit cache accesses suffer from a second metadata check.

4.2. Partial Relocation

Sequential associativity [18] has already supported indirect eviction by enforcing a full relocation scheme. When a cache block is to be evicted due to a conflict, it is relocated to another cache set using the other cipher as depicted in Figure 5. The cache block at the eviction candidate position of the relocation destination cache set is evicted instead.

Although full relocation is effective in thwarting the CT search algorithm, relocation is a power consuming operation. Each relocation incurs two accesses to the cache array to swap both metadata and data, which push up the energy for each cache miss by ∼60% and reduce the available bandwidth of the cache. SeqAss implements a partial but possibly chained relocation scheme where only those cache blocks to be evicted from cache sets with a high conflict level are relocated. Cache blocks from cache sets with a low conflict level can be evicted out of the cache as normal. The total number of relocations is reduced by 85% compared to the full relocation scheme.

We use Figure 5 to explain the partial relocation scheme. Since the green cache set is full, inserting the incoming cache block (colored in red) causes a conflict, and the cache block on way-1 is to be evicted. If the conflict level is higher than a pre-defined conflict threshold (*ct*), the cache block is relocated to another cache set (blue arc); otherwise, it is evicted as normal. If a relocation indeed occurs, the relocated block is inserted at the eviction candidate position to avoid polluting other cache blocks in the destination cache set. As shown in Figure 5, the original eviction candidate (the blue block on way-3) is evicted, if the conflict level of the destination cache set is low. Otherwise, SeqAss would continue to relocate this block until a block from a cache set with a low conflict level is evicted. Note that relocation would not prolong cache access latency as it is processed in the background.

Partial relocation makes it significantly harder for attackers to evict victim's cache block in Evict+Time attacks. Let us consider the example depicted in Figure 7. Assuming an attacker has successfully pushed the victim's cache block (colored in blue) into the eviction candidate position on the victim's cache set 1, she would evict it out of the cache by accessing another congruent address (colored in red). However, the conflict levels of both cache sets of the victim must have been arisen by the attacker's behavior. Consequently, the victim's cache block is relocated to the other cache set

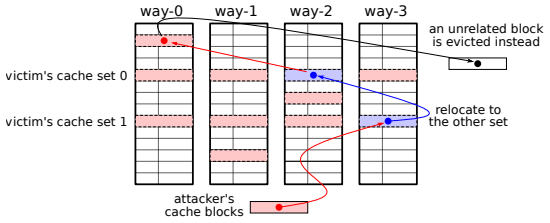

Figure 7. Chained relocation keeps victim's cache block relocated rather than evicted in Prime+Probe attacks.

also with a high conflict level, which triggers a chained relocation resulting an unrelated cache block from another cache set to be eventually evicted. Continuously accessing more congruent addresses maintains the high conflict level of both victim's cache sets and the victim's cache block would be constantly relocated rather than evicted. As shown in Figure 6, the number of congruent addresses required for evicting a victim's cache block increases by another ∼22 times when relocation is enabled (∼700 times compared to 2-skew CEASER-S). The number of addresses (∼50K) is so large that they would have occupied ∼23% of the LLC with enormous noise. Evict+Time remains possible only in theory.

Partial relocation thwarts Prime+Probe attacks. In such attacks, an attacker needs to prime the target cache sets before triggering her victim's access. Afterwards, she probes the addresses used in the prime. Missing anyone of them indicates an access by the victim. To reduce noise and increase attack frequency [13], cache sets should be quietly primed without affecting other unrelated cache sets, and attackers cannot wait too long before triggering the victim's run. The combination of load balanced insertion and partial relocation not only makes priming cache sets extremely difficult and noisy, but also pushes up the conflict level of the primed cache sets. In most situations, the target cache sets are not sufficiently occupied, and an unrelated cache block is evicted (i.e., affected) by the following victim's access, failing the final probe. Even if an address used in the prime is indeed pushed out from the target cache sets, an unrelated cache block is likely to be eventually evicted.

Takeaway 2: *The combination of load balanced insertion and partial relocation thwarts Prime+Probe attacks and makes Evict+Time attacks significantly harder.*

***Performance benefit*:** Although load balance and cache block relocation are existing techniques utilized by Mirage and Chameleon, respectively, SeqAss is the first to combine and apply them in a way that balances defense and power overhead. Instead of blindly forcing all evicted cache blocks to be relocated as defined by the original sequential associativity (also in Chameleon) and consumes high dynamic power, SeqAss relocates only those cache blocks evicted from cache sets with a high conflict level. This is sufficient to thwart Prime+Probe attacks while the number of relocation is reduced by 85%.

Partial relocation helps reduce conflict misses [37] as

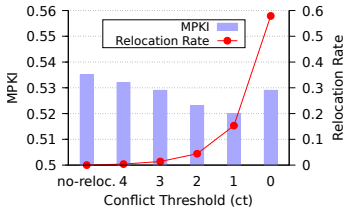

Figure 8. The LLC MPKI and relocation rate using different conflict thresholds when running SPECrate CPU 2017. Each conflict counter is 3-bit wide and saturates at 7.

Algorithm 2: Update of the Conflict Counters

Input: s, the conflicted cache set
Input: *pref*, whether this is a prefetch;
Input: *vbuf*, whether this access hits in the evict buffer;
Input: `CL[]`, the conflict counter for each cache set.
Input: `C_MAX = 7`, the maximum counter value.

```
1 function update(s, pref)   // triggered by cache conflict
2     if pref and CL[s] < ct + 2 then CL[s] = ct + 2
3     else if vbuf               then CL[s] = C_MAX
4     else if CL[s] < C_MAX      then CL[s]++
5 end
```

well. Such misses occur when multiple cache blocks, with good temporal locality, are mapped to the same cache set and competing for the limited space. When these conflict misses push up the conflict level of a cache set over the conflict threshold, the relocation for those competing cache blocks is enabled to avoid further misses. However, not all evicted blocks are with good temporal locality and excessive relocation leads to substantial power overhead. Estimated using the SPECrate CPU 2017 benchmark [26], Figure 8 reveals the variation on the LLC misses per 1K instructions (MPKI) and the number of relocations with different *ct* values. Using 3-bit wide counters and setting $ct = 1$ achieve the maximal MPKI reduction of 2.8%. Only 15.3% of all evictions are relocated. Further relocations by reducing *ct* to 0 return no benefit. Therefore, *ct* is fixed to 1. This small *ct* has a security benefit as well: Just two conflicts would be enough to push up the conflict level and enable relocation for a long period, which is hard to avoid in the prime process.

4.3. Thwart Prefetch-Based Prime+Probe

For performance benefits, SeqAss uses the LRU replacement policy and implements almost the same prefetch operation as Intel processors [27]. When a cache block is prefetched and missing in the LLC, it is inserted into a cache set at the eviction candidate position. The LLC does nothing if the prefetch is a hit. This implementation may allow attackers to launch the prefetch related attacks described in Section 2.3.

To thwart the prefetch-based Prime+Probe attack, SeqAss raises the conflict level to at least $ct + 2$ for missing prefetches, as described on line 2 of Algorithm 2. This is a careful balance between safety and performance. Assuming a cache block is prefetched into the target cache set by an attacker, pushing up the conflict level immediately enforces relocation for the prefetched cache block. Therefore, it is relocated rather than evicted by the victim's access, and the following probe would fail. If the attacker tries to deliberately decay the counter, the minimal number of random evictions required to demote relocation is $\sim S$, the number of total cache sets, since reducing the counter value from $ct + 2$ to ct requires the hardware-controlled index d to overflow twice by the sequential decay process. However, S random evictions should have already triggered an access to the target cache set, which would relocate the prefetched cache block. Experiments show that relocation is normally demoted after $1.5 \cdot S$ evictions. At this time, the prefetched block is likely relocated and then evicted.

