2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping

Moinuddin K. Qureshi
Georgia Institute of Technology
Atlanta, USA
moin@ece.gatech.edu

***Abstract*—Modern processors share the last-level cache between all the cores to efficiently utilize the cache space. Unfortunately, such sharing makes the cache vulnerable to attacks whereby an adversary can infer the access pattern of a co-running application by carefully orchestrating evictions using cache conflicts. Conflict-based attacks can be mitigated by randomizing the location of the lines in the cache. Unfortunately, prior proposals for randomized mapping require storage-intensive tables and are effective only if the OS can classify the applications into protected and unprotected groups. The goal of this paper is to mitigate conflict-based attacks while incurring negligible storage and performance overheads, and without relying on OS support.**

This paper provides the key insight that randomized mapping can be accomplished efficiently by accessing the cache with an encrypted address, as encryption would cause the lines that map to the same set of a conventional cache to get scattered to different sets. This paper proposes *CEASE*, a design that uses *Low-Latency Block-Cipher (LLBC)* to translate the physical line-address into an encrypted line-address, and accesses the cache with this encrypted line-address. We analyze efficient designs for LLBC that can perform encryption and decryption within two cycles. We also propose *CEASER*, a design that periodically changes the encryption key and performs dynamic-remapping to improve robustness. CEASER provides strong security (tolerates 100+ years of attack), has low performance overhead (1% slowdown), requires a storage overhead of less than 24 bytes for the newly added structures, and does not need any OS support.

I. Introduction

Caches alleviate the long latency of main memories by providing data with low latency. Unfortunately, the timing difference between the cache-hit and a cache-miss can be used as a *side-channel* by an adversary to infer the access pattern and obtain unauthorized information from the system. The recently disclosed Spectre [1] and Meltdown [2] vulnerabilities rely on such cache-based side channels to convert the unauthorized data value into a discernible information. While cache attacks have been demonstrated in the past at a smaller scale, the recent vulnerabilities show that cache attacks can affect hundreds of millions of processor systems, and highlight the need to develop efficient solutions to mitigate cache attacks.

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, and use this access pattern to infer secrets (such as AES keys [3]). Conflict-based attacks are feasible when the adversary and the victim share some storage structures. Modern processors pack multiple cores on a single chip and tend to keep the level-1 (L1) and level-2 (L2) caches private to the core. However, the last-level cache (LLC) is typically shared between all the cores to efficiently utilize the cache space. Unfortunately, such sharing makes the LLC vulnerable to cache attacks as an adversary can learn about the access pattern of the victim using LLC evictions, even when the adversary and the victim are executing on different cores. The goal of our paper is to efficiently protect the LLC against such attacks without relying on any software support.

Architectural solutions to mitigate conflict-based cache attacks broadly fall in two categories. First, preservation-based mitigation [4]–[7], 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 results in inefficient use of cache space. Second, randomization-based mitigation [4], [8], whereby the location of the line in the cache is determined randomly and this information is stored in a table. To protect the mapping table from being attacked, the OS is required to group the applications into protected and unprotected groups and only the protected applications are allowed to use the mapping table and randomization. While such *Table-Based Randomization* may be feasible for L1 cache, the size of the mapping tables becomes impractically large for the LLC, as the number of entries in the mapping table increases linearly with the number of cache lines and the number of concurrently running protected applications. Furthermore, the efficacy of prior randomization-based scheme is dependent on the ability of the Operating System (OS) to correctly classify applications into protected and unprotected groups. Ideally, we want a solution that does not incur the storage overhead of large indirection tables and does not require any OS support, and yet provides high performance and strong security.

To develop a practical solution against conflict-based cache attacks, we focus on the set indexing function of the cache, as this function determines the group of lines that get mapped to a given set. To successfully launch an attack, the adversary must find lines that map to a given set – the group of lines that map to the same set of the cache and can cause an eviction is called an *Eviction Set*. The LLC is accessed with a physical line-address (PLA). Conventional LLC designs use a static hash-function (bottom few bits of the PLA) to determine the set, as shown in Figure 1(a), which means an adversary can easily form an eviction set. Even if a complex hash-function

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00068

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

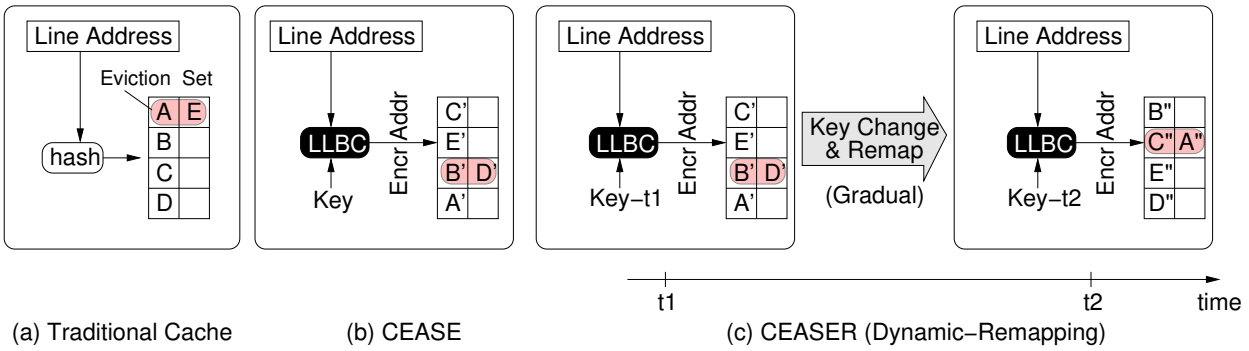

Fig. 1. Mapping of memory lines to the cache locations (a) conventional systems rely on static hashing to determine cache index (b) CEASE encrypts the the physical line-address and uses it to index the cache – the mapping of memory lines to cache index depends on the Encryption-Key (d) CEASER changes the key periodically and performs dynamic remapping, which limits the duration for which the memory line to cache index mapping remains constant.

is used and it is not revealed to the users, an adversary can use timing attacks to determine the eviction set for such a design [9], [10]. Once an adversary infers the mapping for one machine, the adversary can use this information to attack all machines that use the same hashing function.

We provide the key insight that randomized mapping of memory lines to cache locations can be accomplished efficiently by operating the cache on an *Encrypted Address Space (EAS)* instead of the *Physical Address Space (PAS)*, as shown in Figure 1(b). The avalanche effect of encryption would cause lines that have spatial correlation in the PAS (such as those mapping to the same set of the cache) to get scattered throughout the space in EAS. This scattering would happen in an unpredictable fashion, and gets dictated by the Encryption-Key. For example, lines A and E were mapped to the same Set in Fig 1(a), however, in Figure 1(b) A and E get mapped to different sets, and for another key they would get mapped to some other sets. Leveraging this insight, we propose *CEASE*, a Cache operated on Encrypted Address-Space.

CEASE employs a *Low-Latency Block-Cipher (LLBC)* to convert the b-bit *Physical Line-Address (PLA)* into a b-bit *Encrypted Line-Address (ELA)*, and uses this ELA to access the cache. As cache access latency is critical to performance, we study potential designs for LLBC that can perform encryption and decryption with low latency. For our design, we use of a four-stage Feistel-Network [11] that can perform encryption/decryption within 2 cycles, while consuming a storage overhead of ten bytes (for the Encryption-Key). The Encryption-Key of CEASE is initialized to a random value on every reboot, so the mapping of PLA-to-ELA is different for each machine, and even for the same machine this mapping changes every time the machine is restarted. When a dirty line is evicted from the cache, CEASE uses the LLBC to perform decryption and convert the ELA of the evicted line to obtain the PLA of the evicted line, and uses this PLA to perform the writeback. Thus, the Encrypted Line-Address (ELA) is visible only within the LLC, and the operations of rest of the memory system (such as coherence requests, prefetch, writeback) remain unchanged and continue to be performed using the Physical Line-Address (PLA).

