New Attacks and Defense for Encrypted-Address Cache

Moinuddin K. Qureshi
Georgia Institute of Technology
moin@gatech.edu

ABSTRACT

Conflict-based cache attacks can allow an adversary to infer the access pattern of a co-running application by orchestrating evictions via cache conflicts. Such attacks can be mitigated by randomizing the location of the lines in the cache. Our recent proposal, CEASER, makes cache randomization practical by accessing the cache using an encrypted address and periodically changing the encryption key. CEASER was analyzed with the state-of-the-art algorithm on forming eviction sets, and the analysis showed that CEASER with a Remap-Rate of 1% is sufficient to tolerate years of attack.

In this paper, we present two new attacks that significantly push the state-of-the-art in forming eviction sets. Our first attack reduces the time required to form the eviction set from $O\left(L^2\right)$ to $O\left(L\right)$, where L is the number of lines in the attack. This attack is 35x faster than the best-known attack and requires that the Remap-Rate of CEASER be increased to 35%. Our second attack exploits the replacement policy (we analyze LRU, RRIP, and Random) to form eviction set quickly and requires that the Remap-Rate of CEASER be increased to more than 100%, incurring impractical overheads.

To improve the robustness of CEASER against these attacks in a practical manner, we propose *Skewed-CEASER (CEASER-S)*, which divides the cache ways into multiple partitions and maps the cache line to be resident in a different set in each partition. This design significantly improves the robustness of CEASER, as the attacker must form an eviction set that can dislodge the line from multiple possible locations. We show that CEASER-S can tolerate years of attacks while retaining a Remap-Rate of 1%. CEASER-S incurs negligible slowdown (within 1%) and a storage overhead of less than 100 bytes for the newly added structures.

ACM Reference Format:
Moinuddin K. Qureshi. 2019. New Attacks and Defense for Encrypted-Address Cache. In *ISCA '19: 46th International Symposium on Computer Architecture, June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307650.3322246

1 INTRODUCTION

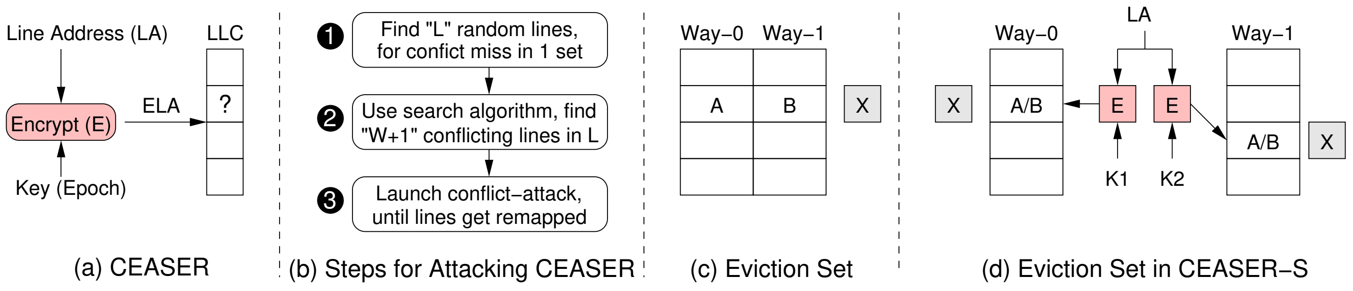
Conflict-based cache attacks are an important class of cache side-channels, where an adversary can carefully orchestrate cache evictions to learn the access pattern of a co-running application. Such attacks have been used to learn secrets, such as the encryption keys for AES [1]. Conflict-based attacks become feasible when the adversary and the victim share some storage structures. Modern processors typically share the last-level cache (LLC) between all the cores to efficiently utilize the cache space. Unfortunately, such sharing makes the LLC vulnerable to attacks, as an adversary can learn the access pattern of the victim by causing LLC evictions, even when the adversary and the victim are scheduled to be on different cores.

Architectural solutions to mitigate conflict-based cache attacks broadly fall in two categories. First, isolation-based mitigation [2–4], whereby the lines of the victim are preserved within the cache, making it harder for the adversary to dislodge the content of the victim. Unfortunately, dedicating portions of the LLC for each core can result in inefficient use of cache space. Second, randomization-based mitigation [2, 5], whereby the location of the line in the cache is determined randomly and this information is stored in a mapping table. To protect the mapping table from attacks, the OS is required to group the applications into protected and unprotected groups and only the protected applications are allowed access to the mapping table. Ideally, we want to avoid the storage overhead of large mapping tables and the OS support.

Our recent work, CEASER [6], enables LLC randomization in a practical manner by exploiting the insight that randomized mapping of memory lines to cache locations can be accomplished efficiently by accessing the cache with an *Encrypted Line Address (ELA)*. The avalanche effect of encryption causes conflicting lines to get scattered throughout the cache sets, as shown in Figure 1(a). Given enough time, an adversary can launch a timing-based attack to determine the group of lines that map to the same set. To avoid this, CEASER periodically changes the keys and performs dynamic remapping of the cache from the old key to the new key. The remapping rate of CEASER is controlled by a parameter called the *Remap-Rate (R)*. CEASER was analyzed with an attack kernel that represented the state-of-the-art in forming eviction sets [7], and this analysis showed that with a Remap-Rate of $R = 1\%$, CEASER can tolerate years of attack while incurring negligible slowdown.

The security analysis of CEASER is based on an attack pattern consisting of three steps, as shown in Figure 1(b). In Step-1, the attacker accesses the cache with L random lines with the aim of getting a conflict miss on one of the cache sets. When a conflict miss occurs, the attacker needs to identify which lines map to the conflicting set. In Step-2, the attacker applies a search algorithm to find the $\left(W+1\right)$ conflicting lines (for a W-way cache) from the L lines. Once the conflicting lines are found, the attacker launches the conflict-based attack (Step-3) until the lines in the conflicting set get remapped. After remapping, the three steps are repeated.

The objective of the search algorithm in Step-2 is to converge on the (W+1) conflicting lines from the L lines. This can be accomplished by holding out one line from the L lines and testing the remaining (L-1) lines for a conflict miss. If the conflict miss occurs, the holdout line does not map to the conflicting set, otherwise it



Line Address (LA)
LLC
Encrypt (E)
ELA
?
Key (Epoch)
(a) CEASER
1 Find "L" random lines, for confict miss in 1 set
2 Use search algorithm, find "W+1" conflicting lines in L
3 Launch conflict−attack, until lines get remapped
(b) Steps for Attacking CEASER
Way−0
Way−1
A
B
X
(c) Eviction Set
LA
Way−0
Way−1
X
A/B
E
E
A/B
X
K1
K2
(d) Eviction Set in CEASER−S

Figure 1: (a) Overview of CEASER, which uses encrypted address and remapping (b) Steps for attacking CEASER to find evictions set (c) Example of eviction set, (d) Our proposal, Skewed-CEASER (CEASER-S), which divides the cache ways into two partitions and allows the line to map to a different set in each partition (depending on the encryption key of that partition). Attacks on CEASER-S are much harder to orchestrate as the attacker needs to find lines that conflict with the target line "X" in both possible locations.

does. Such an algorithm has been used to learn the bank mapping of shared caches in Intel architectures [7] and currently represents the best-known method of forming eviction sets. We call this algorithm *Single Holdout Method (SHM)*. Note that SHM requires $O\left(L^2\right)$ accesses to determine the lines that map to the conflicting set, and these accesses must be completed within the remapping period of CEASER. Therefore, SHM limits the attacker to using small L. The probability that there will be a conflict miss when only a few random lines are present in the attack pattern is vanishingly small – this is the main reason why CEASER with a Remap-Rate of 1% is able to tolerate years of attacks. The security of CEASER is dictated by how quickly the adversary is able to form the eviction set. In this paper, we present two new attacks that significantly increase the state-of-the-art in forming eviction sets.