Takeaway 3: *By pushing up the conflict level to at least $ct + 2$ for the cache set with a prefetched cache block, SeqAss thwarts prefetch-based Prime+Probe attacks.*

4.4. Evict Buffer and On-Demand Remap

Since only 15.3% of evictions are relocated, SeqAss cannot fully stop attackers from collecting congruent addresses using CT-like algorithms (CT and CT-prefetch). However, thwarting all CT-like algorithms may not be secure enough anyway. As described in Section 2.3, attackers may slowly collect congruent addresses using a yet unknown search algorithms, similar to W+W, in the future. Instead of stopping attackers from collecting any congruent addresses, SeqAss endeavors to prevent attackers from using them to launch attacks. Prime+Probe attacks have already been thwarted according to Section 4.2 and 4.3. Evict+Time remains possible only in theory as it would require tens of thousands of congruent addresses.

To prevent attackers from collecting large numbers of congruent addresses using existing search algorithms, an on-demand remap scheme is implemented in SeqAss to keep the number of usable congruent addresses collected by attackers far below the required number for any attacks. A remap is triggered whenever an active search algorithm is detected. This detection utilizes a well-known ping-pong pattern [38], [39], [40], [41] unavoidably asserted by all search algorithms: The target address (cache block) is evicted and later fetched back multiple times during a search.

We build our detector by retrofitting the evict buffer used in TreasureCache [34], which is an incoherent buffer used to prevent Flush+Reload attacks by caching the recently evicted and flushed cache blocks and reducing their reload latency. In SeqAss, a similar evict buffer is added to detect the ping-pong pattern, as depicted in Figure 5. Whenever a cache block is evicted from the LLC, it is copied into the evict buffer. If this block is later re-accessed, it is fetched directly from the evict buffer, and one ping-pong access is recorded. Unlike in TreasureCache, flushed blocks are not copied to the evict buffer; otherwise, the attacker can easily flushing the evict buffer. We deliberately configure the buffer as (randomly) directly-mapped with only $\sqrt{S}$ blocks, rather than a large and fully associative one as used in TreasureCache. This both reduces area overhead and improves

defense. Let us assume that the cache block storing the target address is successfully evicted and copied to the evict buffer, the attacker may want to hide her ping-pong access and flush the target address out of the evict buffer before refetching it. If this is done by random evictions, a large number of them would be required, because each eviction has only a $1/\sqrt{S}$ probability of success. The noise would be unbearable. Even if the attacker has collected an address (cache block X) congruent with the target address (cache block T) in the evict buffer, not to mention this is highly unlikely, she would need to access tens of thousands of addresses congruent with X to force X into the evict buffer and eventually flush T, which is even worse than random evictions. With $\sqrt{S}$ blocks, the storage overhead of the evict buffer is $<$0.05% for a 16-way cache, significantly lower than the 0.45% storage overhead incurred by TreasureCache.

This evict buffer is also used to deliberately slow down CT-prefetch. To collect multiple congruent addresses, the target address is constantly re-fetched and unavoidably hits in the evict buffer. SeqAss utilizes this hit to raise the conflict level of the cache sets storing the target address to the maximum value (line 3 in Algorithm 2). This significantly prolongs the relocation period of the victim's cache set. As later shown in Table 7, the search speed of CT-prefetch is reduced to CT on SeqAss.

The number of ping-pong accesses is recorded by a counter attached to each cache set. When the counter exceeds a pre-defined attack threshold (*at*), a remap is triggered. The two ciphers are re-keyed sequentially. In each remap, one of the ciphers is re-keyed, and all cache blocks mapped using this cipher are gradually relocated to use the new key or the other cipher. As a result, more than half[6] of the congruent addresses collected by the attacker are nullified. The upper bound of the number of collectible congruent addresses is roughly proportional with *at*. We heuristically set *at* to 32 to narrow each attack counter to only 5 bits and limit the maximal number of congruent addresses collectible to under 80 for all CT-like algorithms,[7] as shown in Figure 9. This number is far from enough for launching any reasonable attacks.

Takeaway 4: *On-demand remap keeps the congruent addresses collected by attackers far below the necessary number required for launching any reasonable attacks.*

Since SeqAss does need to finish a remap as quickly as in CEASER-S, it adopts a postponed remap approach proposed in the RollingCache [35]. A spare cipher is added and re-keyed waiting for the next remap. When a remap is triggered, the cipher to be re-keyed is immediately swapped with the spare cipher. To allow normal accesses to the cache blocks using the old cipher, we adopt the same indexing scheme proposed in CEASER-S to allow both ciphers to

6. Newly found congruent addresses distribute evenly between keys but all of the congruent addresses surviving from a remap use the old key. This results in an unbalanced distribution towards the old key, which is to be re-keyed by the next remap, and the $>$50% nullification rate.

7. The result for W+W is not shown as it fails to collect any congruent address during a remap period in its current form.

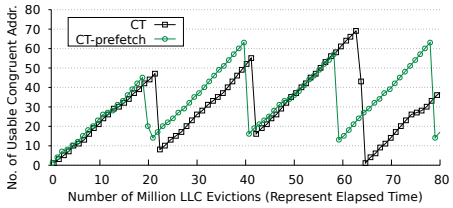

Figure 9. The number of usable congruent addresses collected using CT-like search algorithms on a 16-way 16384-set SeqAss cache (*at* = 32).

be accessible until the remap is finished [3]. Instead of immediately relocating the cache blocks mapped by the old cipher, which would incur 10$\sim$50% data loss, SeqAss waits until most of these blocks are naturally evicted by incoming cache blocks. To do this, the number of cache blocks mapped using each cipher is constantly monitored by three global counters. When the percentage of cache blocks mapped by the old cipher drops below 0.5%, SeqAss starts the relocation process using the same method used in CEASER-S [3]. Therefore, the maximum data loss is reduced to 0.5%. This wait is not long during active attacks, because the attacker is still busy accessing a large number of random addresses. Of course, the attacker may try to stop a remap by retaining a cache block through repeated accesses. SeqAss currently avoids this by setting a maximal waiting time of $16 \cdot N$ LLC accesses, where N is the number of blocks in the LLC.

The evict buffer makes CT and CT-prefetch even harder to use. A typical probe test needs to differentiate cache hits (normally L1 hit) from cache misses (normally LLC miss), where the latency gap between these two is longer than the DDR access time. When the evict buffer is used, the target address is fetched form the evict buffer rather than the memory. The latency gap reduces to the access time of a hit in the LLC, which is significantly shorter than the DDR access time.[8]

***Performance benefit*:** Both the evict buffer and the on-demand remap bring performance benefits. The evict buffer behaves similarly to a VC, which is able to reduce conflict misses and memory accesses. The data loss caused by on-demand remap is reduced to a negligible level thanks to the postponed remap. For normal applications, remap is triggered only when a high number of conflict misses strike on a small number of cache sets. Remap in this situation would rebalance the distribution of data and potentially reduce future conflict misses. According to our estimation later in Section 5.4, adding the evict buffer reduces LLC misses by 1.35% and applying on-demand remap reduces them further by another 3.51%.

5. Security and Performance Analysis

We have implemented the SeqAss cache on FlexiCAS (Flexible Cache Architectural Simulator) [42], along with

8. The access time of a hit in the LLC of Intel i7-9700 is measured around 30 cycles, while the DDR access time is around 150 cycles.

