Randomizing Set-Associative Caches Against Conflict-Based Cache Side-Channel Attacks

Wei Song, *Senior Member, IEEE*, Zihan Xue, Jinchi Han, Zhenzhen Li, and Peng Liu, *Member, IEEE*

Abstract—Conflict-based cache side-channel attacks against the last-level cache (LLC) is a widely exploited method for information leaking. Cache randomization has recently been accepted as a promising defense. Most of recent designs randomize skewed caches rather than classic set-associative caches; however, skewed caches incur substantial performance overhead both in area and runtime. We cautiously argue that randomized set-associative caches can be sufficiently strengthened and possess a better chance to be adopted in the near future. For the first time, a dynamically randomized set-associative cache has been implemented in the LLC of a Linux capable multicore processor. A single-cycle hash logic is designed for randomizing the cache set indices. A multi-step relocation scheme is used to reduce the cost in remapping the cache layout. The randomized cache layout is remapped periodically for limiting the time window available to attackers. An attack detector is implemented to catch attacks in action and consequently trigger extra remaps. The evaluation results show that the randomized LLC has been sufficiently strengthened to thwart all existing fast algorithms for searching eviction sets with only marginal runtime overhead, and small area and power overhead.

Index Terms—Conflict-based cache side-channel, cache randomization, attack detection, computer micro-architecture.

1 Introduction

Conflict-based cache side-channel attacks against the last-level cache (LLC) [1] is a widely exploited method for information leaking. Since the LLC is shared between all processing cores, it allows a malicious software to trigger controlled conflicts, such as evicting a specific cache set with attackers' data [2], to infer security-critical information of a victim program. They have been utilized to recover cryptographic keys [3], break the sandbox defense [4], inject faults directly into the DRAM [5], and extract information from the supposedly secure SGX enclaves [6].

Cache randomization [7]–[11] has recently been accepted as a promising defense. In a randomized cache, the mapping from memory addresses to cache set indices is randomized, forcing attackers to slowly find eviction sets at runtime [2], [12], [13] rather than directly calculating cache set indices. Even when eviction sets are found, attackers cannot easily tell which cache sets are evicted by them. However, cache randomization alone does not defeat conflict-based cache side-channel attacks but only increases difficulty and latency [7], [14]. For this reason, dynamic remapping [7], [10] is used to limit the time window available to attackers, and cache skews [8] have been introduced to further increase the difficulty in finding eviction sets. Most of recent designs [8], [9], [11], [15], [16] randomize skewed caches rather than classic set-associative caches. In an unlikely scenario where safety is overwhelmingly important, the recently proposed MIRAGE cache [11] claims to fully eliminate attacker-controlled associativity evictions by over-providing metadata space and introducing multi-stepped Cuckoo relocation into randomized skewed caches. However, the extensively re-structured LLC incurs a storage overhead of 22%. It is unlikely for any future processors to adopt such a disruptive solution without a strong incentive.

If safety shall act as an add-on to the existing cache structure without significantly hurting performance, we would like to question: *Is the currently widely utilized non-skewed set-associative caches really hopeless to defend?* Our recent work [10] shows that the non-skewed set-associative caches can be made sufficiently safe against conflict-based cache side-channel attacks. Cache randomization and dynamic remapping are necessary to obstruct the latest eviction set search algorithms. The strength of defense is significantly boosted by triggering extra remaps when attacks are detected in action. The performance overhead can be efficiently reduced using multi-step relocation during a remap.

In this paper, we implement the proposed randomized non-skewed set-associative cache into the LLC of the open-sourced Rocket-Chip multicore processor [17], [18]. To our best knowledge, this is the first time that a dynamically remapped randomized cache has been implemented on a Linux-capable multicore processor. It allows us to evaluate defense strength and performance overhead on FPGA rather than some simulators. The experiment results provide strong evidences showing that the classic set-associative cache can be sufficiently enhanced against conflict-based cache side-channel attacks with only marginal performance overhead. The implementation is open-sourced at:
https://github.com/comparch-security/chipyard-random-llc

Overall, this paper makes the following contributions: *A dynamically remapped randomized cache is implemented into the LLC of a Linux-capable multicore processor. A single-cycle hash logic is designed for randomizing the cache set indices. A multi-step relocation scheme is used to reduce the remap cost.*