Our first attack is based on developing a faster search algorithm. The key insight in our attack is that the attack pattern typically has several hundred lines, however, only a few lines (17 lines for a 16-way cache) map to the conflicting set. Rather than testing one line at a time, our method splits the L lines into G groups (each containing L/G lines). Then it holds out one group and tests the remaining (G-1) groups for a conflict miss – if the conflict miss happens, the entire holdout group is removed from L. We call this search algorithm *Group Elimination Method (GEM)*. For a 16-way cache, GEM needs only $37 \cdot L$ accesses, which is 35x faster than SHM for a 1MB cache. To tolerate GEM, the Remap-Rate of CEASER must be increased from 1% to 35%, which would incur significant overhead.

Our second attack is based on exploiting the replacement policy. Both SHM and GEM are agnostic of replacement policy, and simply rely on the fact that when (W+1) lines of the attack pattern are mapped to the same set of a W-way cache, and these lines are accessed again, there will be at least one conflict miss. Nonetheless, the attacker can use the replacement policy to avoid the search algorithm altogether. For example, if (W+1) lines are installed on a W-way set and then accessed again, then LRU replacement will cause misses for all the (W+1) lines mapping to the conflicting set, and these lines can be identified without any search algorithm (time reduced to $2 \cdot L$ instead of $37 \cdot L$). Similarly, for Reuse-based policies (such as RRIP [8]), we can access each line twice before testing to obtain a vulnerability similar to LRU (time reduced to $3 \cdot L$ instead of $37 \cdot L$). If LRU/RRIP is used as the replacement policy for CEASER, the Remap-Rate of CEASER must be increased from 1% to more than 100% to tolerate these attacks, causing unacceptable overheads.

Ideally, we want to make CEASER robust while retaining a low Remap-Rate. Our solution is inspired from the prior work on skewed-associative caches [9–12], which allows the line to map to a different set in each way. We propose *Skewed-CEASER (CEASER-S)*, which divides the cache ways into multiple partitions, and uses a different set of encryption keys for each partition. Therefore, a given line maps to a different set in each partition, as shown in Figure 1(d). An access to CEASER-S probes both the partitions concurrently to determine the cache hit. On a miss, CEASER-S chooses one of partitions randomly and the replacement policy of that partition determines the install location of the incoming line.

Figure 1(c) and (d) compares the eviction set of CEASER and CEASER-S with two partitions. To evict Line X from the two-way CEASER, we need two lines (A and B) such that they map to the same set as X. The probability that two random lines A and B map to the same set as X is $1/S^2$ (S is the number of sets). With CEASER-S, to dislodge a line X, we need two lines A and B such that they can dislodge X from both possible locations. The probability that A and B map to the same set in Way-0 and Way-1 as line X is $1/S^4$.

We develop a model to analyze the time to attack CEASER-S and show that even with a Remap-Rate of only 1%, CEASER-S can tolerate years of attacks (even under idealized search algorithms).

Our paper introduces new attacks that significantly increase the state-of-the art in forming eviction sets, and solves the vulnerability introduced by these new attacks using a simple and practical design. Overall, our paper makes the following contributions:

(1) We present a new attack that significantly improves the state-of-the-art in forming eviction sets by reducing the search time for converging on the conflict list from $O\left(L^2\right)$ to $O\left(L\right)$. This attack is 35x faster than the currently best known attack.
(2) We present a new attack that leverages the replacement policy and avoids the search algorithm altogether. We show that popularly used replacement policies, such as LRU, RRIP, and Random,are all vulnerable. This attack would force the Remap-Rate of CEASER to be more than 100%.
(3) We propose Skewed-CEASER, which combines skewed caching and CEASER to significantly improve the robustness. CEASER-S can tolerate years of attack with Remap-Rate of 1%.

Our evaluation with 68 workloads shows that CEASER-S incurs negligible slowdown (within 1%). The newly added structures of CEASER-S incur a storage overhead of less than 100 bytes.

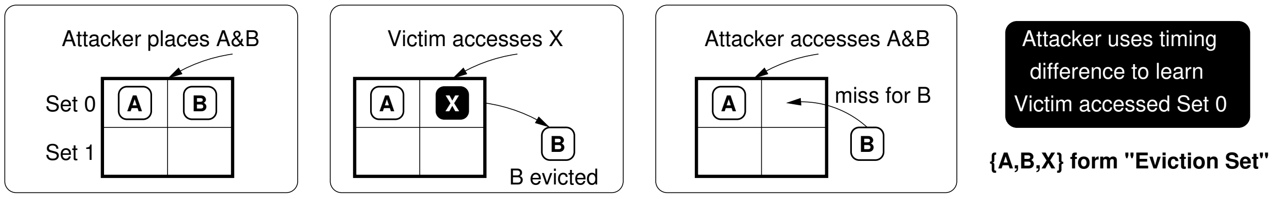


Figure 2: Example of "Prime+Probe" attack. The attacker uses lines A and B to infer that the victim accessed Set 0.

2 MOTIVATION AND BACKGROUND

Our paper analyzes the robustness of randomized caching at mitigating conflict-based attacks. In this section, we provide the background of the conflict-based attacks (Flush-based attacks are discussed in Section 8.3) and relevant schemes.

2.1 Conflict-Based Cache Attacks

In conflict-based attacks, the attacker tries to determine the cache sets have been accessed by a victim program. For example, in the Prime+Probe attack [13, 14], the attacker fills a cache set with its own lines (Prime step), waits for the victim to perform its accesses (Wait step), and then accesses the set again to determine which cache sets have been accessed by the victim (Probe step). Figure 2 shows an example of such a Prime+Probe attack on a two-way cache. The attacker places lines A and B in Set 0, and waits. The victim accesses a line (say line X) that maps to Set 0, which evicts line B. At a later time, the attacker accesses A and B, and measure the time. Given the long latency required for B, the attacker infers that the victim accessed Set 0. Knowing the access pattern of an application can be used to reveal secret information [1].

2.2 Table-Based Randomization

Prior approaches for mitigating conflict-based attacks rely on either preserving the victim lines, or on randomized mapping of the victim lines to cache locations. Examples of preservation-based approaches include PL-Cache [2] (lock lines of sensitive application in the cache), Non-Monopolizable Cache [3] and DAWG [4] (reserve a few ways of the shared cache for each core). Such approaches can result in inefficient use of cache space, as cache may get reserved for the application regardless of the reuse characteristics of the lines.

Randomization-based techniques rely on randomized mapping of the memory line to the cache set, thereby making it harder for the adversary to form an eviction set. Prior proposals [2, 5] used mapping tables to track the location of the line (or the set) in the cache. The disadvantage of such *Table-Based Randomization* schemes is that the mapping tables must scale linearly with the number of lines in the cache. While such tables can be implemented efficiently for small L1 caches [5], they become impractically large for a multi-megabyte LLCs. Furthermore, the effectiveness of these schemes relies on the OS-based classification of applications into protected and unprotected groups (otherwise, the mapping table can be attacked). To defend an 8 MB LLC, the overhead of the mapping table would be 1.25 MB (with OS support) and 8.5 MB (without OS support). Ideally, we want to avoid the significant storage overhead of large mapping tables and the reliance on the OS support.

2.3 CEASER: Algorithmic Randomization

Our recent work, CEASER [6], enables randomized LLC in a practical manner by exploiting the insight that randomized mapping of memory lines to cache locations can be accomplished efficiently by accessing the cache with an *Encrypted Line Address (ELA)*, as shown in Figure 3. The avalanche effect of encryption causes conflicting lines to get scattered throughout the cache sets. The ELA is visible only within the LLC, and the operations of rest of the memory system (such as coherence, prefetch, writeback) remain unchanged.

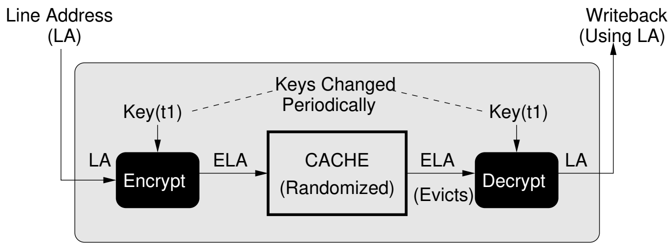


Figure 3: Overview of CEASER

Given enough time, an adversary can launch a timing-based attack to determine the group of lines mapped to the same set. To avoid this, CEASER periodically changes the keys and performs dynamic remapping of the cache from the current key to the next key.[1] The remapping rate of CEASER is controlled by the parameter *Remap Rate (R)*. CEASER was analyzed with the currently best-known algorithm for forming eviction sets [7] and the analysis showed that even with a Remap-Rate of $R = 1\%$ (remapping one 16-way set every 1600 cache accesses), CEASER can tolerate years of attack while incurring negligible performance overheads (approximately 1%).