TABLE 3. PROCESSOR AND LLC PARAMETERS

Processor	Spike, assume freq = 3GHz, IPC=2
L1-I/D	32KB per core, 8-way, LRU
L2	256KB per core, 4-way, LRU, MSI, exclusive
All LLC	16MB shared (16384-set), MESI, inclusive, directory
Baseline	16-way, LRU
SP2021	16-way, LRU, remap @10EV+detect
CEASER-S	2x 8-way partition, SRRIP, remap @100ACC
Chameleon	2x 8-way partition, Random, 8-way VC
Mirage-50	2x 12-way partition, Random, 3-step relocation
Mirage-75	2x 14-way partition, Random, no relocation
SeqAss	16-way, LRU, *ct*=1, remap @detect, *at*=32

other five cache structures, i.e. a *baseline* non-randomized set-associative cache, a randomized set-associative cache (*SP2021* [15]), a randomized skewed cache (*CEASER-S*), the *Chameleon* cache and the *Mirage* cache. We have reproduced various search algorithms (CT, CT-prefetch, CTPP and PPP) and attack methods (Evict+Time, Prime+Probe and prefetch-based Prime+Probe) executable on FlexiCAS. This cache model has also been incorporated into the ISA level simulator Spike [43] to form a behavioral processor simulator (Spike-FlexiCAS) capable of running Linux. This simulator is available from https://github.com/comparch-security/spike-flexicas.

We evaluate the runtime performance of the cache structures using the SPECrate CPU 2017 benchmark suite [26].[9] It is compiled using Speckle [44] and RISC-V GCC [45], and running on a RISC-V Linux compiled using a Spike simulator SDK [46]. The detailed parameters for the simulated processor and the various cache structures are presented in Table 3. The processor frequency and IPC are used to estimate the power consumption.

We choose Spike-FlexiCAS over the commonly used cycle-accurate Gem5 due its modular implementation and high simulation speed. Spike-FlexiCAS allows us to quickly reproduce all the recent randomized caches and configure a three-level cache hierarchy according to Intel Core processors, both of which would be difficult or even impossible using Gem5 [9]. The 5x simulation speed over Gem5 [42] allows us to run various persistent attacks (>100G cache accesses) and 10G instructions per SPEC benchmark case rather than the maximal of 1G instructions in existing papers [6], [8], [9]. The cache miss rate extracted from longer simulation runs is a more accurate estimate of realistic performance. The lack of IPC overhead is a drawback, but LLCs impact IPC mainly through miss rate. We believe that it is favorable to trade in IPC estimation for the benefits of wide structure coverage, practical cache hierarchy and realistic miss rate estimation.

9. Spike adopts a batched execution scheme to speed up the simulation for multi-core processors, which does not support the frequent cross-core synchronization asserted by OpenMP. As a result, the *CPU speed* subset of SPEC CPU 2017 does not stress the cache model properly. Only the *CPU rate* subset is used. Multi-core performance is analyzed in the Appendix.

TABLE 4. PARAMETERS FOR BUCKET-AND-BALL MODEL

	Mirage-75	SeqAss
Balls	256K	256K
Buckets	32K	16K
Bucket Capacity	14 (75% extra)	16
Conflict Level (CL)	balls in bucket	conflict counter value
Increase CL	insert a new ball	insert a new ball
Decrease CL	indirect eviction	conflict and relocation
Success Condition	evict target	evict target

5.1. Effectiveness of Load Balance on Evict+Time

Both Mirage and SeqAss implement load balanced insertion. Mirage provides two cache set indices by dividing the cache into two skews and estimates the conflict level of each cache set using over-provided metadata space. SeqAss also provides two cache set indices using sequential associativity and estimates the conflict level of each cache set using dedicated conflict counters. To compare the effectiveness of these two schemes, we reuse the bucket-and-ball model [9] open-sourced by Mirage and verifies the result on our behavioral cache model.

The bucket-and-ball model was used by Mirage to estimate the number of random addresses required to cause a set-associative eviction (SAE). By over-providing 75% metadata space, Mirage claims that 10^{34} random addresses would be needed to cause an SAE. Although we agree with the estimation, we argue that this may not be sufficient to deprive attackers of their capability of collecting congruent addresses. As described in Section 2.3, we cannot fully rule out the possibility that attackers might slowly collect congruent addresses by exploiting the potential time differences caused by hardware conflicts, similar to W+W, without relying on SAEs. We would like to assume that an attacker has already collected a number of congruent addresses, and evaluate the probability of evicting a target address using these congruent addresses.

We have added SeqAss into the bucket-and-ball model and changed the test condition to throw balls congruent with the target. The detailed parameters are listed in Table 4. Mirage-75 is configured with 75% over-provided metadata space and 2 skews. In each test, a randomly selected target ball is thrown into buckets after an initial warm-up. A number of congruent balls are then thrown into buckets to evict the target ball. While Mirage directly estimates the conflict level of each bucket using the number of balls remaining the bucket, SeqAss maintains a conflict counter for each bucket according to Algorithm 1 and 2.

The test result is depicted in Figure 10. On Mirage-75, the target ball begins to be evicted after 4K balls. The success rate rises to 50% with ∼50K balls, and approaches to 100% with ∼100K balls. The curve for SeqAss is much steeper than Mirage. The target ball begins to be evicted after 32K balls and the success rate approaches to 100% with ∼60K balls. SeqAss is safer than Mirage when the number of congruent balls is smaller than 32K but suffers from a lower ball count for the 100% eviction rate. The partial

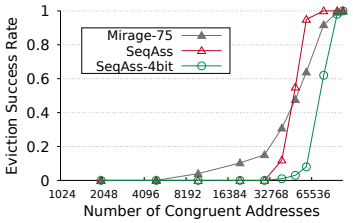

Figure 10. The success rate of evicting the target ball using various numbers of congruent balls. Each result is averaged from 100 bucket-and-ball tests.

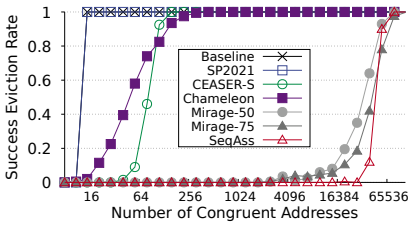

Figure 11. The success rate of evicting a victim's data using different numbers of congruent addresses. Each result is averaged from 200 repeated tests targeting the same victim's data.

TABLE 5. POSSIBILITY FOR PRIME+PROBE ATTACKS

Structure	No. of Addresses	Precision	Recall	F1 Score	Possibility
Baseline	16	100%	100%	1	Definitely.
SP2021	16	100%	100%	1	Definitely.
CEASER-S	67	100%	100%	1	Definitely.
Chameleon	14	70%	1.4%	0.027	Unlikely.
Mirage	512	0%	0%	0	Unlikely.
SeqAss	512	0%	0%	0	Unlikely.

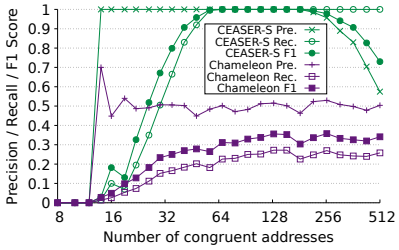

Figure 12. The F-score of detecting a victim's access by Prime+Probe attacks using various numbers of congruent addresses. Each result is collected from 500 positive and 500 negative tests.

relocation in SeqAss does not allow balls from buckets with a high conflict level (CL) to be evicted. A post-mortem analysis shows that the CLs of the two buckets associated with the target ball are quickly raised to the maximal value of 7 in the early stage of each test. When the number of congruent balls becomes sufficiently large, long relocation chains begins to emerge and decay the CLs of target buckets to levels below *ct*, leaving a short window allowing the target ball to be evicted. An easy enhancement is to widen the conflict counter by one bit (raise the maximal CL from 7 to 15), which would make SeqAss stronger than Mirage, as depicted by SeqAss-4bit in Figure 10.

We have conducted a similar eviction test on all randomized caches using our cache model. The result is shown in Figure 11. Since attacking with tens of thousands of congruent addresses is both extremely slow and unbearably noisy, we argue that Mirage and SeqAss are safe from Evict+Time attacks. Even without widening conflict counters, SeqAss has already provided a defense comparable to Mirage. All of the other random caches are clearly vulnerable.