The PLA-to-ELA mapping of CEASE gets dictated by the Encryption-Key, and for a given key, this mapping remains constant. Given enough time, an adversary can still launch a timing-based attack to determine which group of lines map to the same set (we discuss such an attack in Section III-E). CEASE can be made resilient against such attacks by periodically changing the keys and performing dynamic-remapping of the cache lines based on the new key, as shown in Figure 1(c). At time t=t1, the LLBC has key *Key-t1*, and uses a particular mapping. Over time, a new key *Key-t2* is used to remap the contents of the cache and when the conversion finishes the key is updated to Key-t2. We call such a variant of CEASE that performs dynamic-remapping as *CEASER (CEASE with Remapping)*. We perform a bin-and-balls analysis and demonstrate that CEASER provides strong security (tolerates 100+ years of attack) even if the remapping of one cache line is performed per every 100 accesses to the cache, thus limiting the remapping overhead to 1%.

Overall, this paper makes the following contributions:

1) To the best of our knowledge, this is the first paper to advocate operating the on-chip caches on an encrypted address space to mitigate cache attacks. The proposed CEASE design randomizes mapping of memory lines to cache locations without requiring any indirection tables.
2) As cache access latency is critical for performance, we present a practical design for the Low-Latency Block-Cipher. Our design uses a four-stage Feistel-Network, which can perform encryption/decryption within 2 cycles and incurs negligible storage overhead.
3) We propose CEASER (CEASE with Remapping), whereby the keys are changed periodically and the contents of the cache are gradually remapped from the old-key to the new-key. We perform analysis to determine the rate of remapping and show that remapping one line every 100 accesses is sufficient for strong security.

Our evaluation with 134 workloads shows that CEASER incurs a slowdown of only 1.1%. The newly added structures of CEASER incur a storage overhead of less than 24 bytes. CEASER provides strong security (tolerates 100+ years of attack) and does not require any OS/software support.

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

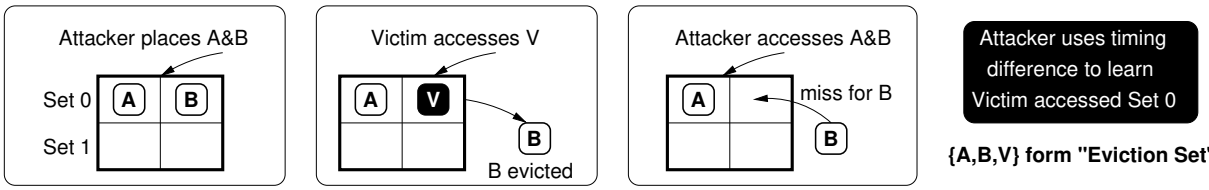

Fig. 2. Example of "Prime+Probe" cache attack. The attacker uses lines A and B to infer that the victim application accessed Set 0

II. Background and Motivation

Modern processors share the last-level cache (LLC) and are vulnerable to attacks whereby a spy can learn the access pattern of a victim application by carefully orchestrating cache evictions. Eviction-based cache attacks can be broadly classified into two categories: Flush-based attacks (e.g. Flush+Reload attack [12]) and Conflict-based attacks (e.g. Prime+Probe attack and Evict+Time attack [13]). Flush-based attacks target accesses to the memory locations that are shared between the attacker and the victim. The attacker flushes a shared line using the *clflush* instruction, waits, and then checks the timing of a later access to that line – if the access incurs shorter latency, then the attacker can infer that the victim application has accessed the line. Flush-based attacks can be mitigated by avoiding the sharing of security-critical data [14] or by restricting the use of clflush to kernel mode [15]. In this work, we focus on mitigating conflict-based attacks and use Prime+Probe attack [13], [16] as a representative example.

A. Conflict-Based Cache Attacks: An Example

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], 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 V) that maps to Set 0, which evicts line B. At a later time, the attacker can access A and B, and measure the time. Given the long-latency now required for B, the attacker can infer that the victim accessed Set 0. Knowing the access pattern of an application can leak secret information [3].

B. Prior Mitigation Approaches

Prior approaches for mitigating conflict-based attacks rely on either preserving victim lines, or on randomized mapping of victim lines to cache locations, as shown in Figure 3.

Examples of preservation based approach include PL-Cache [4] (lock lines of sensitive application in the cache) and Non-Monopolizable Cache [5] (reserve a few ways of the shared cache for each core). Such approaches results in inefficient use of cache space, as cache is reserved for the application/cores regardless of the reuse characteristics of the cache line. Ideally, we seek efficient utilization of cache space.

Examples of randomized mapping includes RPCache [4] and NewCache [8]. These solutions randomize the location of the line (set) in the cache and use a table to keep track of the mapping. For example, the NewCache design [8], shown in Figure 3(b), uses a *Random Mapping Table (RMT)* to track the line-address to cache-address mapping. To avoid attacks on the RMT, the applications must be classified into two categories: protected and unprotected. Each protected application gets a unique RMT-ID which is used to access the RMT. Unprotected applications access the cache directly, without any indirection. The problem with such *Table-Based Randomization* schemes is that the size of the mapping tables must be scaled linearly with the number of lines in the cache and the number of concurrently running protected applications. While such tables may be practical for a small cache (L1), they become impractically large for LLC (e.g.for a 8MB LLC, the mapping table would exceed 1MB). Furthermore, the effectiveness of these schemes is heavily dependent on the ability of the OS to mark applications as protected or unprotected. Ideally, we want to avoid the storage of large tables and have a solution that does not require OS support.

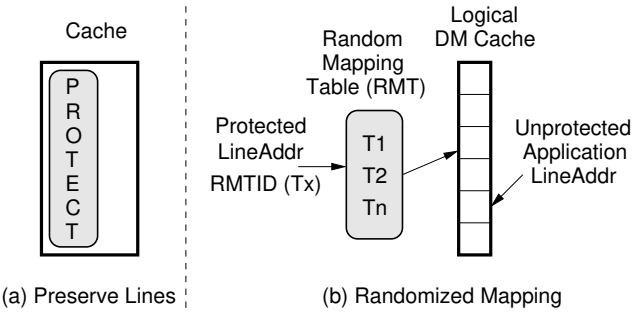

Fig. 3. Prior solutions (a) preservation-based (b) randomization-based

C. Goal and Insight

The goal of this paper is to develop a practical solution to mitigate conflict-based attacks. For a solution to be useful, it is important that it not only provides strong protection against attacks but also has (1) Low performance overheads, (2) Low storage overhead and simple implementation, (3) No reliance on OS or software support, and (4) Localized implementation, which avoids changes to multiple subsystems.

Our paper develops a practical solution based on randomized mapping of lines to cache locations. The key insight in our work is to use encryption to perform randomization of the cache lines efficiently and obviate the need for any indirection table and OS support. We describe our solution next.

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

III. Operating Cache on Encrypted Address Space

Randomized mapping provides protection against conflict-based attacks by making it harder for the attacker to form an eviction set. Our solution leverages encryption to enable randomized mapping for the LLC in an efficient manner. Given the avalanche effect of encryption, lines that map to one set of a traditional cache would get scattered throughout the sets of the cache, and this mapping would be controlled by an encryption key. Our solution, *CEASE (Cache Operated on Encrypted Address Space)* is based on these principles.

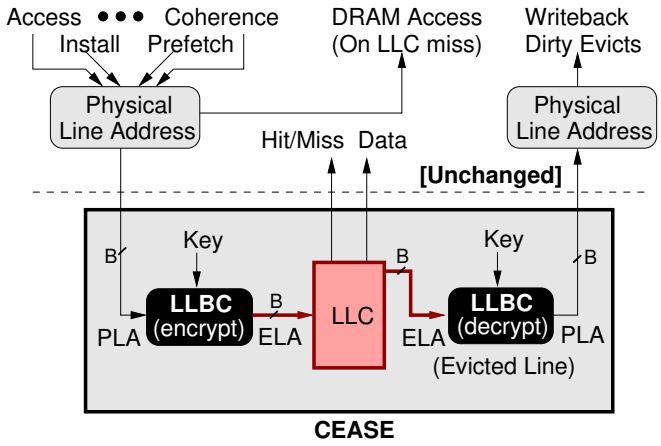

Fig. 4. Overview of CEASE. CEASE uses Low-Latency Block-Cipher (LLBC) to convert between PLA and ELA. The ELA is visible to (and within) the LLC, and all operations outside the cache continue to use PLA.