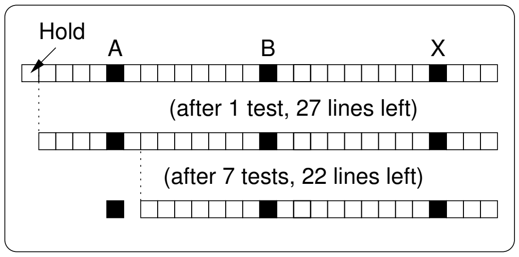
2.4 Goal of this Paper

In this work, we analyze the vulnerability of encrypted-address caches, such as CEASER, to attacks that can rapidly form eviction sets and analyze the impact this can have on the Remap-Rate. Specifically, we try to answer the following questions:

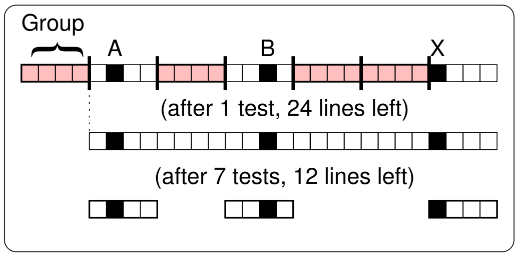
(1) Can we develop attacks that can form eviction sets at a rate faster than the current state-of-the-art algorithm [7]?
(2) What is the impact of cache replacement policy on the ability to form eviction sets quickly?
(3) How can we defend against faster attacks while still retaining a negligibly small remapping overhead?

We first describe our two new attacks to rapidly form eviction sets, and then a practical solution to defend CEASER against such attacks.

[1]The gradual remapping of CEASER is done at set-granularity, with one 16-way set getting remapped every 1600 accesses to the cache. For details, please refer [6] (Sec-IV).



(a) Single−Holdout Method (SHM)



(b) Group−Elimination Method (GEM)

Figure 4: SHM and GEM search for an attack pattern containing L=28 lines with A, B, and X mapping to the conflicting set.

3 ATTACK-1: FAST SEARCH ALGORITHM

The security analysis of CEASER is based on a pattern in which the cache is accessed with L random lines with the aim of getting a conflict miss on one of the sets. This is a trial and error step and is done until a conflict miss occurs. Even when a conflict miss occurs, the attacker still needs to identify which lines are mapped to the conflicting set. This is done using a search algorithm that takes the L lines as an input and returns the $(W+1)$ conflicting lines for a W-way cache. We first describe the search algorithm that was used to analyze CEASER. We then describe a new search algorithm that is substantially faster at forming the eviction set and reduces the complexity of search from $O(L^2)$ to $O(L)$, where L is the number of lines in the attack pattern. We then discuss impact on the Remap-Rate of CEASER to tolerate these faster attacks.

3.1 Current Algorithm: Single-Holdout Method

To determine the "W+1" conflicting lines in the "L" attack lines, the analysis in CEASER employed the state-of-the-art search algorithm that was used by Liu et al. [7] to learn the bank indexing function of Intel cache designs. We call this algorithm *Single Holdout Method (SHM)*. The search process of SHM is based on holding out one line, and testing the remaining lines for a conflict. If the conflict happens, the holdout line does not map to the conflicting set and can be removed from the list. This process is repeated until the list has only "W+1" lines left, as shown in Algorithm 1.

Algorithm 1: Single-Holdout Method (SHM)

```
Input: L random lines that cause conflicts in 1 set
Output: "W+1" lines that map to the conflicting set

While (L has more than "W+1" lines):
    Hold out one line from L
    Test remaining (L-1) lines for a conflict
    If conflict occurs, remove holdout line from L

Return L
```

Note that SHM requires $O(L^2)$ accesses to determine the lines that map to the conflicting set and these accesses must be completed within the remapping period of CEASER ($100 \cdot N$, where N is the number of lines in the cache and CEASER uses a Remap-Rate of 1%), otherwise the conflicting lines will get remapped to other sets and the attacker would be unable to form an eviction set. Therefore, SHM limits the attacker to attacking at most $L = \sqrt{100 \cdot N}$ lines (1280 lines for a 1MB cache bank containing 16384 lines). The probability that there will be a conflict miss when only a few random lines are allowed in the attack pattern becomes vanishingly small – this is the main reason why CEASER with a Remap-Rate of 1% is able to tolerate years of attacks.

If a faster search algorithm is used, then the attacker may be able to accommodate a larger number of lines in the attack pattern, which would make it more likely for the conflict to occur, and the attacker to succeed within a shorter period of time.

3.2 Proposed: Group-Elimination Method

Given L attack lines, the SHM algorithm holds back one line and tests the remaining (L-1) lines for a conflict. However, we note that out of the L lines (L is typically more than 1000), only a few lines will map to the conflicting set (e.g. 17 lines for a 16-way cache). We can make the search process more efficient by holding out a group of lines instead of just one line – in this way, we can discard the entire group if the remaining lines still cause a conflict, as it would mean none of the lines in the holdout group are necessary to cause a conflict miss. With this insight, we propose the *Group-Elimination Method (GEM)* search that is substantially faster at converging on the group of conflicting lines, and is shown in Algorithm 2.

Algorithm 2: Group-Elimination Method (GEM)

```
Input: L random lines that cause conflicts in 1 set
Output: "W+1" lines that map to the conflicting set

While (L has more than "W+1" lines):
    Split L into G groups
    For each of the G groups:
        Holdout the group
        Test remaining (G-1) groups for a conflict
        If conflict occurs, remove holdout group from L

Return L
```

Note that if the number of groups (G) were set to be equal to L, then GEM will degenerate into SHM. Figure 4 explains the differences between SHM and GEM for an attack pattern containing 28 lines (L=28) for a two-way cache (so, there are three conflicting lines, A, B, and X). With SHM, we hold 1 line and test the remaining 27 lines for a conflict. This will eliminate only one line (left most).

After 7 trials, we would be left with line "A" + 21 remaining lines. With GEM, we form 7 groups of 4 lines each. We hold one group and test the remaining 6 groups for a conflict. In the first test, the left-most group is eliminated. After 7 tests, all the groups that are shaded get eliminated and only 12 lines remain (the three groups containing A, B, and X). Thus, GEM is able to remove 2x more lines after 7 trials compared to SHM, and this efficiency only increases when there are thousand(s) of lines in the attack pattern, as would be expected for a multi-megabyte LLC.

3.3 Analytical Model for GEM Search

The effectiveness of GEM search depends on the group size. If the group size is too small, GEM becomes similar to SHM. If the group size is too large, then the likelihood of the group containing a conflicting line increases and it makes it less likely that the group will get eliminated from consideration (therefore dividing the number of attack lines into only two groups is not a useful option for a highly associative cache). To keep the algorithm simple, we set the number of groups (G) equal to the number of conflicting lines (W+1). The lines that form a group are selected randomly from L.