The result of SeqAss matches perfectly with the bucket-and-ball test, but Mirage-75 shows a slightly weaker defense. We find out that Mirage is more sensitive to the warm-up between tests than SeqAss. A bucket-and-ball test always starts from empty buckets, but tests on cache model rely on a warm-up of accessing random addresses twice the size of the LLC. This is not sufficient to remove the unbalanced conflict levels left by previous tests in Mirage, leading to the weaker result. This is unlikely to be exploitable by attackers in practice.

In addition, we are surprised to find that the defense provided by Chameleon is even weaker than CEASER-S. The target address is likely evicted by accessing only 64 congruent addresses. By analyzing the simulation traces, we find out that the oblivious (random but not load balanced) relocation distribution allows the target address to be swapped into the VC and later pushed out the cache by an indirect eviction. With a sufficient number of congruent addresses, the VC begins to relocate the attacker placed addresses back to the main cache array. Since these addresses are congruent, each of them has a 25% probability (assuming 2-skew) to be relocated back to the cache set storing the target address, where it gets a $1/16$ probability of swapping the target address into the VC. Once this occurs, the target address is stuck in the VC and doomed to be pushed out by a later indirect eviction.

In summary, Mirage and SeqAss are the only caches reasonably safe from Evict+Time attacks. The defense provided by SeqAss is comparable to Mirage. All of the other random caches are vulnerable once the attacker has slowly collected tens of congruent addresses.

5.2. Effectiveness on Prime+Probe

In a Prime+Probe attack, the attacker primes the target cache set with her own congruent addresses and expect that the following victim's run knocks one of her addresses out of the cache. To evaluate the possibility of launching Prime+Probe attacks, we run multiple Prime+Probe attacks with different numbers of congruent addresses. These attacks are evenly divided into positive ones, where the target address is indeed accessed by the victim, and negative tests, where a random address is accessed. Table 5 reveals the F-score [47], and the variance of F-score on CEASER-S and Chameleon is depicted in Figure 12.

Baseline and SP2021 caches are obviously defenseless. CEASER-S is not much better, as priming with 64∼200 congruent addresses achieves the perfect F-score (F1 = 1). Prime+Probe attacks are unlikely to succeed on Mirage or

TABLE 6. POSSIBILITY FOR PREFETCH-BASED PRIME+PROBE ATTACKS

Structure	No. of Addresses	Precision	Recall	F1 Score	Possibility
Baseline	1	100%	100%	1	Definitely.
SP2021	1	100%	100%	1	Definitely.
CEASER-S	12	97.6%	98.6%	0.981	Definitely.
Chameleon	16	52.9%	43.2%	0.476	Unlikely.
Mirage	110	0%	0%	0	Unlikely.
SeqAss	110	0%	0%	0	Unlikely.

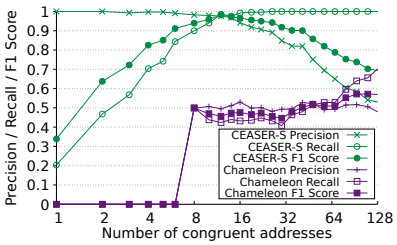

Figure 13. The F-score of detecting a victim's access by prefetch-based Prime+Probe attacks using various numbers of congruent addresses. Each result is collected from 500 positive and 500 negative tests.

SeqAss even with 512 congruent addresses. Priming with more congruent addresses is normally unfavorable, since it would further bring down the attack frequency and accuracy, both of which are essential to Prime+Probe attacks. Prime+Probe on Chameleon may bring some visibility to attackers. By priming with 14 congruent addresses, an attacker may observe her victim's access (precision $> 50\%$) at a very low recall rate (1.4%). Since this visibility is highly affected by noise, direct Prime+Probe attacks are unlikely to succeed. However, attackers may use Prime+Probe for cache occupation attacks [48], [49] (out of the scope of this paper). With only 48 congruent addresses, the attacker obtains a $\sim$20% recall rate to detect whether a memory access occurs.

We have assessed the prefetch-based Prime+Probe attack in a similar fashion and the results are presented in Table 6 and Figure 13. As expected, the number of congruent addresses is significantly reduced, which allows the attacker to monitor her victim's accesses at a much higher high frequency [13]. Baseline, SP2021 and CEASER-S remain defenseless, while SeqAss and Mirage remain safe. On Chameleon, prefetch-based Prime+Probe provides no visibility, but it is easier to launch occupation attacks than Prime+Probe. The required number of congruent addresses is reduced to 8 and the recall rate rises to 50%.

5.3. Effectiveness on Search Algorithms

Table 7 demonstrates the success rate of collecting congruent addresses using various search algorithms. It is found that CT and CT-prefetch are the only algorithms continuing to work after enforcing random insertion. CT-prefetch is much faster than CT due to the reduced number of evictions. No search algorithm works on Mirage and Chameleon. Since SeqAss enforces partial rather than full relocation, it cannot

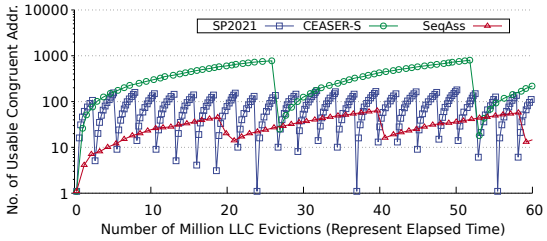

Figure 14. The number of usable congruent addresses collected using CT-prefetch on SP2021, CEASER-S and SeqAss when remap is enabled.

stop but only slow down CT or CT-prefetch. CT-prefetch shows no speed benefit against CT on SeqAss.

Although attackers can collect congruent addresses on a SeqAss cache, the on-demand remap restricts the number of usable congruent addresses collectible by an attacker. Figure 14 records the number of usable congruent addresses collected by attackers using CT-prefetch on SP2021, CEASER-S and SeqAss. Once a remap is done, the addresses previously congruent with the target address become non-congruent, and the number of usable congruent addresses drops sharply. However, attackers can easily collect more than 50 and 400 congruent addresses on SP2021 and CEASER-S, respectively. These are more than enough for both Evict+Time and Prime+Probe attacks. SP2021 and CEASER-S are vulnerable. On SeqAss, the number of usable congruent address never rises over 80, but tens of thousands of congruent addresses are required by Evict+Time, as shown by Figure 11. SeqAss is safe with a wide margin.

5.4. Performance Overhead

Let us start with the area overhead. Compared with the baseline (a non-randomized set-associative cache), the storage overhead of SeqAss comes from increased address tag bits (needed by all randomized caches), a cipher id, the cache set predictor and the conflict counter (Section 4.1), the evict buffer and the attack counter (Section 4.4). Table 8 presents the detailed storage overhead of all cache structures listed in Table 3. Like all randomized caches, the address tag in SeqAss is widened from 28 to 42 bits, assuming a 48-bit address system. One bit is added to each cache block to record the cipher used for its mapping (cipher id). Each cache set is attached with a 3-bit conflict counter and a 5-bit attack counter, leading to 8 extra bits per cache set. The storage overhead of the cache set predictor and the evict buffer is counted separately and added into the total bits and area results.

The ASIC area is estimated using CACTI-6.5 [50] in a 32nm technology. The logical area of the ciphers, update and decay logic of the conflict counters, and remap control logic is negligible and not estimated according to the discussion in [9]. SeqAss incurs a small storage overhead of 3.04% extra bits and 3.72% extra area compared with baseline, and this overhead is significantly smaller than Mirage. It is 16.6% and 28.8% lower than Mirage-50 and Mirage-75, respectively.

TABLE 7. SUCCESS RATE OF FINDING ONE CONGRUENT ADDRESS FOR AN ARBITRARILY SELECTED TARGET ADDRESS USING VARIOUS SEARCH ALGORITHMS. REMAP IS DISABLED. RESULTS ARE AVERAGED FROM 1000 REPEATED RUNS FOR THE SAME TARGET ADDRESS AND REPRESENTED IN SUCCESS *rate* AND NUMBER OF LLC *evictions*.