A. Overview of the CEASE

Figure 4 provides an overview of CEASE. Similar to a conventional cache, CEASE is accessed using the Physical Line-Address (PLA). CEASE uses a *Low-Latency Block-Cipher (LLBC)* to convert the b-bit PLA into a b-bit Encrypted Line-Address (ELA) uses this ELA to access the cache. With CEASE, the cache organization and hit/miss detection all remains unchanged – just that instead of receiving an access for a physical line-address A (which may get mapped to some Set X), the cache receives an access for an encrypted line address B (which may get mapped to a different Set Y). Internally, the tag-store entry of the cache continues to have the usual metadata such as valid bit, dirty bit, replacement state, coherence bits, and the tag. The tag-store and the data-store of the cache remain unchanged.

If the accessed PLA is present in the cache, the cache indicates a hit and provides the data. On a miss, the cache may evict a dirty line, and this line must be written back to the memory. As the tag information in the cache is based on encrypted address, we must first convert it into the original physical address. Fortunately, the same LLBC circuit (with minor changes) can be used to perform decryption and convert the ELA to PLA. As the evicted line now has the original physical address, the line can be written to back to memory without the memory needing to know about the ELA.

The encrypted address (ELA) is visible only to (and is present within) the LLC. The rest of the system remains oblivious to the presence of the ELA and continue to operate in a traditional manner using the PLA. Thus, coherence requests, store requests, prefetch requests, all continue to access the cache with only the PLA and without being aware of the ELA. CEASE internally converts the PLA to ELA to access the cache and re-converts ELA to PLA while interacting with the external systems (such as on writebacks to memory).

B. Considerations for the Block-Cipher

Block ciphers provide a one-to-one mapping from a B-bit plaintext to B-bit ciphertext. The number of bits that we want to encrypt (the line address) is usually quite small. For example, we consider a system with a 46-bit physical address space (capable of addressing up-to 64TB of memory), so the line-address is only 40 bits. We need a block cipher that efficiently converts a 40-bit PLA into a 40-bit ELA. The commonly used encryption algorithms, such as AES, operate at 128-bit to 256-bit granularity, and incur a latency of tens of cycles. We want a block-cipher that operates at low-width, is secure, and does not incur significant latency overheads.

We observe that our usage of block cipher is different in a fundamental way, in that the adversary has no direct visibility to the ciphertext, so the adversary cannot memorize plaintext-ciphertext pairs, which is typically the biggest weakness of small-width block ciphers. For example, small-width block cipher (such as DES, which operates on 64-bits) are usually considered insecure because an adversary can either do a brute force search for the key or memorize plaintext-ciphertext pairs. Therefore, most of the secure block-ciphers (such as AES) have now moved to 128-bit or 256-bit blocks. Fortunately, in our case, the encrypted line-address is not visible to the adversary, so memorization-based attacks are not a concern, and we can use small-width block ciphers. We describe the design of our low-latency block-cipher (LLBC) next.

C. Low-Latency Block-Cipher Using Feistel-Network

One popular method to build block ciphers is the Feistel-Network [11]. Feistel-Networks are simple to implement, incur low-latency, and are widely used in encryption algorithms, such as the Data Encryption Standard (DES) [11] and Blow-Fish [17]. Feistel-Network has been studied extensively and theoretical work has shown that for well-chosen round functions, "having 3 stages is sufficient to make the block cipher a pseudo-random permutation, while 4 stages are sufficient to make it a *strong* pseudo-random permutation" [18]. Therefore, in our solution, we use a four-stage Feistel-Network.

Figure 5 shows the logic for the four-stage Feistel-Network operating on a 40-bit line-address. Each stage splits the 40-bit input into two parts (L and R) and has an output which is split into two as well (L' and R'). R' is equal to L. L' is computed using an XOR operation on R and the output of a *Round Function (F)* which accepts L and a randomly chosen key (K). Each stage requires a 20-bit key, which means the network requires one 80-bit key (one quarter for each stage).

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

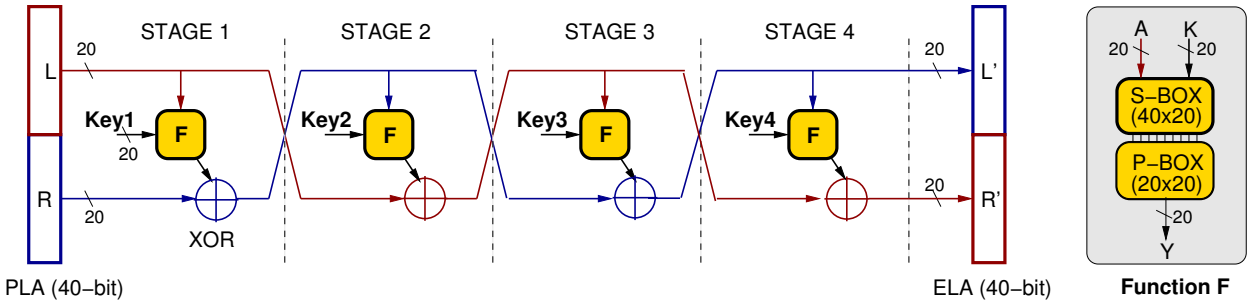

Fig. 5. A Low-Latency Block-Cipher (LLBC) implementation using four-stage Feistel-Network (the F function is based on a Substitution-Permutation Network)

The round function (F) of the Feistel-Network need not be invertible. For the round function (F), we use a substitution-permutation network (SPN) that can provide obfuscation with low-latency and zero storage overheads. We assume that the SPN boxes are configured differently for each stage and they get fixed at design time. The S-Box in our design has 40-bit input and 20-bit output block that is configured such that each bit of the S-Box is computed as XOR of 20 randomly selected inputs. Figure 6(a) and (b) show the logical representation of the S-Box (for 4-bit input and 2-bit output) and the storage-free physical implementation. The P-Box routes one-bit of the input to one (randomly selected) bit of the output, as shown in Figure 6 (c) and (d), for 4-bit input and 4-bit output. The P-Box does not incur any latency due to gate delays.

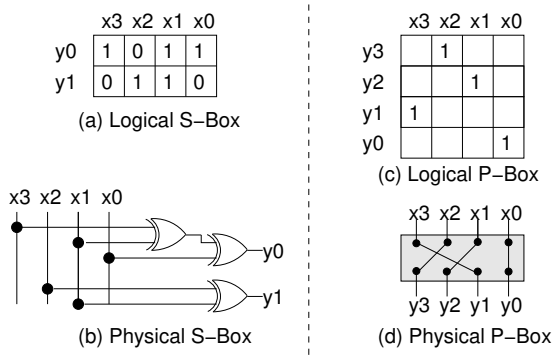

Fig. 6. Components of Substitution-Permutation Network. The (a) logical view and (b) physical implementation of an S-Box. The (c) logical view and (d) physical implementation of a P-Box.

D. Storage and Latency Analysis of Proposed LLBC

Computing each bit of the SPN incurs a delay of 5 two-input XOR gates, which means that each stage of the Feistel-Network has a delay of 6 XOR gates. Thus, for four-stages, the critical path delay will be 24 XOR gates, which can be computed within two clock cycles of even an aggressively pipelined processor (typically the clock cycle of modern processors is designed to perform 15-20 gate operations). The total storage for the proposed LLBC is 80-bits (for the keys).

Note that the same hardware can be used to perform both encryption and decryption, with only minor changes to the sequence in which the operations are performed. So, we do not need two separate LLBC for encryption and decryption (it is shown separately in Figure 4 for simplicity for explanation).

E. Attack Model for the Static Design

CEASE uses an encrypted address to scramble the memory to cache mapping, and this mapping depends on the encryption key. The key is initialized to a random value using a hardware-based PRNG (pseudo-random number generator) when the machine is powered up. So, even if the adversary learns the mapping of one machine, the adversary cannot use it to attack another machine (in fact, mapping learned for a given machine is no longer valid once that machine is restarted). Unfortunately, for a given uptime, the key remains constant, and hence the memory to cache-set mapping is constant. An adversary can use a timing-based attack to learn the mapping.

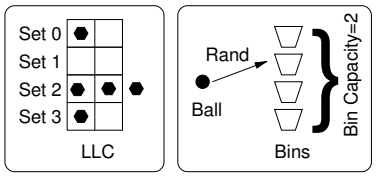

Fig. 7. An attack forms "Eviction Set" for one cache set (a) LLC with 4-sets and 2-ways, (b) Equivalent bin-and-balls model used in our analysis.