With G=(W+1), each group will have L/(W+1) lines, and the probability that the group does not contain any conflicting lines is given by Equation 1.

$$Prob\left(Conflict\ free\ Group\right) = \left(1 - \frac{\left(W+1\right)}{L}\right)^{\frac{L}{W+1}} \approx \frac{1}{e} \quad (1)$$

Thus, the probability that the group gets eliminated after being held out is approximately equal to $1/e$ (approximately 37%) for large L. After (W+1) trials, we would have tested each group, and the list is expected to shrink by 37% to 63%. This process is repeated iteratively until L reduces to (W+1) lines.[2] The total number of accesses for GEM to converge on the list of the conflicting lines is given by Equation 2.

$$Total\ Accesses = L \cdot W \cdot \left(1 + 0.63 + 0.63^2 + 0.63^3 + ...\right) \quad (2)$$

The number of terms that must be accounted in the above equation depends on the rounds it takes for L to reduce to (W+1). For the range of L that is suitable for modern LLCs, the total number of accesses is approximately equal to $2.3 \cdot W \cdot L$. For a cache with 16 ways, if the attacker starts with L lines then the number of accesses to learn the conflict list with GEM is approximately equal to $37 \cdot L$ as opposed to $O\left(L^2\right)$ with SHM algorithm. Given that L needs to be in the range of thousand(s) to have a non-negligible probability of causing a conflict miss in one of the cache set, GEM provides a significant reduction in search time (almost 35x faster for 1MB cache bank and 300x faster for an 8MB cache).[3]

[2]When the number of lines (L) in the attack becomes small (approximately $2.7 \cdot W$, or equivalently 44 lines for a 16-way cache), SHM becomes faster than GEM. When this threshold is reached, we set the number of groups (G) in GEM to L.

[3]We cannot prove the optimality of GEM. It is a valid question to ask whether there are substantially faster search algorithms compared to GEM. We were also curious to know, so, on Oct 5, 2018, we published a modified version of this problem (find "S" spies in "K" agents) on a website that regularly publishes mathematical puzzles [15]. The puzzle received several responses, however, even the best solutions had efficiency similar to GEM. This gives us some confidence that it is non-trivial to improve on GEM search.

3.4 Implications of GEM on CEASER

With GEM search, the attacker can form eviction sets quickly and the remapping rate of CEASER must be adjusted to take this fast search algorithm into account. Furthermore, for a given level of associativity, the GEM search reduces the time complexity of search from $O\left(L^2\right)$ to $O\left(L\right)$, which means that larger caches do not automatically become significantly more robust with CEASER as the time complexity of search is linear instead of quadratic. Table 1 shows the time for the attack to succeed for an 8MB LLC and a 1MB LLC-Bank with GEM for different Remap-Rate.[4] We use CEASER with a default Remap-Rate of 1% and SHM. The model for attack time is similar to that used in our prior work [6].

Table 1: Time for Attack to Succeed with CEASER under SHM and GEM search and Remap-Rate (R). Note that CEASER remains vulnerable only until the next remap (< 1 millisecond).

Remap-Rate (R)	Search	8 MB LLC	1 MB Bank
1%	GEM	< 1 second	< 1 second
10%	GEM	27 minutes	3 minutes
20%	GEM	9 days	1 day
30%	GEM	23 years	3 years
35%	GEM	> 100 years	20 years
1% (default)	SHM	> 100 years	> 100 years

Thus, to tolerate GEM search, the Remap-Rate of CEASER must be increased to 35%. This would mean remapping a 16-way set every 50 accesses to the cache. Each remap incurs reading the line, invaliding the original location of the line, and writing the remapped line into a new set. Under a Remap-Rate of 35%, the cache active power almost doubles, there are significantly more misses due to remapping, and there is longer latency due to increased cache contention. Our evaluations show that increasing the remapping rate of CEASER to 35% incurs an average slowdown of 6%. Ideally, we want to have randomized caches without incurring significant overhead due to remapping. Unfortunately, even if we were willing to pay for this significant overhead, the system may still not be secure to newer and faster attacks, which we discuss next.

4 ATTACK-2: EXPLOIT REPLACEMENT POLICY TO OBVIATE SEARCH

Our second attack is based on exploiting the cache replacement policy to avoid the search algorithm. Both SHM and GEM are agnostic of the replacement policy, and simply rely on the fact that when (W+1) lines of the attack pattern are mapped to the same set of a W-way cache, and accessed again, there will be at least one conflict miss. While this approach general and is applicable to arbitrary replacement policies, it is less efficient than an approach that exploits the replacement policy to launch an even faster attack. In this section, we show how faster "search-free" attacks are possible for popular replacement policies, such as LRU, RRIP, and Random.

[4]Note that the vulnerability with CEASER lasts only until the lines get remapped, so even when the attacker succeeds, this success is short-lived. The total time required to remap all the lines in CEASER is in the order of a few milliseconds and the attacker is forced to use most of this time to learn the eviction sets. Therefore, the numbers shown in Table 1 denote a vulnerability of less than 1 millisecond for a given attack period. For example, for a 1MB LLC-Bank with Remap-Rate of 10%, CEASER would be vulnerable for less than 1 millisecond every 3 minutes.

4.1 Attack Using LRU Policy

If CEASER is implemented using a cache that employs LRU, then the attacker can leverage the thrashing property of LRU to avoid the search. For example, if the attack contains "L" lines that cause a conflict in one of the set, then the attacker will first access all the L lines, and then test these lines for hit/miss in exactly the same sequence. For a W-way cache, the pattern will have at-least "W+1" misses, and all of the missing lines will be from the conflicting set. So, after just $2 \cdot L$ accesses, the attacker will have the list of conflicting lines, without the need for any search algorithm.

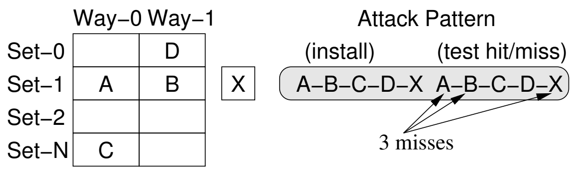


Figure 5: Fast attack by exploiting LRU policy. The attack needs 2L accesses and avoids a search algorithm.

We explain this attack with an example. Figure 5 shows a 2-way cache, where a pattern of 5 lines (A, B, C, D, X) is enough to cause conflict in one of the sets. The attacker will first access these five lines (A-B-C-D-X) to ensure that they are installed in the cache. Then the attacker will sequentially test if each line is present in the cache. There will be hits for lines C and D, and misses for lines A, B, and X. The lines that miss form an eviction set.

The attack exploiting LRU replacement is an order of magnitude faster than the GEM search. To tolerate this attack, the Remap-Rate of CEASER must be increased to more than 600% – remapping 6+ lines on every access to the cache, incurring impractical overheads.

4.2 Attack Using Reuse-Based Policies

When the number of lines exceed the capacity of the set, LRU is known to cause a thrashing like pattern. Several cache replacement policies have tried to address this behavior [8, 16–20]. Such replacement policies give preference to lines that have high reuse and try to do early eviction of lines that have low reuse. Without loss of generality, we will use RRIP [8] as the representative example.

The idea behind developing an attack using RRIP is to access each line in the access pattern twice. This ensures that the line gets promoted to a high-priority state, and the replacement policy would degenerate into something akin to LRU. We explain this attack with an example. Figure 6 shows a 2-way cache, where a pattern of five lines (A, B, C, D, X) is enough to cause conflict in one of the sets. The attacker will first access these five lines, each two times, to ensure that they are installed in the cache and upgraded to the high-priority state. Then the attacker sequentially tests if each line is present in the cache. There will be hits for lines C and D, and misses for lines A, B, and X. The lines that miss form an eviction set. This attack is equally applicable to replacement policies that are built on PC information (e.g. SHIP [17]) as the second access will update the line to high-priority state, regardless of the state at install.

To tolerate the attack exploiting RRIP, the Remap-Rate of CEASER must be increased to more than 500% – remapping 5+ lines on every access to the cache, incurring impractical overheads.

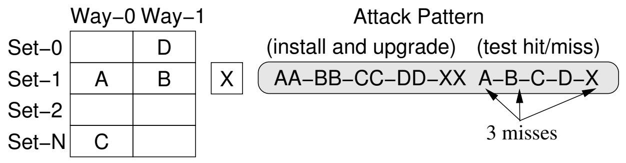


Figure 6: Fast attack by exploiting RRIP policy. The attack needs 3L accesses and avoids a search algorithm.

4.3 Attack Using Random Replacement

Both LRU and RRIP policies maintain a precise ordering of lines, and the attacks exploit the properties of these orderings. One may deem that we can avoid the vulnerability from replacement policies by simply using a random replacement policy, which avoids any ordering of lines within the cache set – so it would be much harder for the adversary to form a list of conflicting lines. We show that a cache with random replacement policy is still vulnerable to attacks.