	CT		CT-prefetch		CTPP		PPP	
Structure	rate	evictions	rate	evictions	rate	evictions	rate	evictions
Baseline	**100%**	**264K**	**100%**	**16.5K**	**100%**	**549K**	**8.7%**	**443K**
SP2021	**100%**	**263K**	**100%**	**16.8K**	**100%**	**531K**	**7.9%**	**443K**
CEASER-S	**100%**	**259K**	**100%**	**30.5K**	0.4%	1.07M	0.4%	480K
Chameleon	0%	265K	0%	105K	0%	584K	0%	453K
Mirage	0%	253K	0%	247K	0%	609K	0%	469K
SeqAss	**86.1%**	**345K**	**85.7%**	**344K**	0%	604K	0%	455K

TABLE 8. COMPARISON OF LLC STORAGE OVERHEAD

Structure	meta bits per set	cache sets	total meta bits	meta area (mm^2)	bits per block	total data bits	data area (mm^2)	total bits	overhead in bits	total area (mm^2)	overhead in ASIC area
Baseline	16x(28+11)	16K	10.2M	2.267	512	134.2M	28.30	144.7M	—	30.63	—
SP2021	16x(42+12)	16K	14.2M	3.287	512	134.2M	28.30	148.9M	2.93%	31.74	3.62%
CEASER-S	8x(42+12)	32K	14.2M	3.287	512	134.2M	28.30	148.6M	2.73%	31.65	3.33%
Chameleon	8x(42+11)	32K	13.9M	3.216	512	134.2M	28.30	148.1M	2.37%	31.52	2.91%
Mirage-50	12x(42+29)	32K	27.9M	7.175	531	139.2M	29.67	167.1M	15.5%	36.85	20.3%
Mirage-75	14x(42+29)	32K	32.6M	10.92	531	139.2M	29.67	171.8M	18.7%	40.59	32.5%
SeqAss	16x(42+12)+8	16K	14.3M	3.323	512	134.2M	28.30	149.1M	3.04%	31.77	3.72%

TABLE 9. ACCESSES AND MISSES OF VARIOUS CACHE STRUCTURES

Structure	Acc. PKI	Miss PKI	Reloc. PKI	Remap PGI	Mem Acc. PKI
Baseline	4.785	0.559	0.000	0.000	0.953
SP2021	4.662	0.601	0.074	0.241	1.006
CEASER-S	4.700	0.545	0.025	0.078	0.924
Chameleon	4.810	0.565	1.131	0.000	0.968
Mirage-50	4.605	0.580	0.000	0.000	0.983
Mirage-75	4.667	0.586	0.000	0.000	0.999
SeqAss	4.607	0.495	0.078	0.008	0.845

TABLE 10. OVERHEAD REDUCTION BY EACH TECHNIQUE IN SEQASS

	LLC Misses PKI	Reloc. PKI	Remap PGI	Mem Accesses PKI
Baseline	0.559	0	0	0.953
+LdBal	0.535 (-4.29%)	0	0	0.911 (-4.41%)
+Reloc.	0.520 (-6.98%)	0.08	0	0.897 (-5.88%)
+EvBuf	0.513 (-8.23%)	0.079	0	0.876 (-8.08%)
+Remap	0.495 (-11.4%)	0.078	0.008	0.845 (-11.3%)

We evaluate the runtime and power performance by running the SPECrate CPU 2017 benchmark suite [26] on the Spike processor simulator [43], where Spike's cache model is replaced with our implementations of the various cache structures listed in Table 3. For each benchmark case, detailed cache performance data and simulation traces are recorded for executing 10G instructions. CACTI 6.5 is used to estimate the static power and access energy of each SRAM block. Consequently the dynamic energy of a single LLC access is approximated by accumulating the energy of all SRAM blocks visited by the LLC access. The total dynamic power is calculated as the total dynamic energy divided by the estimated running time using the IPC specified in Table 3. The miss rate and power performance of all benchmark cases is depicted in Figure 15. The overall cache miss rate is summarized in Table 9.

According to the results in Figure 15a and 15b, SeqAss incurs the least LLC MPKI and memory access overhead in all randomized cache structures for most benchmark cases. To be specific, a total of 11 out of the 23 benchmark cases benefit from reduced cache misses, including several memory heavy cases, i.e., 502.gcc, 505.mcf, 507.cactuBSSN, and 520.omnetpp. Overall, SeqAss achieves a significant reduction of 11.4% and 11.3% on LLC MPKI and memory accesses, respectively, compared with baseline, while most other randomized caches incur extra cache misses and memory accesses. Mirage-75's LLC MPKI and memory accesses are 18.4% and 18.2% higher than SeqAss, respectively. To demonstrate the performance benefits of the individual techniques introduced in SeqAss, we gradually enable them and test for performance. According to the results in table 10, each of the techniques is able to reduce cache misses.

Combining the energy estimation from CACTI 6.5 and the detailed performance data from simulation, Figure 15e and Table 11 reveal the power overhead. All randomized cache structures introduce power overhead. The power overhead incurred by Mirage is obviously unacceptable. The 3.43% power overhead of SeqAss is in line with SP2021, CEASER-S and Chameleon, and it is 14.4% and 22.1% lower than Mirage-50 and Mirage-70, respectively.

We would also like to raise the issue of dynamic power overhead, which was overlooked by previous work. On average, the total power consumption is dominated by static power. Dynamic power accounts for only 2∼3% of the total power. However, this ratio is related to the amount of memory access issued by applications. For memory heavy applications, such as 519.lbm, the ratio of dynamic

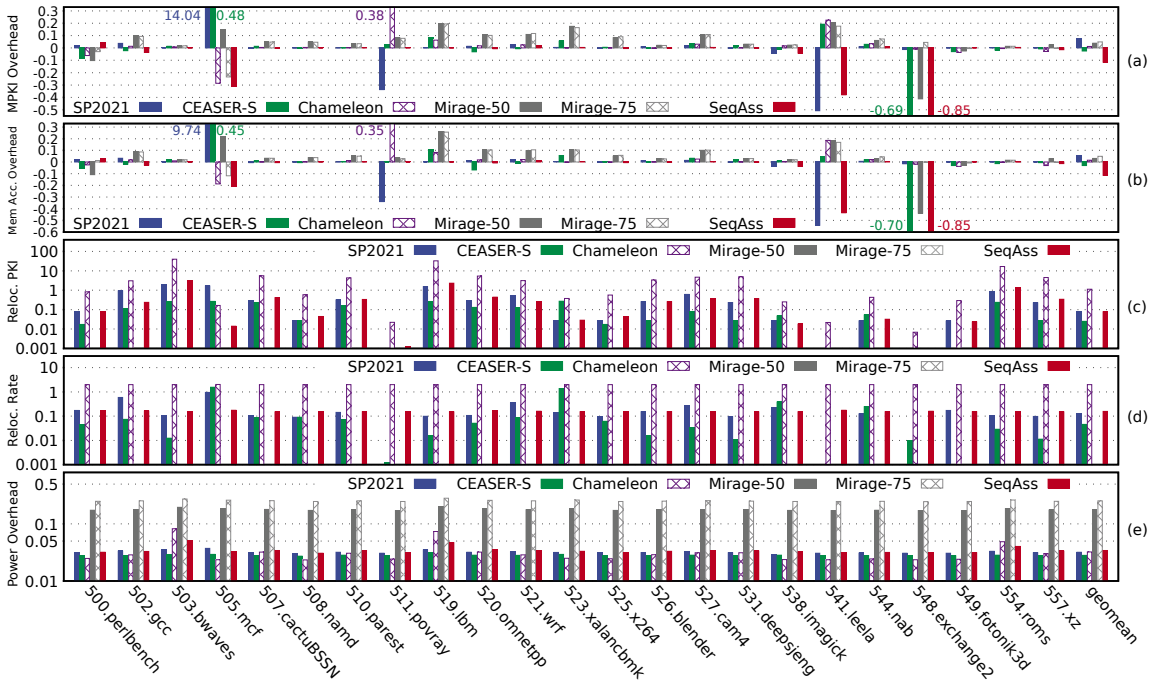

Figure 15. The LLC MPKI overhead (a), memory access overhead (b), relocation PKI (c) and power overhead (d) of running the SPECrate CPU 2017 benchmark suite on various cache structures. Each benchmark case is executed on the Spike processor simulator for 10G instructions. The y-axis of relocation PKI (c) and power overhead (d) is scaled in Log10.