Even if the attacker does not know which line maps to which cache set, the attacker can form a group of L random lines and use a timing attack to check if any line of the L lines miss in the cache. A cache with S sets and W ways has a total of $N = S \cdot W$ lines. However, L can be expected to be lower than N, due to the non-uniformity of lines to set mapping inherent in a random assignment (as shown in Figure 7(a), where five random lines are enough to cause an eviction in Set 2).

We analyze the expected value of L using bins and balls analysis, as shown in Figure 7(b). Balls are randomly thrown into the bins until one bin overflows. Our baseline 8MB 16-way cache has 8192 sets, and we found that only 42% of the lines need to be in the attack to cause one eviction. Thus, the attacker can form a group of $L = 0.42 \cdot N$ lines and check for an eviction. However, the attacker still does not know which of the L lines belong to the conflicting set. The attacker can learn this by sequentially removing one line from the L lines and checking if the remaining accesses are eviction-free, if so, the removed line maps to the conflicting set. To learn the eviction set, the attacker needs approximately L^2 accesses [10]. Our baseline 8MB LLC has 128K lines (L=0.42×128K), so it takes an attacker only 22 seconds to learn the eviction set.

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

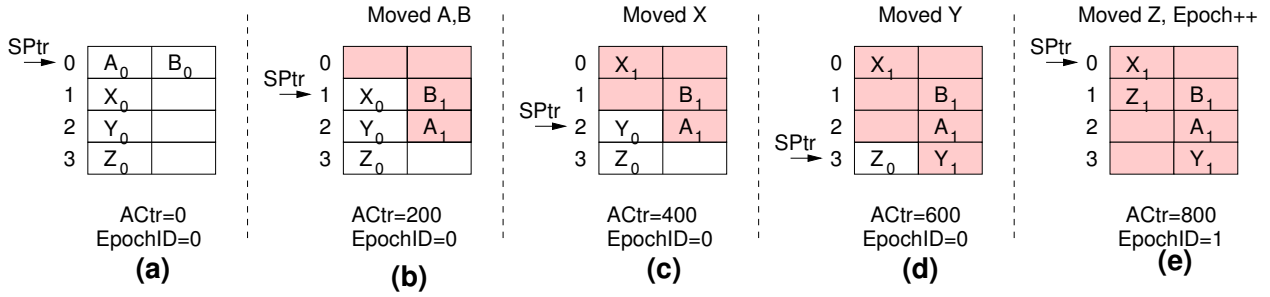

Fig. 8. Example of CEASER with gradual remapping for a cache with four sets (0-3). The subscript with the line denotes the EpochID with which the line was remapped. After every 200 access to the cache (tracked by the access counter, ACtr), all the lines of the set pointed by SPtr gets remapped based on the key of the next Epoch. The shaded area of the cache represents parts of the cache that have undergone remapping based on the key of the next epoch.

IV. CEASER: Dynamic Remapping and Key Change

The weakness of CEASE is that it forms evictions sets at boot time and they remain static throughout, making it possible for an adversary to learn the eviction sets. If we change the lines that form the eviction sets periodically, it will make it much harder for an adversary to learn the eviction sets. Based on this insight, we propose *CEASE with Dynamic-Remapping (CEASER)*, which accomplishes this by periodically changing the key and remapping the lines based on the new key.

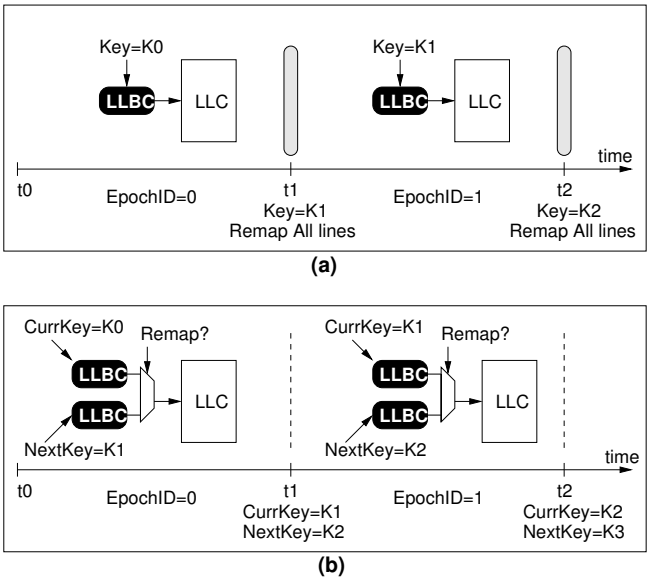

Fig. 9. Dynamic-Remapping in CEASER (a) Bulk-Remap (b) Gradual-Remap

A. Overview of Dynamic-Remapping in CEASER

CEASER divides the time into epochs and each epoch has its own key. In each epoch, CEASER uses the key of the epoch to perform encryption and decryption. Figure 9 describes two designs for CEASER: (a) bulk-remapping and (b) gradual-remapping. In the bulk-remapping design, at the end of the epoch, the keys are changed and all the cache lines are remapped using the new-key (to perform remapping, the line is read, the encrypted line address is converted into physical address with the old key, this physical address is re-encrypted with the new key and installed in the cache, and the old location is invalidated). This design is impractical because it requires simultaneous remapping of all the lines.

The second design gradually remaps parts of the cache during the entire period using the key (NextKey) corresponding to the next epoch. By the time the epoch ends, all the lines in the cache get remapped using the NextKey, so no additional remapping is required at the end of the epoch. At the end of the epoch, the current key (CurrKey) is set to the NextKey and the NextKey is initialized to a random value. On each access, the incoming PLA is translated into an ELA using both the CurrKey and NextKey, and the appropriate one is chosen depending on whether that line has already been remapped using the NextKey. As the gradual-remap based design spreads the remapping throughout the epoch, it is a more practical design. We use gradual-remap as the default design for implementation of CEASER.

B. Practical CEASER via Gradual-Remapping

The rate of remapping of the CEASER is regulated by a parameter *Accesses-Per-Line-Remap (APLR)*. For practical reasons, CEASER performs remapping of all lines in the set at any given time. So, if we set APLR=K, CEASER would remap one set containing W ways every $W \cdot K$ accesses. The entire cache containing N lines will get remapped after $K \cdot N$ accesses, which is also the epoch period.

To perform remapping, CEASER is equipped with two registers: *Set-Relocation Pointer (SPtr)* and *Access-Counter (ACtr)*. The SPtr keeps track of *which* set in the cache that should be remapped next. ACtr is incremented on each access to the cache and determines *when* to trigger the next remap.

Figure 8 shows an example of gradual mapping for a cache containing 4 sets (0-3). For simplicity, we assume five lines are resident in the cache (A, B, X, Y, Z). The subscript with the line denotes the encrypted address of the line based on the EpochID (so, A_0 is the encrypted address of A with the key of EpochID=0, and A_1 is the encrypted address of A based on the key of EpochID=1). The cache begins with SPtr=0, ACtr=0, and EpochID=0, as shown in Figure 8(a). We assume CEASER is set to remap one line every 100 accesses, so one set of the two-way cache will get remapped every 200 accesses.

After 200 accesses to the cache (ACtr=200), the lines resident in Set 0 (A_0, and B_0) get remapped. CEASER performs decryption of the line address (A_0 to A) using the CurrKey, re-encrypts the line with the key (NextKey) of the next epoch (A to A_1), and installs this line (A_1) in the cache using the

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

regular replacement policy. A1 goes to Set 2 and B1 goes to Set 1. The SPtr is incremented as shown in Figure 8(b).

After another 200 accesses (ACtr=400), Set 1 undergoes relocation. Note that we need to relocate only X_0 and not B_1, as B_1 corresponds to the next epoch. After remapping, X_1 gets relocated to Set 0, as shown in Figure 8(c).

After another 200 accesses (ACtr=600), Set 2 undergoes relocation and line Y_0 gets remapped, as shown in Figure 8(d). After another 200 accesses (ACtr=800), Set 3 undergoes relocation and line Z_0 gets relocated to Set 1 after getting remapped with NextKey. Note that now the entire cache has undergone remapping and contains lines that belong only to the next epoch. When this happens, the EpochID is incremented, Sptr and ACtr are reset, CurrKey is set to NextKey, NextKey is set to a random value, and the process repeats.