Figure 7 shows the attack that uses the properties of random replacement to avoid the search algorithm. The attacker installs the given line "X" in the cache. Then the attacker installs a random line (Y) and tests if X is still present. There is a probability $p = 1N$ (N is the number of lines in the cache) that Y will evict X. If there is a miss for X, we know that Y conflicts with X, so we add Y to the conflict list. This process is repeated until the list has W elements.

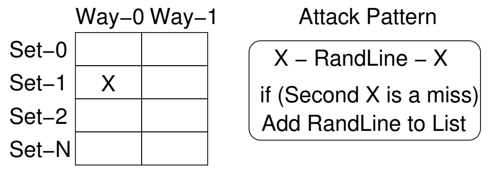


Figure 7: Fast attack by exploiting Random policy.

On average, it will take the attacker $2 \cdot W \cdot N$ accesses to form the conflict list ($32 \cdot N$ for a 16-way cache). The number of accesses in this attack pattern is smaller than the number of accesses possible ($100 \cdot N$) during the remapping period of CEASER with a default Remap-Rate of 1%. Thus, CEASER with the default Remap-Rate remains vulnerable to an attack under a Random Replacement policy, albeit this vulnerability is not as severe as with LRU or RRIP.

4.4 Implications of Attacks on CEASER

To avoid the vulnerability due to replacement policies, a designer may try to combine random replacement and LRU/RRIP to make attacks harder, however, the robustness of such a scheme will again depend on the details of the implementation. Note that, even for any arbitrary replacement policy (the details of which are not known to the attacker), the attacker can always use GEM to determine the conflicting lines and this would necessitate a Remap-Rate of 35%.

Ideally, we want to be able to use any replacement policy for CEASER (without being concerned about its impact on the security), have a low remapping rate (no more than a few percent), and still have strong security (able to tolerate years of attack). We discuss such a robust and practical design next.

5 SKEWED-CEASER

An attack on CEASER consists of three parts: First, form L lines such that there is one set that has a conflict. Second, obtain a list of W conflicting lines (for a W-way cache) that can evict the given address from the cache. Third, use this conflict list to launch a conflict-based attack on the victim. The process of learning the conflict list must be repeated after remapping. The key challenge with the attacks described in Section 3 and Section 4 is that they can learn the list of conflicting lines quickly, and to avoid this we needed to significantly increase the Remap-Rate of CEASER. Our solution for secure and low-overhead extension of CEASER is inspired from Seznec's work on skewed-associative caches [9].

5.1 Flexibility of Skewed Caches

In a conventional set-associative cache, each way of the cache uses the same function to hash the line address to the set. CEASER is built on top of a set-associative cache, therefore, it suffers from the same limitation, in that all the ways of the cache use the same indexing function to map the encrypted line to the set of CEASER. In a skewed-associative cache, each way of the cache uses a different hashing function, so a given line can map to different sets in different ways. For example, as shown in Figure 8(b), the skewed cache uses two hashing functions h1 and h2 to index the respective ways of the cache. A given line X could be located in Set-1 for Way-0 but Set-2 for Way-1, based on the different hashing functions. While skewed-cache reduces conflict misses when the cache has lower associativity, its effectiveness reduces when the cache is highly associative. Therefore, modern designs of highly-associative LLC typically do not opt for a skewed organization (for example, our evaluations show that implementing the baseline 8MB 16-way LLC as a skewed-associative cache provides less than 0.1% speedup).

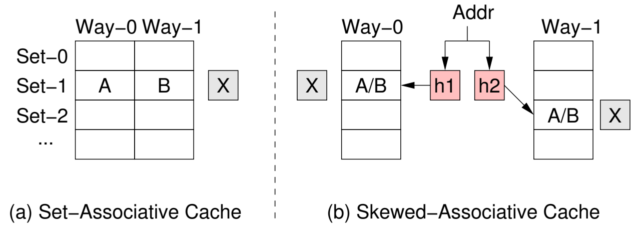


Figure 8: Forming eviction sets with (a) Set-Associative Cache (b) Skewed-Associative Cache, with two hashing functions h1 and h2. The conflicting lines A and B must dislodge the target line "X" from both possible locations under h1 and h2.

5.2 Robustness Due to Multiple Locations

While skewed-cache has negligible impact on performance of a highly associative cache, they provide an important property – each line can be resident in multiple possible sets, therefore the groups of lines with which the given line may conflict keep changing depending on the way in which the line is resident. This property has important security implications, as it will make it much harder for the adversary to form an eviction set, as the eviction set depends on the way in which the line is resident. Figure 8(a) and (b) shows the eviction set for the target line "X" for a set-associative cache and a skewed-associative cache, respectively.

Let there be S sets in the cache. To evict the target line X from the two-way set associative cache, we will need two lines (A and B) such that they map to the same set as X. Let us pick two random lines, A and B to form an eviction set for X. The probability that two random lines A and B map to the same set as X is $1/S^2$. Therefore, the probability that we can dislodge X with two random lines in a set-associative cache is $1/S^2$. In such a scenario, we can use A-B to repeatedly dislodge X from the set-associative cache.

For the skewed-cache, let us again pick two random lines, A and B, to form an eviction set with X. However, to be able to guarantee that line X is evicted from the skewed cache, we will need the two lines A and B such that they can dislodge X from both possible locations.[5] The probability that A and B map to the same set in Way-0 and Way-1 as line X is $1/S^4$. Thus, skewed-cache can offer S^2 resilience in robustness compared to the set-associative implementation. Given that modern LLCs have thousand(s) of sets, this represents several orders of magnitude improvement in robustness.

5.3 Limitations of Static Hashing

Note that the security of the skewed-cache design depends on the unpredictability of the hashing functions. Skewed cache is typically implemented with static hashing functions (that stay constant throughout the machine up-time and across different machines). Such hashing functions can be learned by an adversary and then it would be easy to attack the skewed-cache design. For example, if the attacker knows the hashing functions, h1 and h2, then the attacker will form the attack pattern using only the lines that go to the same set as the target address under both h1 and h2. Thus, for security, it is important that the hashing functions of skewed-cache are both (a) unpredictable and (b) dynamically changing. We combine the principles of skewed-cache and CEASER to form a robust and practical design, called *Skewed-CEASER (CEASER-S)*, while obviating the need for defining unpredictable and dynamic hash functions.

5.4 Design of Skewed-CEASER

Similar to skewed-cache, CEASER-S allows each line to get mapped to different sets, depending on the way in which the line in resident. However, unlike skewed-cache, CEASER-S uses multiple instances of CEASER, instead of hashing functions, to determine the set mappings. CEASER-S divides the cache ways into multiple partitions, and uses a different set of encryption keys for each partition, therefore the given line gets mapped to a different set in each partition. The default CEASER-S design contains two partitions, whereas, the design can be generalized into *P* partitions (each containing W/P ways) and is denoted as *CEASER-SP*. For example, CEASER-S design with four partitions would be denoted as CEASER-S4. Note that CEASER-S1 would be the same as the original CEASER design.

For simplicity, we describe the organization of CEASER-S assuming two partitions. The baseline set-associative cache has S sets and W ways. CEASER-S would logically split the cache into two halves: Left and Right, each containing S sets and W/2 ways. Each half implements CEASER with different sets of encryption keys, therefore a given line will map to a different set in each partition.

[5]If one of the lines (say line A) conflicts with X only in Way-0 but not in Way-1, then if the pattern of A-B is repeated, then A can get mapped to a non-conflicting location in Way-1. This will mean only two lines, B and X, will be contending with each other and both can reside conflict-free in the two ways of a skewed-cache.