TABLE 11. COMPARISON OF LLC POWER (OVERHEAD) (UNIT: W)

Structure	leak	dyn.	total	dyn. of 519.lbm
Baseline	5.93	0.120	6.047	0.440
SP2021	6.11	0.132	6.240 (3.19%)	0.480 (9.20%)
CEASER-S	6.09	0.128	6.218 (2.82%)	0.475 (7.91%)
Chameleon	6.07	0.176	6.242 (3.22%)	0.769 (75.0%)
Mirage-50	6.94	0.181	7.126 (17.8%)	0.712 (61.2%)
Mirage-75	7.39	0.198	7.588 (25.5%)	0.771 (75.2%)
SeqAss	6.11	0.144	6.255 (3.43%)	0.555 (26.1%)

power rises to 7% for baseline, and 11.6% for Chameleon. Chameleon is particularly worse than others due to its high relocation rate. Relocation is an energy consuming operation. Each relocation incurs two accesses to the cache array to swap both metadata and data, consuming an extra of ∼60% energy beyond the energy incurred by a cache miss (see Table 13 in Appendix B). Reducing the amount of relocation is important for reducing dynamic power overhead. Chameleon incurs a large number of relocations. After the LLC is fully warmed up, each cache miss unavoidably triggers a cache eviction, during which the conflicted cache block is moved to the VC. This block is later relocated back to the main cache array and one block in the main cache array is swapped back (relocated to) the VC. Therefore, each cache miss incurs two relocations. As shown in Figure 15c and 15d, Chameleon incurs the highest number of relocations and its relocation rate is fixed at 2. The number of relocations incurred by SeqAss is significantly lower than Chameleon. SeqAss has a negligible remap rate of only 0.008 remap per 1G instructions (PGI). Most of the relocations are incurred by partial relocation. Complying with the estimation in Section 4.2, only ∼15.6% of evictions trigger relocations even when on-demand remap is enabled. Thanks to the pattern-ignorant decay process, the relocation rate is almost fixed with a small standard deviation of 0.77%. according to the results showing in Figure 15d.

To demonstrate the impact of reduced relocation on dynamic power overhead, the last column of table 11 reveals the dynamic power needed to run 519.lbm, which is much higher on randomized caches than on baseline. SP2021 and CEASER-S are relatively power efficient as they consume only 8∼9% more than baseline. This overhead shoots up to 61.2∼75.2% for Mirage and 75.0% for Chameleon. Relocation is the root cause for Chameleon's extremely high power overhead. Mirage still suffers from its large storage overhead as the per-access energy increases proportionally with the array size. Since SeqAss incurs much less relocations than Chameleon, it consumes 26.1% more dynamic power than baseline. Although this is higher than SP2021 and CEASER-S, it is 49.1% and 48.9% lower than Mirage and Chameleon, respectively. The dynamic power reduction against Mirage and Chameleon is significant.

6. Conclusion

A new randomized cache structure based on the sequential associativity, namely the SeqAss cache, has been proposed. SeqAss achieves a defense as strong as Mirage, as they are the only two cache structures reasonably safe

from Evict+Time attacks and thwart Prime+Probe attacks. Rather than relying on techniques intrusive to the traditional cache structure, such as cache skews, over-provided metadata space, and separated data storage, SeqAss retains the set-associative structure and the LRU replacement policy. Instead of raising cache miss rate, SeqAss actually reduces it by 11.4%. Its area and power overhead is 28.8% and 22.1%, respectively, which is significantly lower than Mirage. When running memory heavy applications, it incurs ∼50% lower dynamic power overhead compared to Mirage and Chameleon.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China under grant No. 62172406 and the CAS Pioneer Hundred Talents Program. Any opinions, findings, conclusions, and recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] M. Yan, R. Sprabery, B. Gopireddy, C. W. Fletcher, R. H. Campbell, and J. Torrellas, "Attack directories, not caches: Side-channel attacks in a non-inclusive world," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, May 2019, pp. 888–904.

[2] J. Kim, S. van Schaik, D. Genkin, and Y. Yarom, "iLeakage: Browser-based timerless speculative execution attacks on Apple devices," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, Nov. 2023, pp. 2038–2052.

[3] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 775–787.

[4] ——, "New attacks and defense for encrypted-address cache," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM, Jun. 2019, pp. 360–371.

[5] X. Zhang, H. Gong, R. Chang, and Y. Zhou, "RECAST: Mitigating conflict-based cache attacks through fine-grained dynamic mapping," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 3758–3771, 2024.

[6] 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 USENIX Security Symposium (Security)*. USENIX Association, Aug. 2019, pp. 675–692.

[7] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "PhantomCache: Obfuscating cache conflicts with localized randomization," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, Feb. 2021.

[8] T. Unterluggauer, A. Harris, S. Constable, F. Liu, and C. Rozas, "Chameleon cache: Approximating fully associative caches with random replacement to prevent contention-based cache attacks," in *Proceedings of the IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, Sep. 2022, pp. 13–24.

[9] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *Proceedings of the USENIX Security Symposium (Security)*. USENIX Association, Aug. 2021, pp. 1379–1396.

[10] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, May 2019, pp. 39–54.

[11] A. Purnal and I. Verbauwhede, "Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache," *arXiv*, vol. 1908.03383, Aug. 2019, https://arxiv.org/abs/1908.03383.

[12] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, May 2021, pp. 987–1002.

[13] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, Nov. 2021, pp. 2906–2920.

[14] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "CaSA: End-to-end quantitative security analysis of randomly mapped caches," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2020, pp. 1110–1123.

[15] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! But we can fix it," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, May 2021, pp. 955–969.

[16] W. Song, Z. Xue, J. Han, Z. Li, and P. Liu, "Randomizing set-associative caches against conflict-based cache side-channel attacks," *IEEE Transactions on Computers*, vol. 73, no. 4, pp. 1019–1033, Apr. 2024.

[17] A. Bhatla, Navneet, and B. Panda, "The Maya cache: A storage-efficient and secure fully-associative last-level cache," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Jun. 2024, pp. 32–44.

[18] B. Calder, D. Grunwald, and J. S. Emer, "Predictive sequential associative cache," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Feb. 1996, pp. 244–253.

[19] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA)*. Springer, 2006, pp. 1–20.

[20] W. Song and P. Liu, "Dynamically finding minimal eviction sets can be quicker than you think for side-channel attacks against the LLC," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. USENIX Association, Sep. 2019, pp. 427–442.

[21] Z. Xue, J. Han, and W. Song, "CTPP: A fast and stealth algorithm for searching eviction sets on Intel processors," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. ACM, Oct. 2023, pp. 151–163.

[22] T. Kessous and N. Gilboa, "Prune+PlumTree — Finding eviction sets at scale," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, May 2024, pp. 3754–3772.

[23] J. P. Thoma and T. Güneysu, "Write me and I'll tell you secrets — Write-after-write effects on Intel CPUs," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. ACM, Oct. 2022.

[24] A. Seznec, "A case for two-way skewed-associative caches," in *Proceedings of the Annual International Computer Architecture Symposium*, May 1993, pp. 169–178.

[25] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. S. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM, Jun. 2010, pp. 60–71.

[26] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017 — Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, Apr. 2018, pp. 41–42.

[27] Y. Guo, X. Xin, Y. Zhang, and J. Yang, "Leaky way: A conflict-based cache covert channel bypassing set associativity," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Oct. 2022.