- *W. Song, Z. Xue, J. Han, and Z. Li are with Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, Beijing, China, and the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China.*
  E-mail: {songwei, xuezihan, hanjinchi, lizhenzhen1}@iie.ac.cn
- *P. Liu is with the Pennsylvania State University, University Park, USA.*
  E-mail: pxl20@psu.edu

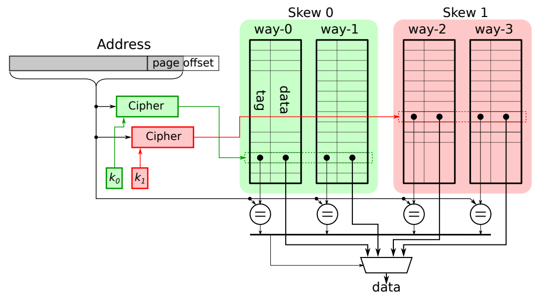


Fig. 1. A randomized skewed cache with two skews over four ways.

An attack detector is implemented to catch attacks in action and consequently trigger extra remaps. All existing eviction set search algorithms have been ported and verified failing to find eviction sets. The average runtime performance overhead is below 1% when evaluated using the SPEC CPU 2006 benchmark suite.

The paper is organized as follows: Section 2 introduces the necessary background for understanding the paper. Section 3 reasons why randomizing skewed caches might be bad. The design of a randomized set-associative cache is expanded in Section 4 and implemented into the Rocket-Chip multicore processor in Section 5. The strength of the defense and its performance overhead are evaluated in Section 6. The limitations and related work are discussed in Section 7. Finally, the paper is concluded in Section 8.

2 Background

The main objective of cache randomization is to deprive attackers from usable eviction sets [7]–[11]. Fig. 1 presents a randomized skewed cache whose four cache ways are evenly divided into two skew partitions independently indexed. Instead of using a subset of address bits, the cache set index is generated from a cipher taking the whole address and a hardware managed key as inputs. Assuming the encryption algorithm is unbroken and the key is not leaked, the cache set index is a random number unobservable to attackers. Therefore, they can no longer construct eviction sets simply by picking addresses but dynamically search for congruent addresses at runtime, which can be intolerably slow [12], [13]. However, three types of fast search algorithms have been proposed to drastically reduce the number of cache accesses required for finding an eviction set.

Group elimination (GE) is an optimization of an old algorithm against the Intel's complex address scheme [8], [12]. It starts with a large eviction set of random addresses and quickly trim it into a minimal one with only W addresses, where W is the number of ways. In each trim round, the remaining addresses are divided into $W+1$ groups. Since a minimal eviction set contains only W addresses, there is at least one removable group containing none of the W addresses. By sequentially testing whether the set is still an eviction set without a certain group, the removable group is found and removed. The trim continues until a minimal set is produced. On average, it requires $\mathcal{O}(SW^2)$ cache accesses to find a minimal eviction set in an S set LLC [13].

Conflict testing (CT) is an algorithm first proposed to find eviction sets in caches using random replacement [8], where an attacker can collect an eviction set by sequentially testing multiple random address whether any of them are congruent with the target address. The target address is accessed first to make it cached in the LLC. Then a random address is accessed. If this address is congruent with the target address, it might replace the target address by a chance of $\frac{1}{W}$ thanks to the random replacement. This condition is checked by a timed re-assess of the target address. An eviction set is produced when enough congruent addresses are collected. Overall, any random address might conflict with the target address by a probability of $\frac{1}{S \cdot W}$. The total number of cache accesses is estimated around $\mathcal{O}(SW^2)$. This algorithm is also effective for permutation-based replacement, such as least-recently used (LRU). To find a minimal eviction set with W addresses, the number of cache accesses is also around $\mathcal{O}(SW^2)$.

Prime, prune and probe (PPP) is also a search algorithm exploiting the LRU replacement [8], [19]. An attacker starts with accessing a large set of random addresses (prime set) to prime the LLC. Since self-conflicts would naturally occur, a prune process is used to remove conflicted addresses until all addresses remaining in the prime set are simultaneously cached. To collect congruent addresses from the prime set, the attacker makes a timed re-access of the target address and the prime set sequentially. Addresses with long latency (miss in the LLC) are congruent with the target. In an ideal (noiseless) scenario, just enough addresses are found in a single probe as the order of accesses observed by the LLC is the same order initiated by the attacker. The overall number of cache accesses is only $\mathcal{O}(SW)$, which is the smallest in all the three fast algorithms. This algorithm can be used when other types of replacement policies is used, such as the random replacement policy, but the overall number of accesses rises to $\mathcal{O}(SW^2)$, because the number of congruent addresses found in each search decreases to around one, leading to multiple search rounds [19].