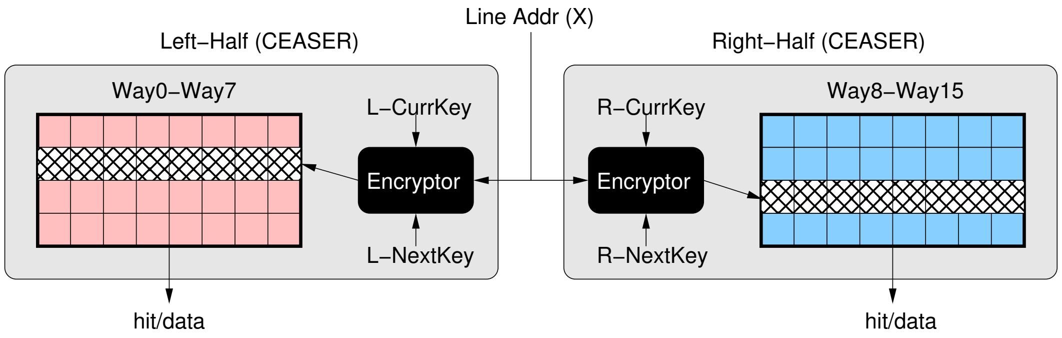


Figure 9: CEASER-S: The 16-way cache is logically split into two halves: left and right, each containing 8 ways. Each half operates as an independent 8-way CEASER with a different set of encryption keys (so a line gets mapped to a different set in each half).

Organization: Figure 9 shows an overview of CEASER-S. The 16-way baseline cache is split into two 8-way caches, and CEASER is implemented in each half using an independent set of keys. Note that an implementation of CEASER with gradual mapping requires two keys (CurrKey and NextKey). The left half uses keys L-CurrKey and L-NextKey, and the right uses R-CurrKey and R-NextKey.

Access: On an access to a convention cache, all the W-ways of the cache are checked for a hit. In CEASER-S, the W/2 ways in each of the left-half and the right-half are checked concurrently (so the number of checks remains the same as the baseline W-way cache).

Hit: If there is a hit in either half, the cache returns the data from the line that had the hit and updates the replacement state.

Miss: On a miss, CEASER-S randomly picks the half in which to install the line. As each half has eight ways, the victim is decided by the replacement policy (LRU, RRIP, SHiP etc.) of the chosen half.

Remapping: For controlling the remapping of CEASER-S, we use the same parameter, *Remap-Rate (R)* and perform remapping at a *set* granularity. With the default Remap-Rate of 1% and a 16-way cache, after 1600 accesses, one set gets remapped in each half.

5.5 Security Analysis of CEASER-S

In this section, we analyze the security for the default implementation of CEASER-S with two partitions (we note that CEASER-S becomes even more robust with more partitions). We want CEASER-S to be robust to any search algorithm and replacement policy. Therefore, we keep the security analysis of CEASER-S independent of these two choices. For an adversary to repeatedly evict a given address with *W* conflicting lines, the adversary needs to find W lines such that they conflict with the target address in both the Left-Cache and Right-Cache of CEASER-S (if the line conflicts in only one half, then half of such lines are expected to get mapped to a non-conflicting set in the other half and will not participate in evictions). We call such a line that conflicts on both locations as a *hard-conflict* line. If the cache has S sets, the probability that a random line will have a hard conflict is given by Equation 3.

$$P = Prob\left(HardConflict\right) = \frac{1}{S^2} \tag{3}$$

Given a Remap-Rate of R and a cache containing N lines, the epoch (E) of CEASER-S is given by Equation 4.

$$EpochPeriod\left(E\right) = \frac{N}{R} \tag{4}$$

All lines are guaranteed to undergo remapping within the time period equal to one Epoch. Therefore, E bounds the available time for the attacker to converge on at-least W lines that have a hard conflict. The probability that there are W or more hard-conflict lines in E accesses is given by a binomial distribution as shown in Equation 5.

$$Prob\left(\geq W\ in\ E\right) = 1 - \Sigma_{k=0}^{k=(W-1)} \binom{E}{k} \cdot P^k \cdot \left(1-P\right)^{E-k} \tag{5}$$

Assumptions: To derive the time for a successful attack, we make three assumptions: (1) It is sufficient to simply have *W* hard conflicts in the attack pattern, without needing to identify which lines are hard conflicts (so, we assume an idealized search algorithm). (2) No other application accesses the cache during the epoch period (otherwise, the attacker can only do a reduced fraction of the accesses in the epoch). (3) The attacker has enough time to not only form an eviction set but also to attack the victim (if most of the epoch goes in forming an eviction set, it would leave little time to attack).

Even when an attacker succeeds in forming an eviction set, this vulnerability gets removed when the lines get remapped. Given that it takes a few milliseconds to remap the entire cache and the attacker is forced to spend most of the time in learning the eviction set, the window of vulnerability is quite small (1 millisecond or less). Table 2 shows the time for a successful attack, as the Remap-Rate of CEASER-S is varied from 1% (default) to 0.1%. We perform this analysis for both the baseline 8MB cache and a 1MB bank (attacker focuses on one bank, CEASER-S implemented per bank).

Table 2: Vulnerability of CEASER-S (approximately 1 millisecond of vulnerability every period of successful attack)

Remap-Rate (R)	8MB LLC	1 MB Bank of LLC
1% (default)	1 ms every 100+ years	1 ms every 18 years
0.5%	1 ms every 100+ years	1 ms every day
0.1%	1 ms every 68 years	1 ms every second

Thus, even under the severely conservative assumptions, and the default Remap-Rate of 1%, CEASER-S can tolerate years of continuous attacks (providing a vulnerability of approximately 1 millisecond every 18 years). Note that CEASER-S provides two degrees of freedom – the robustness of CEASER-S can be enhanced even further by increasing the Remap-Rate, or the number of partitions, or both.

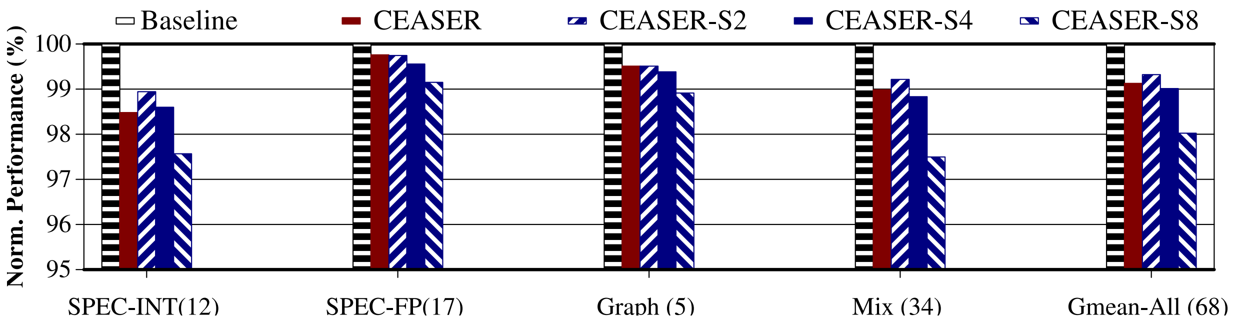


Figure 10: Normalized performance of CEASER and CEASER-S (with different number of partitions) with respect to the baseline. The number beside the category denotes the number of workloads in that category. CEASER-S2 and CEASER-S4 incurs an average slowdown of 0.7% and 1.0% respectively across all workloads (unlike baseline and CEASER, they can tolerate years of attack).

6 EXPERIMENTAL METHODOLOGY

6.1 Configuration

We use a Pin-based x86 simulator with a detailed memory model. Table 3 shows the configuration used in our study. The L3 cache is 8MB shared between all the cores and incurs a latency of 24 cycles. All caches use a linesize of 64 bytes. For all implementations of CEASER and CEASER-S, we use a latency of two cycles for encryption and decryption. The default Remap-Rate for both CEASER and CEASER-S is assumed to be 1% (one 16-way set gets remapped every 1600 accesses to the LLC). For all LLC implementations, including the baseline, we use the SRRIP policy.