With gradual mapping, lines from two different epoch (current and next) can co-reside in the cache at any given time. Therefore, the encrypted line address must be identified with the EpochID (EID) with which the line was remapped. Otherwise, two lines in a set can end up having the same encrypted address, one for the current epoch and one for the next epoch. For example, in Figure 8(b), if A1 equals Y0, and Y0 has not been remapped yet, we will have two lines with an identical encrypted address. However, if we also stored EID with each cache line, we can identify precisely which encrypted address belongs to which line. As only two EpochIDs can be valid at any given time (current or next), we need only one-bit (EID) to identify the EpochID of the line.

We use a default value of APLR=100 in our design, which means the remapping overhead is limited 1% (on average, one line gets remapped per 100 accesses). Our analysis in Section V will show that this rate of remapping is sufficient to tolerate 100+ years of continuous attack.

C. Cache Access with CEASER

On an access to the LLC, we do not know if the line has been already remapped during this epoch or not. However, we can determine this by first performing the encryption using the CurrKey and checking if the set index of the line maps to a set that is yet to undergo remapping (set index is greater than or equal to SPtr). If so, we should use the mapping using the Currkey, otherwise use the mapping using the NextKey, as shown in Figure 10. Once the encrypted address and the corresponding EpochID (current or next) is available, the cache access/install can proceed in the normal manner – with the tag match on the encrypted line address (plus the EID).

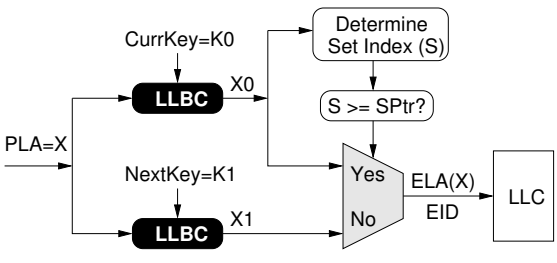

Fig. 10. Translating PLA to ELA using CEASER. Encryption is performed with both CurrKey and NextKey and the correct ELA to access the LLC.

D. Analyzing CEASER for a Banked Shared-Cache

In our studies, we assume that there is no timing difference in access to different parts of the LLC. Some processor designs use a banked LLC, where the banks are connected via a ring, and access to different banks have different latencies. An adversary can exploit this timing differences to determine the line to bank mapping and focus the attack on one bank [10].

Our bank-agnostic implementation of CEASER would determine the line-to-bank mapping based on the encryption key. However, if the designer wants the flexibility to regulate the line-to-bank mapping (using a proprietary function), then CEASER can still be implemented at a per-bank granularity, with each bank having its own remapping circuit. In both the *bank-agnostic* and *bank-aware* implementations of CEASER, the effective size of the cache under attack would get reduced to one bank. For example, banking would degenerate our 8MB LLC into a 1MB LLC for the attacker. Fortunately, per the analysis shown in Section V (Table I), for a 1MB 16-way cache, remapping one line every 100 accesses is still sufficient to provide strong security even for the banked design.

V. Security Analysis of CEASER

For CEASER, the rate of remapping affects the time that the attacker has to learn about the mapping before the mapping changes. For security, we want the remapping to happen frequently, and to reduce the overheads of relocation, we want the remapping to happen infrequently. The rate of remapping is controlled by APLR (Accesses-Per-Line-Remap), which we denote as K. We use a default value of K=100 in our design, which means the remapping overhead is limited to 1% (one line gets remapped per 100 accesses). We analyze if this rate of remapping is sufficient for strong security.

A. Attack Model and Assumptions

We make the following (severely conservative) assumptions:

1) Simply knowing the eviction set of one of the cache-set is sufficient to declare the attack successful (even though this may not be the same set shared by the victim).
2) The lines in the attack do not get remapped during the entire epoch (the gradual move of CEASER would cause some attack to get relocated but to simplify the analysis we make the assumption of bulk remapping).
3) No other application accesses the cache during the epoch period (otherwise, those accesses can cause the epoch to end earlier and the attacker will be able to do only a reduced fraction of total accesses in the epoch).
4) The adversary has enough time to not only form an eviction set but also to infer the information of the victim (if most of the epoch goes in forming an eviction set, it would leave very little time to attack the victim).

We also assume an idealized encryption algorithm which can randomly map a given line to any arbitrary location in memory with an equal probability (in Appendix-A, we perform an *Avalanche Test* on our proposed LLBC to demonstrate that even a 1-bit difference in the physical address space causes a large number of bits to change in the encrypted address space).

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

B. Analytical Model for Attack

Let there be N lines in the cache, organized as S sets and W ways. Let CEASER be set to remap one line every K accesses to the cache, so the epoch period will be $K \cdot N$ accesses. To attack the cache, the adversary can form a group of L random lines, and test if any of these lines miss in the cache. Using the bins-and-balls analysis, we can estimate the fraction of the total cache lines (F) that must be attacked ($L = F \cdot N$) to encounter one of the L lines that misses (similar to Figure 7). Once the attacker forms a group of lines, at least one of which missed in the cache, the attacker can remove one line and test if the miss still happens. If the miss did not happen, then the removed line is part of the eviction set, otherwise, it is not. This process is repeated for all L lines to learn the eviction set of one of the (unknown) set of the cache. At each step, the timing test is done at least R times to get a reliable estimate. To launch this attack the adversary would need $R \cdot L^2$ accesses. The attack would not succeed if all the lines in the attack are guaranteed to get remapped before the adversary can perform $R \cdot L^2$ accesses. So, we can bound the epoch period as follows:

$$Epoch\_Period < R \cdot L^2 \tag{1}$$

The number of lines (L) in the attack represents a fraction (F) of the total cache lines. So,

$$K \cdot N < R \cdot (F \cdot N)^2 \tag{2}$$

Therefore,

$$K < R \cdot F^2 \cdot N \tag{3}$$

To estimate F, we again use the bins-and-balls analysis. We leverage the observation the number of balls in a given bin follow a Poisson process, when the balls are thrown at random. In particular, if the average number of balls per bin is λ, then the probability that a given bin has B balls is given by:

$$P(bin\_has\_B\_balls) = \frac{e^{-\lambda} \cdot \lambda^B}{B!} \tag{4}$$

We are interested in a set having more lines mapped to it than the associativity of the cache (for our 16-way LLC, we are interested in the set having 17 or more lines). We can use Equation 4 to determine the average number of ways that must be full for one of the set to overflow (with a given probability), and use this information to determine the fraction F.

C. Time for Successful Attack

The resilience of CEASER to attacks is dictated by the rate of remapping – the quicker the remap, the less time the adversary has to learn the mapping. Table I shows the time for a successful attack, as the rate of remapping of CEASER is changed from 100 (default) to 2000. We perform this analysis for both the baseline 8MB cache and a 1MB bank (attacker focuses on 1 bank, CEASER implemented per bank). Furthermore, we conservatively assume R=2 (only two trials are needed to get an accurate estimate of the timing, instead of the tens/hundreds that are typically required).

TABLE I
TIME FOR ATTACK TO SUCCEED WITH CEASER.

Accesses-Per-Line-Remap	8MB LLC	1 MB Bank of LLC
100 (default)	> 100 years	> 100 years
200	> 100 years	21 years
500	> 100 years	16 days
1000	> 100 years	5 hours
2000	37 years	5.2 minutes
No-Remap (CEASE)	22 seconds	0.4 seconds

Thus, remapping is crucial for the security of CEASER. Without remapping, the attack succeeds in less than one minute (and the exposed lines continue to remain vulnerable). With remapping, the attacker needs 100+ years to expose one set (and that too only for a few microseconds). Furthermore, CEASER is even more effective for larger caches because the number of accesses required to learn the eviction set increases approximately in proportion to N^2 [10] (N is the number of lines in the cache), whereas the time to remap the entire cache increases only linearly with N.

VI. EXPERIMENTAL METHODOLOGY

A. Configuration

We use a Pin-based x86 simulator with a detailed memory model. Table II 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 CEASER, we encryption latency of 2 cycles and APLR=100.

TABLE II
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
DRAM Memory-System	
Bus frequency	800 MHz (DDR 1.6 GHz)
Channels	2 (8-Banks each, 2KB row buffers)
tCAS-tRCD-tRP-tRAS	9-9-9-36

B. Workloads

We use a diverse set of workloads for our study, including all the 29 workloads from the SPEC2006 benchmarks suite and 5 workloads from the GAP benchmark suite [19]. For each benchmark, we use a representative slice [20] of 1 billion instructions. These 34 benchmarks are run in rate-mode where each core runs a copy of the benchmarks. Additionally, we use 100 mixes formed randomly from these 34 benchmarks.