[28] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Transactions on Computer Systems*, vol. 6, no. 4, pp. 393–431, Nov. 1988.

[29] A. Agarwal and S. D. Pudar, "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, May 1993, pp. 179–190.

[30] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of AMD's cache way predictors," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, Oct. 2020, pp. 813–825.

[31] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the USENIX Security Symposium (Security)*. USENIX Association, Aug. 2014, pp. 719–732.

[32] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer, S. Mangard, and D. Gruss, "Scatter and split securely: Defeating cache contention and occupancy attacks," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, May 2023, pp. 2273–2287.

[33] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, "SEV-Step a single-stepping framework for AMD-SEV," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 1, p. 180–206, Dec. 2023.

[34] M. Li, K. Bu, C. Miao, and K. Ren, "TreasureCache: Hiding cache evictions against side-channel attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 5, pp. 4574–4588, Sep. 2024.

[35] D. Ojha and S. Dwarkadas, "RollingCache: Using runtime behavior to defend against cache side channel attacks," *arXiv*, vol. 2408.08795, Aug. 2024, https://arxiv.org/abs/2408.08795.

[36] F. Canale, T. Güneysu, G. Leander, J. P. Thoma, Y. Todo, and R. Ueno, "SCARF — A low-latency block cipher for secure cache-randomization," in *Preedings of the USENIX Security Symposium (Security)*, Aug. 2023, pp. 1937–1954.

[37] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, 1989.

[38] A. Harris, S. Wei, P. Sahu, P. Kumar, T. M. Austin, and M. Tiwari, "Cyclone: Detecting contention-based cache information leaks through cyclic interference," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, Oct. 2019, pp. 57–72.

[39] H. Fang, M. Doroslovački, and G. Venkataramani, "Reuse-trap: Repurposing cache reuse distance to defend against side channel leakage," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. IEEE, Jul. 2020, p. 6.

[40] K. Wang, F. Yuan, R. Hou, Z. Ji, and D. Meng, "Capturing and obscuring ping-pong patterns to mitigate continuous attacks," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Mar. 2020, pp. 1408–1413.

[41] F. Yuan, K. Wang, J. Ying, R. Hou, L. Zhao, P. Li, Y. Zhu, Z. Ji, and D. Meng, "Architecting the autocuckoo filter to defend against cross-core cache attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 4, pp. 1280–1294, 2023.

[42] J. Han, Z. Wang, H. Ma, and W. Song, "Spike-FlexiCAS: RISC-V processor simulator supporting flxible cache architecture configuration," *International Journal of Software and Informatics*, vol. 15, no. 3, pp. 329–348, Sep. 2025.

TABLE 12. RANDOM SELECTED BENCHMARK CASES

	cases
mcore-0	500.perlbench-0, 500.perlbench-1, 520.omnetpp, 554.roms
mcore-1	502.gcc-1, 502.gcc-3, 502.gcc-4, 503.bwaves-0
mcore-2	503.bwaves-1, 505.mcf, 538.imagick, 541.leela
mcore-3	503.bwaves-2, 507.cactuBSSN, 525.x264-0, 557.xz-2
mcore-4	502.gcc-2, 508.namd, 544.nab, 548.exchange2
mcore-5	521.wrf, 526.blender, 549.fotonik3d, 557.xz-1
mcore-6	502.gcc-4, 520.omnetpp, 527.cam4, 557.xz-0
mcore-7	500.perlbench-0, 500.perlbench-1, 500.perlbench-2, 503.bwaves-2
mcore-8	502.gcc-0, 503.bwaves-0, 503.bwaves-3, 531.deepsjeng
mcore-9	502.gcc-1, 503.bwaves-1, 521.wrf, 523.xalancbmk
mcore-10	502.gcc-3, 510.parest, 557.xz-0, 557.xz-2
mcore-11	505.mcf, 519.lbm, 527.cam4, 538.imagick
mcore-12	500.perlbench-1, 503.bwaves-2, 525.x264-1, 554.roms
mcore-13	525.x264-2, 526.blender, 541.leela, 544.nab
mcore-14	500.perlbench-0, 520.omnetpp, 548.exchange2, 557.xz-2
mcore-15	502.gcc-0, 525.x264-1, 554.roms, 557.xz-0
mcore-16	500.perlbench-1, 502.gcc-3, 526.blender, 538.imagick
mcore-17	503.bwaves-3, 505.mcf, 525.x264-2, 557.xz-1
mcore-18	502.gcc-4, 510.parest, 541.leela, 549.fotonik3d
mcore-19	519.lbm, 523.xalancbmk, 525.x264-0, 531.deepsjeng

[43] A. Waterman, T. Newsome, C.-M. Chao, and others, "Spike RISC-V ISA simulator," 2021, https://github.com/riscv/riscv-isa-sim.

[44] C. Celio, A. Waterman, P. Dabbelt, and others, "Speckle," 2024, https://github.com/comparch-security/speckle-2017.

[45] P. Dabbelt, K. Cheng, C. Müllner, and others, "RISC-V GNU compiler toolchain," 2023, https://github.com/riscv-collab/riscv-gnu-toolchain/releases/tag/2023.12.12.

[46] J. Xu, Y. Zhou, and others, "RISC-V GNU compiler toolchain," 2025, https://github.com/comparch-security/spike-sdk-spec2017.

[47] N. Chinchor, "MUC-4 evaluation metrics," in *Proceedings of the Conference on Message Understanding*. Association for Computational Linguistics, Jun. 1992, pp. 22–29.

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

[49] T. Verma, A. Anastasopoulos, and T. Austin, "These aren't the caches you're looking for: Stochastic channels on randomized caches," in *Proceedings of the IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, Sep. 2022, pp. 37–48.

[50] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2007, pp. 3–14.

Appendix A.
Performance for Running Parallel Applications

This section evaluates the performance of different randomized LLCs when multiple applications are running simultaneously on parallel cores. We configure the Spike simulator to a 4-core processor and execute 20 combinations of 4 randomly selected SPECrate CPU 2017 benchmark cases. These combinations are detailed in Table 12. There are 36 test inputs belonging to a total of 23 benchmark cases.

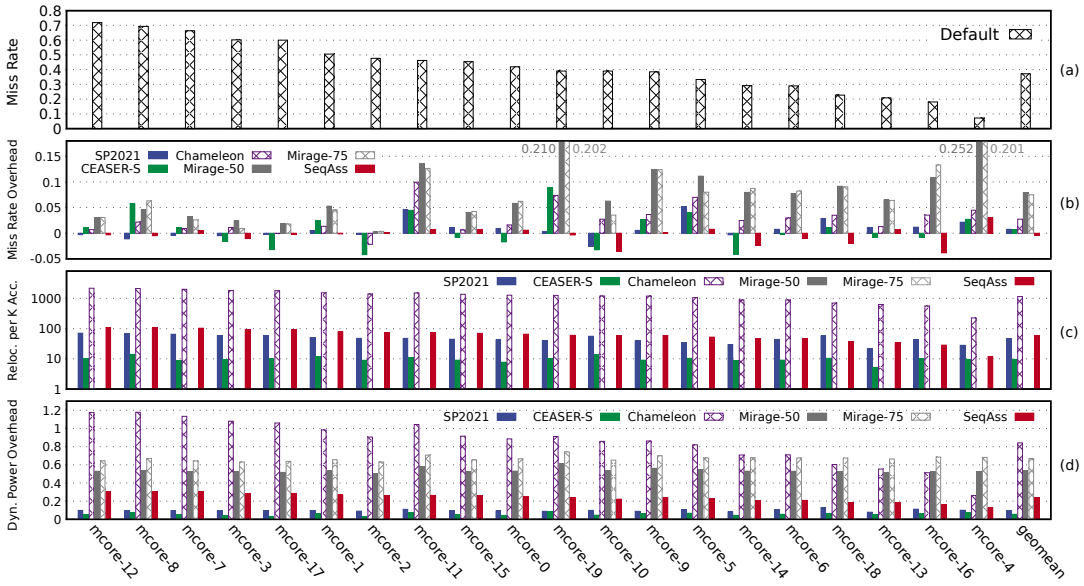

Figure 16. The LLC miss rate (a), LLC miss rate overhead (b), relocation per K LLC accesses (c) and dynamic power overhead (d) of running four randomly selected SPECrate CPU 2017 benchmark cases in parallel. Each combination of benchmark cases is executed on the Spike processor simulator for 200M LLC accesses. Combinations are ordered by their LLC miss rates on the default LLC (traditional non-randomized set-associative cache). The y-axis of relocation per K LLC accesses (c) is scaled in Log10.