Table 3: Baseline Configuration

Processor	
Core parameters	8-cores, 3.2GHz
L1 and L2 cache	32KB, 256KB 8-way (private)
Last Level Cache	
L3 (shared)	8MB, 16-way, 24 cycles SRRIP [8] replacement policy
DRAM Memory-System	
Bus frequency	800 MHz (DDR 1.6 GHz)
Channels	2 (8-Banks each, 2KB row buffer)
tCAS-tRCD-tRP-tRAS	9-9-9-36

6.2 Workloads

We use a diverse set of workloads for our study, including all the 29 workloads from the SPEC2006 benchmarks suite and five workloads from the GAP benchmark suite [21]. For each benchmark, we use a representative slice [22] of one billion instructions. These 34 benchmarks are run in rate-mode where each core runs a copy of the benchmarks. Additionally, we use 34 mixes formed using random combinations of the 34 benchmarks.

We perform timing simulation until all benchmarks in the workload finish executing a minimum of one billion instructions. For measuring aggregate performance, we use the weighted speedup. We report normalized performance as the ratio of weighted speedup of the given design to the baseline.

7 RESULTS

7.1 Impact on Performance

Figure 10 shows the performance of CEASER and CEASER-S (with the number of partitions varying from 2 to 8). Note that the performance is normalized to the baseline (so higher is better, and 100% denotes no degradation). We report the geometric mean across the workload suites. Overall, across all 68 workloads, CEASER causes an average degradation of 1% whereas the default CEASER-S (with two partitions) causes a degradation of 0.7%. With more partitions the overhead of CEASER-S increases, becoming 1% for four partitions (CEASER-S4) to 2% for eight partitions (CEASER-S8). This happens because, with a larger number of partitions, each partition gets a reduced number of ways, and this limits the ability to employ intelligent replacement policies to select the best victim within the partition. For example, CEASER-S16 would degenerate the cache into using random replacement, as the victim partition would get selected randomly and the partition would contain only 1 way. We recommend using CEASER-S with 2 to 4 partitions. We use two partitions in our default implementation of CEASER-S.

7.2 Sensitivity to Remap-Rate

We use a default Remap-Rate (R) of 1% and show that this rate is sufficient to provide strong security (tolerate years of attack) and that the performance overhead with this rate of remapping is quite small (within 1%). Figure 11 shows the performance of CEASER-S when the Remap-Rate is increased from 1% to 10%. At lower Remap-Rate, the performance impact is negligibly small.

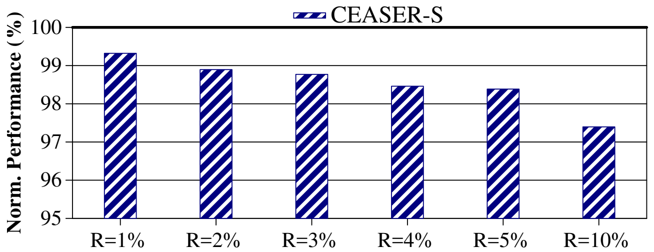


Figure 11: Impact of Remap-Rate (R) on CEASER-S.

7.3 Impact on Cache Misses

CEASER affects cache misses due to randomization (which can affect conflict misses, either positively or negatively) and remapping (the remapped line may evict a useful line from the remapped set). We analyze the impact of CEASER and CEASER-S on the MPKI (misses per 1000 instructions) of the LLC. Table 4 shows the MPKI of the baseline 8MB LLC, CEASER, and CEASER-S for the 34 rate-mode workloads.

Table 4: Impact of MPKI of LLC (8MB).

Workload	Baseline	CEASER	CEASER-S
astar	0.5	0.7	0.5
bwaves	18.7	18.7	18.7
bzip2	3.5	3.8	3.7
cactusADM	5.3	5.2	5.2
calculix	0.0	0.0	0.0
dealII	2.4	2.4	2.4
gamess	0.0	0.0	0.0
gcc	16.6	16.9	16.7
GemsFDTD	9.8	9.8	9.8
gobmk	0.4	0.4	0.4
gromacs	0.5	0.6	0.6
h264ref	0.5	0.6	0.6
hmmer	0.5	0.8	0.7
lbm	31.9	31.9	31.9
leslie3d	7.6	7.6	7.6
libquantum	25.4	25.4	25.4
mcf	67.8	69.0	69.0
milc	25.8	25.6	25.6
namd	0.1	0.1	0.1
omnetpp	21.0	21.2	21.2
perlbench	0.8	0.8	0.8
povray	0.0	0.0	0.0
sjeng	0.4	0.4	0.4
soplex	26.9	26.9	26.9
sphinx3	11.6	11.1	11.2
tonto	0.1	0.1	0.1
wrf	6.6	6.6	6.6
xalancbmk	2.2	2.2	2.2
zeusmp	4.8	4.8	4.8
BC	84.5	84.6	84.5
BreadthFS	37.2	37.3	37.3
ConnComp	85.8	86.0	86.0
PageRank	46.0	46.2	46.2
SSSPath	118.8	119.0	119.0
Average	19.5	19.6	19.6

For sphinx3, randomization causes the MPKI to get reduced from 11.6 in the baseline to 11.1 with CEASER and 11.2 with CEASER-S, and this reduction in MPKI results in slight performance improvement compared to the baseline. For other workloads, the impact on MPKI is negligible. Overall, similar to CEASER, CEASER-S increases the average MPKI of the baseline from 19.5 to 19.6.

7.4 Sensitivity to LLC Capacity

Our default configuration contains an 8MB shared LLC. Figure 12 shows the performance of CEASER and CEASER-S when the capacity of LLC is varied from 8MB to 64MB (performance is normalized against the baseline with an equal-capacity LLC). Across all LLC sizes, the average slowdown of CEASER-S is between 0.7% and 1.5%. The increased slowdown at large LLC happens mainly because the 2-cycle decryption latency plays a larger role when the execution time gets reduced due to the higher hit-rate of the larger LLC.

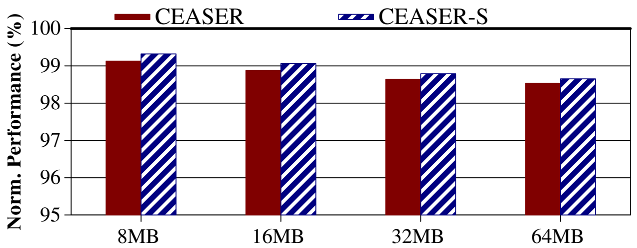


Figure 12: Impact of LLC-Capacity on Overheads.

7.5 Implications on Energy and Power

The power overheads of CEASER-S comes from the encryption circuit, the remapping of lines (one line per 100 access), and the extra misses (0.4% extra memory accesses). With CEASER-S, the overall system power increases by approximately 0.3% and the overall system energy by less than 1% (mainly due to the slowdown).

7.6 Logic and Storage Overheads

With CEASER-S, the cache is logically operated as two half-sized caches, each requiring the overheads of CEASER. CEASER requires two block-cipher circuits and selection logic to select between these two circuits. The logic overhead of CEASER (for the two block-cipher circuits and the selection logic) is less than 3500 two-input gates, which is quite small (similar to computing SECDED code for a 64-byte line). CEASER-S doubles the logic overheads to approximately 7000 gates, which is still negligibly small.

Similar to CEASER, CEASER-S requires the 1-bit Epoch-ID with each line (to identify whether the CurrKey or the NextKey was used). Each half also needs storage for the global metadata such as two keys (80 bits each), a SetPointer, and an access counter, for a total overhead of 24 bytes. So the total storage overheads of CEASER-S gets doubled to 48 bytes, which is still quite small.

7.7 CEASER-S with More Partitions

We describe a default CEASER-S design with two partitions, where each partition has half the number of ways. This design allows each line to map to two possible locations and we found that such a design is sufficient for strong security. The CEASER-S design can be generalized to P partitions ($P \leq W$, where W is the number of ways in the cache), with each partition containing W/P ways. With increased P, the design offers even more robustness as a line can map to P possible sets and the attacker will need to dislodge the line from all of the possible locations. The logic and storage overheads of this design increases in proportion to P. Note that, even with four partitions, the storage overhead remains less than 100 bytes.

8 RELATED WORK

Our paper analyzes the robustness of randomized caches at mitigating conflict-based attacks. We discuss the closely related work.