We perform timing simulation until all benchmarks in the workload finish executing a minimum of 1 billion instructions. For measuring aggregate performance, we use the weighted speedup metric. We report normalized performance as the ratio of weighted speedup of the proposed design to the baseline.

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

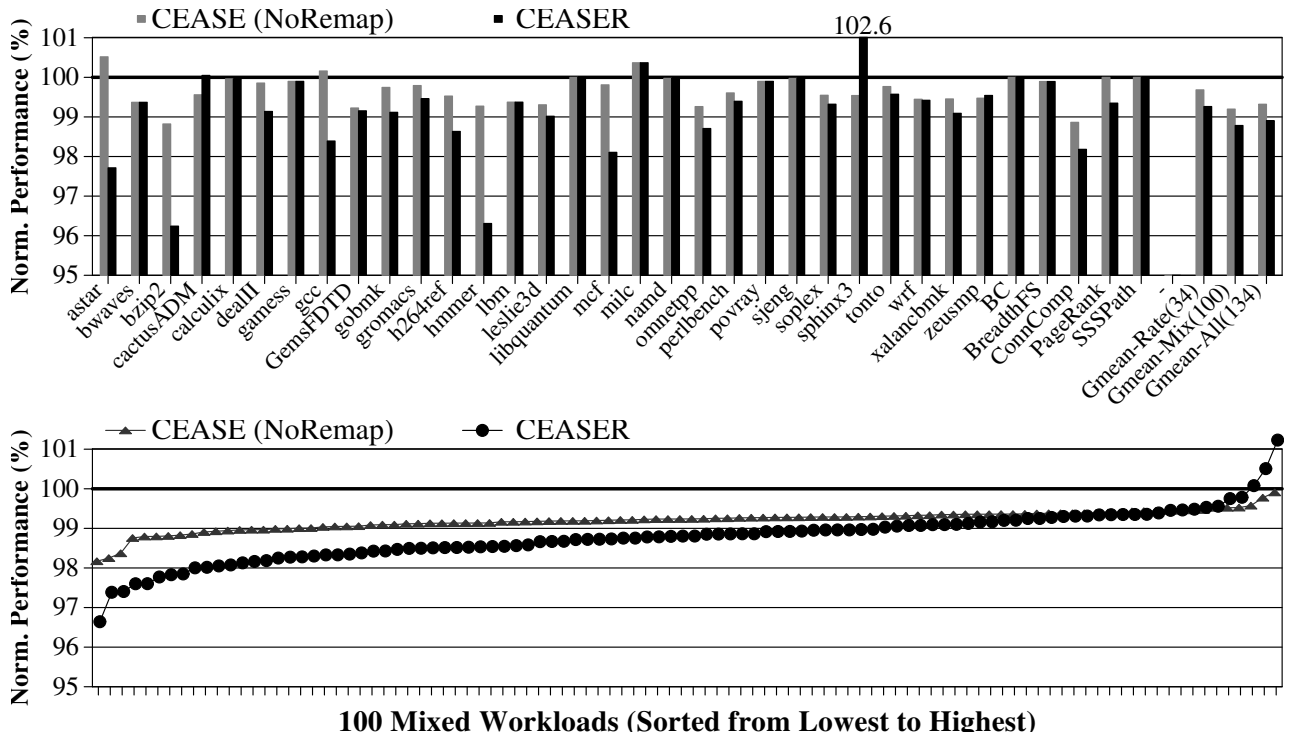

Fig. 11. Normalized performance of CEASE (no remapping) and CEASER for the rate-mode workloads (top) and mixed-workloads (bottom). For the 34 rate-mode workloads, CEASE and CEASER incur an average performance loss of 0.32% and 0.74% respectively. Across all 134 workloads the average performance loss is 0.68% and 1.1% respectively.

VII. Results and Analysis

A. Impact on Performance

There are three sources of performance impact of CEASER. First, potential change in cache miss-rate because of the change in the group of lines that get mapped to a given set of the cache, which alters the conflict misses (this can sometimes improve hit rate and sometimes degrade hit rate, although the overall impact is negligible). Second, the increase in access latency of the cache due to the latency of the encryption process (translating the physical line-address into an encrypted line-address). And, third, extra misses incurred due to the relocation of lines as such relocated lines can evict a useful line from the set where the line got remapped to (albeit by design, such relocations are done at a rate of 1 every 100 accesses, so the impact on miss rate due to relocation is bounded to 1%).

Figure 11 shows the performance of CEASE (design without dynamic remapping) and CEASER (with dynamic remapping) for 34-rate workloads and 100 mixed workloads. Note that the performance is normalized to the baseline (so higher is better, and 100% denotes no degradation). For the rate-mode workloads, CEASE and CEASER cause an average performance loss of 0.32% and 0.74%, respectively. Sphinx3 sees a 2% improvement because of the reduction in conflict misses. Overall, across all 134 workloads, CEASE and CEASER incur a slowdown of only 0.68% and 1.1% respectively.

B. Sensitivity to Encryption Latency

CEASER utilizes low-latency block-cipher (LLBC) to perform encryption and decryption. The algorithm is chosen to minimize the latency of such translations, while exploiting the fact that the adversary does not have a direct access to the encrypted value. We use a latency of 2 cycles for encryption/decryption, based on the estimated logic delays.

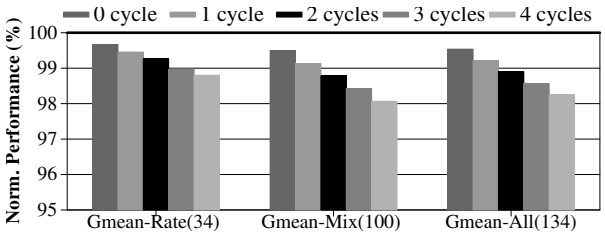

Fig. 12. Impact of the latency of encryption on the performance of CEASER.

Figure 12 shows the performance of CEASER as the latency of encryption is changed from 0 cycles to 4 cycles. With a zero-cycle latency overhead, we could reduce the performance loss of CEASER to 0.5%, whereas with a 4-cycle latency, the performance loss would be 1.75%. Thus, the performance overhead can still be reduced further using lower latency implementations of block ciphers.

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

C. Impact of CEASER on LLC MPKI and Miss-Rate

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 on the miss-rate and MPKI (misses per 1000 instructions) of the LLC. Table III shows the miss rate and MPKI of the baseline 8MB LLC and with CEASER, for the 34 rate-mode workloads.

TABLE III
IMPACT OF CEASER ON THE MPKI AND MISS-RATE OF LLC (8MB).

Workload Name	MPKI Baseline	MPKI CEASER	MissRate(%) Baseline	MissRate CEASER
astar	0.5	0.7	21.5	27.8
bwaves	18.7	18.7	100	100
bzip2	3.5	3.6	52.6	56.8
cactusADM	5.3	5.2	97.7	96.6
calculix	0.0	0.0	68.5	68.7
dealII	2.5	2.4	65.7	66.4
gamess	0.0	0.0	3.4	3.8
gcc	16.3	16.7	80.9	82.8
GemsFDTD	9.8	9.8	92.8	93.2
gobmk	0.4	0.4	28.2	33.3
gromacs	0.6	0.6	35.6	37.5
h264ref	0.5	0.6	35	42.6
hmmer	0.5	0.8	17.5	28
lbm	31.9	31.9	99.9	99.8
leslie3d	7.6	7.6	91.5	92.1
libquantum	25.4	25.4	100	100
mcf	67.6	68.8	72.4	73.8
milc	25.8	25.6	99.8	99
namd	0.1	0.1	72	73
omnetpp	20.9	21.1	87.7	88.2
perlbench	0.8	0.8	46.7	48
povray	0.0	0.0	2.2	2.2
sjeng	0.4	0.4	87.3	87.5
soplex	26.9	26.9	97.7	97.7
sphinx3	11.6	11.1	89.4	85.5
tonto	0.1	0.1	4.9	7
wrf	6.6	6.5	97	96.8
xalancbmk	2.2	2.2	68.9	69.9
zeusmp	4.8	4.8	96.4	96.3
BC	84.5	84.5	98	98.1
BreadthFS	37.2	37.3	96.7	96.8
ConnComp	85.7	85.9	95.7	96
PageRank	46.0	46.2	94.6	94.9
SSSPath	118.8	119.0	98	98.1
Average	19.5	19.6	70.5	71.7

Some low-MPKI workloads (such as astar and gobmk) see a noticeable increase in MPKI, however, given these workloads are low-MPKI, the impact on performance remains negligible. For sphinx3, we notice that randomization causes the MPKI to get reduced from 11.6 in the baseline to 11.1 with CEASER, and this 5% reduction in MPKI causes improved performance of CEASER compared to the baseline. Overall, CEASER has only a negligible impact on the missrate of the LLC, increasing it from 70.5% to 71.7%, and also a negligible impact on MPKI, increasing the average MPKI from 19.5 to 19.6.