The random selection is uniformly distributed to ensure each test input is included in at least two test combinations. During the execution, the cache model records performance counter values from the LLC and reports an accumulated result after executing 200M LLC accesses. According to Table 9, the average LLC accesses per K instructions for single-core is around 4.8; therefore, 200M LLC accesses is roughly equivalent to running 10.4 G instructions for each benchmark case in the 4-core parallel run.

Using the same evaluation method described in Section 5.4, the performance result is presented in Figure 16. The combinations are ordered according to their LLC miss rates shown in Figure 16a. The randomly selected combinations demonstrate significantly different LLC performance as the miss rate varies from 7.22% (combination *mcore-4*) to just 71.9% (combination *mcore-12*).

Figure 16b demonstrates the LLC miss rate overhead against the default (the traditional non-randomized set-associative) LLC. Among all randomized cache structures, SeqAss is the only one that achieves a miss rate reduction on average, although this reduction is only 0.42%, which is much less than the 11.4% achieved in single-core. The mix of multiple applications increases the access pressure for the LLC and reduces the available locality exploitable by the LRU replacement policy and the increased cache associativity. Nevertheless, SeqAss still achieves a miss rate reduction for a total 11 out of the 20 combinations. All of the randomized caches providing strong defenses, i.e., Chameleon and Mirage, suffer from much higher LLC miss rate than SeqAss. The average miss rate is increased by 2.73%, 7.93%, and 7.49% for Chameleon, Mirage-50 and Mirage-75, respectively.

Relocation is an important contributor for the extra dynamic power introduced by randomized caches. Figure 16c reveals the number of relocations triggered by serving K LLC accesses on all randomized cache structures. The result is comparable to that shown in Figure 15c. SeqAss incurs around the same number of relocations with SP2021, which is higher than CEASER-S but significantly lower than Chameleon. The result indicates that Chameleon would suffer from high consumption of dynamic power, which is verified in Figure 16d. Chameleon consumes the highest dynamic power, which is 84.2% higher than default. Mirage also suffers from high dynamic power due to its high storage overhead. Mirage-50 and Mirage-75 incur 53.4% and 66.8% overhead in dynamic power, respectively. The 23.7% overhead asserted by SeqAss is significantly lower than Chameleon and Mirage. This result is also comparable to that presented in Table 11.

In summary, SeqAss demonstrates observable performance benefit when multiple applications are running simultaneously on parallel cores. It is the only randomized cache structure managed to reduce the average LLC miss rate. All of the other randomized cache structures, especially the ones providing strong defenses, i.e. Chameleon and Mirage, suffer from increased LLC miss rate. The extra dynamic power incurred by SeqAss is moderate, while Chameleon and Mirage suffer from significantly higher power overhead.

Appendix B. Estimation of Dynamic Energy per Access

To detail the dynamic energy consumption of various randomized caches, we have set up a simplified model

TABLE 13. ENERGY PER ACCESS (nJ)

	hit	miss	relocation
Baseline	1.406	3.268	N/A
SP2021	1.485	3.374	2.481
CEASER-S	1.485	3.367	2.481
Chameleon	1.475	3.958	1.242
Mirage-50	2.083	4.933	2.749
Mirage-75	2.334	5.278	N/A
SeqAss	1.518	4.020	2.482

for estimating the dynamic energy. The dynamic power of running a SPEC CPU 2017 benchmark case is estimated as:

$$P_{\text{dyn}} = \frac{N_{\text{hit}} \cdot E_{\text{hit}} + N_{\text{miss}} \cdot E_{\text{miss}} + N_{\text{reloc}} \cdot E_{\text{reloc}}}{N_{\text{inst.}}/(f \cdot \mathit{IPC})}$$

where N_{hit}, N_{miss}, and N_{reloc} are the total number of LLC hits, misses, and relocations, respectively, and E_{hit}, E_{miss}, and E_{reloc} are the energy consumed by a single LLC hit, miss, and relocation, respectively. N_{hit}, N_{miss}, and N_{reloc} are provided by the simulation traces. The total running time is calculated as $N_{\text{inst.}}/(f \cdot \mathit{IPC})$: the total number of executed instructions $N_{\text{inst.}}$ divided by the number of instructions executed per second ($f \cdot \mathit{IPC}$), where f and IPC are provided in Table 3. To obtain the estimate of P_{dyn}, we need to model E_{hit}, E_{miss}, and E_{reloc}. Since most of the energy per cache access is consumed by accessing SRAMs, we estimate the total energy by accumulating the energy incurred by SRAMs visited by each cache access:

$$E_{\text{hit}} = k_h \cdot E_{\text{meta,set}} + E_{\text{data}} + E_{\text{repl.}} \quad (1)$$

$$E_{\text{miss}} = E_{\text{hit}} + 2E_{\text{meta,way}} + k_d \cdot E_{\text{data}} + E_{\text{repl.}} + E_{\text{other}} \quad (2)$$

$$E_{\text{reloc}} = 2(E_{\text{meta,way}} + E_{\text{data}}) \quad (3)$$

For each LLC hit, a set of metadata ways are read in parallel ($E_{\text{meta,set}}$), the data is fetched from the data array (E_{data}) and the replacement state is updated ($E_{\text{repl.}}$). For skewed caches, two metadata sets from both skews are read in parallel ($k_h = 2$). SeqAss needs extra $\sim$10% metadata access due to prediction errors ($k_h = 1.1$). Besides all the energy consumption equivalent to a hit, each LLC miss needs to acces the metadata block twice ($2E_{\text{meta,way}}$, one for writeback and one for memory fetch), potentially write back a dirty block, store the fetched block ($k_d = 1.5$ considering 50% probability of writing back), and update the replacement state after eviction ($E_{\text{repl.}}$). Some other operations may be required as well (E_{other}, i.e. detectors in SP2021 and SeqAss, victim cache/buffer in Chameleon and SeqAss, extra meta access in Mirage). When a relocation occurs, two cache blocks swap places by two meta and data accesses. Chameleon is special as its relocation happens between the cache array and the VC. Subsequently, its $E_{\text{reloc}} = E_{\text{meta,way}} + E_{\text{data}} + E_{\text{VC}}$. Based on this model, Table 13 shows the estimated energy per access in all cache structures.

Appendix C. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

The paper introduces a sequential associative cache, denoted the SeqAss cache, that offers protection against conflict-based attacks, including Prime+Probe and Evict+Time. SeqAss incorporates optimized load balance and cache relocation mechanisms to reduce power consumption and the storage overhead in the ASIC area.

C.2. Scientific Contributions

- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field
- Other

C.3. Reasons for Acceptance

1) The paper presents a new cache design that repurposes sequential associativity employed by the established set-associative LLCs for security with a load-balanced insertion extension, which addresses a long-known issue that structural disruption and performance overhead seem unavoidable in such a design.
2) The paper provides a valuable step forward in an established field. The proposed cache design provides defense against conflict-based cache attacks that is as effective as state-of-the-art solutions, but with better performance and lower area and power overhead.

C.4. Noteworthy Concerns

1) The paper only includes results of the SPEC benchmark suite but not other general application workloads.