8.1 Randomization-Based Mitigation

Randomization-based techniques rely on randomized mapping of the memory line to the cache, thereby making it harder for the adversary to form an eviction set. CEASER [6] uses encrypted address and remapping for cache randomization. Prior proposals [2, 5] use mapping tables to track the location of the line in the cache. While such tables can be implemented efficiently for small L1 caches (using CAM lookups [5]), they become impractically large for a multi-megabyte LLCs. Furthermore, such *Table-Based Randomization (TBR)* schemes need OS to classify applications into protected and unprotected groups (otherwise, the mapping table can be attacked).

Table 5 shows the storage overheads of TBR with OS support (shared mapping table), TBR without OS support (private per-core mapping tables), CEASER, and CEASER-S. TBR requires mapping tables exceeding 1MB (with OS support) or exceeding 8MB (if no OS support is provided). CEASER-S not only incurs negligible storage but also avoids the latency of looking up large tables. Furthermore, CEASER-S provides stronger security than CEASER.

Table 5: Storage Overhead for Additional Structures

Scheme	Storage Overheads (for 8MB LLC)
Table-Based (with OS support)	1.25 megabytes
Table-Based (without OS support)	8.5 megabytes
CEASER (OS support not required)	24 bytes
CEASER-S (OS support not required)	48 bytes

A concurrent work [23] (published at IEEE S&P 2019) looks at the theory and practice of finding eviction sets in a randomized cache. This work also developed an efficient algorithm for learning eviction set that is similar to GEM. In our paper, we not only propose the GEM attack but also another attack (based on exploiting the replacement policy), which is an order of magnitude faster than GEM. We also show that our proposed solution of CEASER-S is robust to both these attacks.

8.2 Isolation-Based Mitigation

Isolation-based techniques rely on preserving the data of the victim application, by making it harder for the attacker to evict the data of the victim application. This is typically done by allocating dedicated cache space to protected (security-sensitive applications) [2][24][25]. *Non-Monopolizable (NoMo) Cache* [3] and *Dynamically-Allocated Way Guard (DAWG)* use way partitioning to isolate the cache allocated to different cores. The efficacy of these schemes relies on the ability of the OS to carefully classify applications. Ideally, we want to mitigate attacks without relying on any OS or software support.

Note that techniques that rely on way-partitioning can result in inefficient use of cache space, as cache resources may get allocated to the applications, regardless of the reuse characteristics. Furthermore, way partitioning is limited in scaling by the number of ways and starts to become effective when the number of cores exceeds the number of ways in the cache (e.g. 32-core chip with a 16-way LLC).

8.3 Tolerating Flush-Based Cache Attacks

Our paper is focused on mitigating conflict-based attacks. However, the attacker can use the `clflush` instructions to explicitly evict the lines that are shared between the attacker and the victim. Techniques that use randomization alone are ineffective against *Flush-based* attacks, as the attack is insensitive to the location of the line in the cache. Instead, to mitigate Flush-based attacks, prior studies advocate *duplicating* the shared line within the cache using either the ProcessID [26] or the DomainID [4], so that both the victim and the attacker can get their own copy of the shared line, and evicting the attacker line would not dislodge the victim line. CEASER-S is fully compatible with such a solution. For example, accessing CEASER-S using a combination of the ProcessID/DomainID and the physical line address (as done in prior works) can avoid Flush-based attacks by automatically creating duplicates of shared lines.

9 CONCLUSION

Randomizing the line-to-cache mapping is an effective way of mitigating conflict-based cache attacks. Our recent work, CEASER, makes randomized caching practical by accessing the cache with an encrypted line address and using periodic remapping of cache contents. The efficacy of a randomized cache is dictated by how quickly the attacker is able to form eviction sets. In this work, we provides two new attacks that significantly push the state-of-the-art in forming evictions set, namely (1) We develop a faster search algorithm that can quickly converge on an eviction set, given a large number of random lines that cause a conflict in one of the cache set, (2) We develop an attack that exploits the cache replacement policy to form eviction sets at a rate that is orders of magnitude faster than currently known attacks. To mitigate these faster attacks, the Remap-Rate of CEASER must be increased beyond the practical limits.

To enable cache randomization, while incurring negligible remapping overheads, we propose *Skewed-CEASER (CEASER-S)*, a design that divides the cache ways into multiple partitions and allows the line to be mapped to a different set in each partition. Such a design makes it harder for the adversary to form an eviction set as the line could be in multiple possible locations, and the conflicting lines are not guaranteed to map to the same partition as the target line. Our analysis shows that CEASER-S provides strong security (tolerates several years of attack), has low performance overhead (within 1%), requires a storage overhead of less than 100 bytes for the added structures, and does not need any OS support. The robustness of CEASER-S can be further increased by either increasing the Remap-Rate, or the number of partitions, or both.

While we analyzed our solution only for shared LLC, it is also applicable to other structures, such as the coherence directory [27] that may be vulnerable to conflict-based attacks. As cache attacks become common [28] [29] [30] [31], it has become vital to develop practical and effective solutions for protecting cache structures.

DEDICATION

This paper is dedicated to the memory of Prof. Sudhakar Yalamanchili. Sudha was a true mentor who helped me grow as a young faculty member at Georgia Tech. He was always available, always willing to listen, always willing to share that elderly advise, always trying to form bridges between groups. We will miss you Sudha.

ACKNOWLEDGEMENTS

Special thanks to Andre Seznec for discussion and feedback. We thank Olivier Roeder for posting the "spy problem" [15] on fivethirtyeight.com. We thank Sung Ha Kang, Greg Bleckherman, Hyesoon Kim, Swati Gupta, Tushar Krishna, Vinson Young, and Sanjay Kariyappa for solutions and/or discussions of the spy problem. We also thank Gururaj Saileshwar and Prashant Nair for discussions.

REFERENCES

[1] D. J. Bernstein, "Cache-timing attacks on AES," tech. rep., 2005.

[2] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[3] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 8, Jan. 2012.

[4] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," in *51st Annual IEEE/ACM International Symposium on Microarchitecture*, Oct 2018.

[5] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 83–93, 2008.

[6] M. Qureshi, "CEASER: Mitigating Conflit-Based Cache Attacks via Encrypted-Address and Remapping," in *51st Annual IEEE/ACM International Symposium on Microarchitecture*, Oct 2018.

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

[8] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 60–71, ACM, 2010.

[9] A. Seznec, "A case for two-way skewed-associative caches," in *Annual International Symposium on Computer Architecture (ISCA)*, 1993.

[10] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed compressed caches," in *Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[11] E. Quiñones, E. D. Berger, G. Bernat, and F. J. Cazorla, "Using randomized caches in probabilistic real-time systems," in *21st Euromicro Conference on Real-Time Systems, ECRTS 2009, Dublin, Ireland*, 2009.

[12] V. Young, C. Chou, A. Jaleel, and M. Qureshi, "Accord: Enabling associativity for gigascale dram caches by coordinating way-install and way-prediction," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 328–339, 2018.

[13] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2006.

[14] C. Percival, "Cache missing for fun and profit," in *The Technical BSD Conference*, 2005.

[15] *How Much Will It Cost To Sniff Out The Spies? (Riddler Nation at FiveThirtyEight.com on Oct 5, 2018).* https://fivethirtyeight.com/features/how-much-will-it-cost-to-sniff-out-the-spies/.

[16] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, (New York, NY, USA), pp. 381–391, ACM, 2007.

[17] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 430–441, ACM, 2011.

[18] V. Young, C.-C. Chou, A. Jaleel, and M. Qureshi, "Ship++: Enhancing signature-based hit predictor for improved cache performance," in *The 2nd Cache Replacement Championship (CRC-2 Workshop in ISCA 2017)*, 2017.

[19] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[20] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[21] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 45–57, Oct. 2002.

[23] P. Vila, B. Kopf, and J. F. Morales, "Theory and practice of finding eviction sets," in *2019 2019 IEEE Symposium on Security and Privacy (SP)*, pp. 695–710, may 2019.

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

[25] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *21st USENIX Security Symposium*, 2012.

[26] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 341–353, ACM, 2017.

[27] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[28] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[29] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

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

[31] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *51st Annual IEEE/ACM International Symposium on Microarchitecture*, Oct 2018.