D. Sensitivity to Remapping Interval

The rate of remapping of CEASER is regulated by the parameter APLR (Accesses-Per-Line-Remap). In our default implementation of CEASER, we use an APLR=100, which means one cache line gets remapped every 100 accesses to the cache (for simplicity, we remap one set of a W-way cache every 100*W accesses). We showed that this rate is sufficient to provide strong security and that the performance overhead with this rate of remapping is quite small (almost 1%).

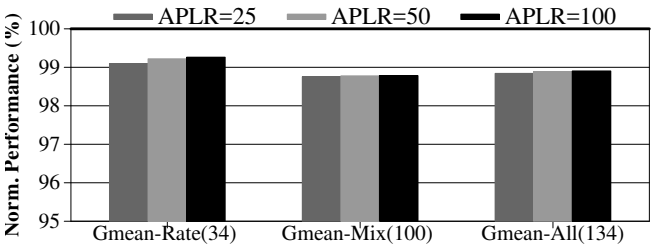

Fig. 13. Impact of the remapping frequency on slowdown of CEASER. Increasing remap frequency by 4x increases slowdown from 1.1% to 1.2%

Figure 13 shows the performance of CEASER normalized to the baseline, for an APLR of 25, 50, and 100 accesses. The impact of reducing the APLR is quite small, as performance loss of CEASER increases from 1.1% to 1.2% if the APLR is reduced from 100 to 25. So, a designer may choose to reduce APLR somewhat and still not incur significant slowdown.

E. Sensitivity to LLC Capacity

Our default configuration contains an 8MB shared LLC. In this section, we study the performance sensitivity of CEASER for different sizes of LLC. Figure 14 shows the normalized performance of CEASER when 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 is between 1% and 1.5%. Thus, CEASER incurs negligible slowdown even for large LLCs.

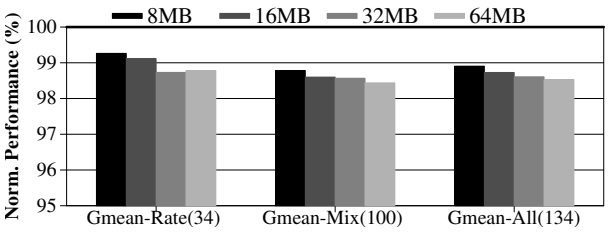

Fig. 14. Performance Impact of CEASER for systems with larger LLC.

F. Analyzing Other Overheads: Logic, Storage, and Power

Logic: CEASER requires two LLBC circuits and selection logic to select between these two circuits. Each LLBC with our proposed design requires approximately 1680 two-input xor gates, and the selection logic requires approximately 100 two-input gates. So, the total logic overheads of CEASER is less than 3500 two-input gates, which is quite small (similar to computing SECDED code for a 64-byte line).

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

Storage for New Structures: The storage overhead of the LLBC is quite small (80 bits for the keys). CEASER needs two such keys. CEASER also requires a 13-bit SPtr pointer (for an 8MB 16-way cache), a 13-bit Access Counter (ACtr) and a 1-bit EpochID. The total storage overhead for the newly added structures of CEASER is less than 24 bytes.

Per-Line Storage: CEASER also requires that each cache line be appended with a 1-bit EID to denote the EpochID whose key was used for encrypting the line address.

Energy and Power Overheads: The power overheads of CEASER comes from the encryption circuit, the remapping of lines (one line per 100 access), and the extra misses (0.4% higher memory traffic). With CEASER, the overall system power increases by approximately 0.2% and the overall system energy by less than 1% (mainly due to the slight slowdown).

VIII. Related Work

Our paper focuses on developing efficient mitigation solutions for conflict-based attacks. There are several prior works that have investigated this problem. In this section, we discuss the prior work, comparing and contrasting where appropriate.

A. Preservation-Based Mitigation Techniques

Preservation-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). For example, **PLCache** [4] uses cache line locking to lock the lines of the protected applications. **CATalyst** [21] uses the Cache Allocation Technology (CAT) to reserve a given number of ways for the security-sensitive applications. And, **StealthMem** [22] uses page-coloring to ensure that sensitive data gets mapped to cache sets that do not receive contention from other applications. The efficacy of all these schemes rely on the ability of the OS to carefully classify applications into protected and unprotected groups, and use the cache preservation only for the protected groups. Ideally, we want to mitigate attacks without relying on any OS or software support. Furthermore, these solutions do not use cache space efficiently, as cache resources are used for the protected applications, regardless of the reuse.

Non-Monopolizable (NoMo) Cache [5] allocates a fixed number of ways to each core, and performs dynamic partitioning only on the remaining ways. While NoMo can protect victim lines from getting evicted, NoMo becomes impractical for the LLC, given that the LLC is shared by a large number of cores. For example, our baseline 16-way LLC is shared by 8-cores, and reserving even 1-way for each of the core would make half of the LLC unavailable for dynamic partitioning, leading to inefficient use of the LLC resources. Nonetheless, an SMT-based system can still use NoMo for the L1 and L2 cache (private) and CEASER for the LLC.

Recent studies [15], [23] have exploited the inclusion property of LLC to provide preservation. For example, **Relaxed Inclusion Cache (RIC)** [23] classifies the working set into private, read-only, and critical pages and relaxes inclusion for a subset of the working set. RIC requires OS support for classification, changing the way how coherence requests are handled (snoop filters), and requires cache flush on thread migration. Ideally, we want to avoid these changes and the need for OS support. **SHARP** [15] avoids selecting a replacement victim in the LLC that is present in the L1 or L2, and when this happens it tries to select another victim. After a certain number of such trails, if a victim is not found then a random victim is chosen. SHARP counts such episode of random victim with an *AlarmCounter*, and informs the OS of a possible attack if the AlarmCounter exceeds a certain threshold. Thus, SHARP requires (a) changes to the on-chip network to send query between the LLC and the L1/L2 for replacement victim, and (b) OS support to handle the AlarmCounter (and a policy for false positives). Ideally, we want to mitigate attacks without requiring any OS support.

B. Randomization-Based Mitigation Techniques

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 [4], [8] use mapping tables to track the location of the line (or the set) in the cache. For example, **RPCache** [4] moves all lines belonging to a protected application from one set to another randomly selected set and uses a *Permutation Table (PT)* to remember the set-to-set mapping. **NewCache** [8] performs randomized mapping on a per-line granularity, whereby a line is mapped to a set in the direct-mapped cache and a *Random Mapping Table (RMT)* is used to track the line-address to cache-location mapping.

The disadvantage of such table-based randomization schemes is that the mapping tables must scale linearly with the number of lines in the cache and the number of concurrently running protected applications. While such tables can be implemented efficiently for small L1 caches [8], 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 (otherwise, the mapping table can be attacked). Table IV shows the storage overheads of table-based randomization (TBR), with OS support (shared mapping tables) and without OS support (private per-core mapping tables), and CEASER.

TABLE IV
Storage Overhead of Additional Structures (for 8MB LLC).

Scheme	Storage Overheads
Table-Based (with OS support)	1.25 megabytes
Table-Based (without OS support)	8.5 megabytes
CEASER (OS support not required)	24 bytes

TBR requires mapping tables exceeding 1MB (with OS support) or exceeding the capacity of the LLC (if no OS support is provided). CEASER, not only incurs negligible storage but also avoids the latency of looking up large mapping tables. Thus, CEASER has lower storage, better performance, and no reliance on OS support, which makes it feasible to incorporate randomization in large LLCs.

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

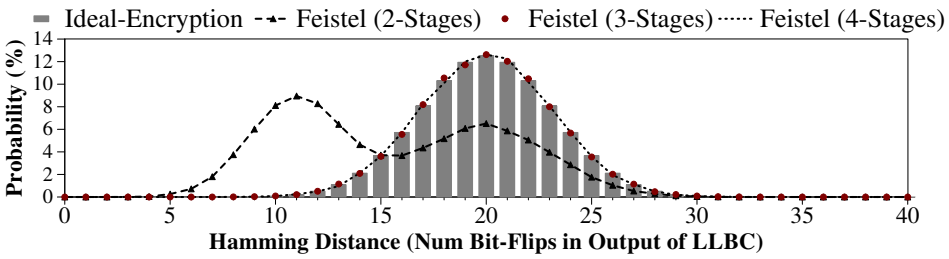

Fig. 15. Avalanche Test for demonstrating the diffusion property of the proposed LLBC. We flip a random bit of the physical line address and measure the hamming distance change in the ciphertext of LLBC. Feistel-Network with 3 stages (red circle) or 4 stages (dotted-line) provide bit-flips close to an ideal encryption (bar graph). However, having only two stages in the Feistel-Network is insufficient to provide close-to-ideal bit-flips for the Avalanche Test.

C. Software-Based Mitigation Techniques

Cache attacks can be mitigated by rewriting the security-sensitive applications to not keep critical information (such as encryption tables) in memory and instead compute the critical information on-the-fly [13], [24]. Unfortunately, such implementations tend to be 2x to 4x slowdown [24]. Our solution avoids the effort in identifying critical pieces, rewriting the software, and the slowdown of such methods.

D. Randomized Addressing for Cache and Memory

Prior studies [25]–[28] have used XOR-based indexing functions for reducing conflicts in cache-sets and memory-banks. However, such functions can be learned by an adversary using timing-based attacks. Once the attacker obtains the mapping for one machine, the attacker can use the information to attack other machines that use the same mapping functions.

Start-Gap Wear-Leveling [29], [30] of non-volatile memories used Feistel-Network for static randomization of the entire memory space, however, with Start-Gap the keys for the Feistel-Network remains unchanged during the system uptime.

IX. Conclusion

To mitigate conflict-based cache attacks, this paper proposes an efficient design that randomizes the memory line to cache location mapping without relying on storage-intensive indirection tables or OS support. Our solution, CEASER, accesses the cache with an encrypted line-address and performs periodic remapping of the address space to limit the time the adversary has to learn the mapping. CEASER provides strong security (tolerates 100+ years of continuous attack), has low performance overhead (1% slowdown), requires a storage overhead of less than 24 bytes for the newly added structures, and does not need any support from the OS/software. While we analyzed CEASER only for a shared LLC, CEASER can also be used to protect against attacks on other shared structures, such as against directory-based attacks [31]. Exploring such extensions are a part of our future work.

Acknowledgments

Special thanks to Gururaj Saileshwar for discussions on the Attack Model. We also thank Mohammad Arjomand, Poulami Das, Sanjay Kariyappa and the anonymous reviewers of MICRO-2018 for their feedback and suggestions.

Appendix-A: Avalanche Test for LLBC

Claude Shannon identified *confusion* and *diffusion* as the two vital properties for a secure cipher [32]. Confusion is the property that the ciphertext should depend on several bits of the key, thus obscuring the connections between the two. In our proposed implementation of LLBC, each bit of the SPN is dependent on half of the bits of the respective key, thus satisfying the property of confusion. Diffusion is the property that the change of even a single bit of the plaintext should (statistically) change half of the bits in the ciphertext.[1]

In this section, we demonstrate the diffusion property for our proposed implementation of LLBC, using an *Avalanche Test* [33], [34]. We take a random 40-bit PLA (P1) and compute the encrypted output (E1). Then, we randomly flip one bit of the PLA (P2) and recompute the encrypted output (E2). We compute the hamming distance between the encrypted outputs (E1 and E2). For an ideal encryption algorithm, we expect close to half of the bits of the encrypted output to change.

To compute the hamming distance for an ideal encryption, we model each bit flip as a Bernoulli random variable with probability p = 0.5 across 40 bits. For our proposed LLBC, we perform the Avalanche Test using 1 million Monte-Carlo trials (1000 different random configuration of the LLBC are used and for each such design we perform the test for 1000 random values). Figure 15 shows the percentage of times the number of bits in the encrypted value changed by a given amount (from 0 to 40 bits). The bar graph shows the variation expected for an ideal encryption. We show the data for Feistel-Network of Figure 5 with 2-stages, 3-stages, and 4-stages. With a 2-stage Feistel-Network, the number of hamming distance change in the encrypted output is less than ideal (10-12 bits flip instead of 18-22 bits). However, with a 3-stage or a 4-stage Feistel-Network, we get diffusion close to ideal encryption. Thus, Feistel-Network with 3-stages (delay of 18 gates, or only 1 cycle) or 4-stages (delay of 24 gates, or 2 cycles) are sufficient for providing strong diffusion. Nonetheless, CEASER can be implemented with alternative designs of block-ciphers as well.

[1]Diffusion causes the lines that are conflicting in the Physical Address Space (PAS) to get scattered randomly in the Encrypted Address Space (EAS). For example, consider an alternative circuit that computes EAS=XOR(PAS, Key). This circuit does not provide a good amount of diffusion as one bit-flip of PAS causes exactly one bit-flip in the EAS. If two PAS lines A and B form eviction set on say Cache-Set-X, then with such a circuit, (A XOR Key) and (B XOR Key) will still form an eviction set (on some other Cache-Set-Y).

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] 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.

[2] 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.

[3] D. J. Bernstein, "Cache-timing attacks on AES," Tech. Rep., 2005.

[4] 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.

[5] 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.

[6] J. Kong, O. Aciicmez, J. P. Seifert, and H. Zhou, "Architecting against software cache-based side-channel attacks," *IEEE Transactions on Computers*, vol. 62, July 2013.

[7] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," in *IACR Eprint archive*, 2005.

[8] 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)*, 2008.

[9] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Mapping the intel last-level cache," in *IACR Cryptology ePrint Archive*, 2015.

[10] 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*, May 2015.

[11] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. CRC Press, Inc., 1996.

[12] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on aes to practice," in *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[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] F. Liu and R. B. Lee, "Random fill cache architecture," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

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

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

[17] B. Schneier, "Description of a new variable-length key, 64-bit block cipher (blowfish)," in *Fast Software Encryption, Cambridge Security Workshop*, 1994.

[18] M. Luby and C. Rackoff, "How to construct pseudorandom permutations from pseudorandom functions," *SIAM J. Comput.*, vol. 17, 1988.

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

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

[21] 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.

[22] 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.

[23] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. B. Abu-Ghazaleh, D. V. Ponomarev, and A. Jaleel, "Ric: Relaxed inclusion caches for mitigating llc side-channel attacks," *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017.

[24] "Software mitigations to hedge AES against cache-based software side channel vulnerabilities," in *IACR Eprint archive*.

[25] A. González, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through xor-based placement functions," in *International Conference on Supercomputing (ICS)*, ser. ICS '97, 1997.

[26] H. Vandierendonck and K. D. Bosschere, "Xor-based hash functions," *IEEE Transactions on Computers*, vol. 54, 2005.

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

[28] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling ways and associativity," in *International Symposium on Microarchitecture (MICRO)*, 2010.

[29] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *International Symposium on Microarchitecture (MICRO)*, 2009.

[30] M. K. Qureshi, A. Seznec, L. A. Lastras, and M. M. Franceschini, "Practical and secure pcm systems by online detection of malicious write streams," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[31] 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.

[32] C. E. Shannon, "Communication theory of secrecy systems," *The Bell System Technical Journal*, vol. 28, 1949.

[33] H. Feistel, "Cryptography and computer privacy," *Scientific American*, vol. 228, 1973.

[34] "On the design of s-boxes," in *Advances in Cryptology CRYPTO '85 Proceedings*, ser. Lecture Notes in Computer Science, H. Williams, Ed., 1986, vol. 218, pp. 523–534.

Authorized licensed use limited to: University of Michigan Library. Downloaded on May 27,2020 at 15:10:03 UTC from IEEE Xplore. Restrictions apply.