Several derived algorithms have been proposed based on these fast algorithms. A couple of optimizations have been made in [20] to speed up the PPP algorithm in randomized skewed caches, where a search with multiple rounds is required to find an eviction set. One technique is to boost the probability of finding a congruent address by adding the found addresses as extra targets in each probe round. A similar idea has been utilized to optimize the CT algorithm, namely *CT-fast*. Whenever a congruent address is found (the target address is evicted), all the previously found congruent addresses are accessed after re-accessing the target to make the target easier to evict [21]. Instead of checking congruence by evicting the target address, detecting the prolonged latency due to the LLC enforced serialization of parallel writes to the same cache set was also found effective [22]. The resulted algorithm, namely *W+W*, operates significantly faster than the GE algorithm.

The aforementioned search algorithms can easily defeat statically randomized caches. As a result, a randomized cache has to periodically remap its content by updating the hardware managed key (k_0 and k_1 in Fig. 1). This forces an attacker to dynamically search eviction sets and finish an attack in the same remap period. Short remap period increases the hardness to launch an attack [7], [14]. However, frequent remaps lead to significant performance loss. During the remap process, all cache blocks in the LLC are sequentially relocated using the updated key. When there is no available space at the new location, a cache block is evicted to make space [7]. Our experiments show that 40% to 50% cache blocks are evicted for this reason. To reduce the performance overhead while thwarting attacks, the remap period is carefully selected. It was originally reported that a remap is required for every 47K accesses to a 1024-set 16-way LLC (only three accesses per cache block, ACC3) [8], which is an unbearably short period. This is why skewed caches are preferred and the period can be increased to 1.6M accesses (ACC100) [8] when two skews are utilized. However, ACC100 was later found vulnerable to eviction sets composed of partially congruent addresses [9], [10]. Consequently, ScatterCache pushes the number of skews to the maximum of 16 [9] while MIRAGE tries to fully eliminate associativity evictions by over-providing metadata [11]. Both of them potentially incur substantial performance overhead.

3 Reasoning Against Skewed Caches

Instead of advocating the use of randomized skewed caches, we cautiously argue that randomized non-skewed (classic set-associative) caches can be sufficiently strengthened and possess a better chance to be adopted in the near future than their skewed counterparts.

The performance benefit of using skewed caches is not proven by commercial processors and introducing it purely for security purpose might be ill-fated. With our best effort, we found that skewed caches [23] have not been adopted in the LLCs of any commercially available modern processors. The performance benefit of a skewed cache is the increased cache associativity by reducing conflict misses [23]. As the number of ways grows in modern processors, the benefit of extra cache associativity diminishes [8]. The required partitioning of cache sets undesirably reduces the efficiency of the LRU/RRIP replacement policy [12], [24] adopted by the LLCs in modern processors. Excessive skewing, such as ScatterCache, actually hurts performance.

The area and runtime performance overhead of MIRAGE is heavy. As the most advanced defense based on skewed caches, MIRAGE [11] claims to fully eliminate attacker-controlled associativity evictions by over-providing metadata space and introducing multi-stepped Cuckoo relocation. According to its own estimation, the storage overhead of the over-provided metadata space has already approached 22%. The actual overhead would be much higher due to the reduced memory density, as metadata array is partitioned, and the significantly under-estimated logic overhead. The runtime performance loss was estimated to 2.0% in term of cycles per instruction (CPI) based on simulator results, which we believe is also under-estimated. There is no discussion regarding the potential blocking scenarios due to conflicts between parallel LLC accesses. An LLC is a non-blocking cache serving multiples accesses concurrently, each of which is a multi-step transaction fulfilled in an atomic fashion: either a transaction is fully served or blocked without updating any internal state. To guarantee data consistency, conflicting transactions (accessing the same cache set) must be detected and serialized. Cache skews complicate the conflict detection logic as potential conflict is checked simultaneously in all skews. This becomes even more complex in MIRAGE as data and metadata are stored separately, leading to extra source of conflict. It is really difficult to accurately evaluate the area and runtime performance impact without a real hardware implementation.

The non-skewed set-associative caches can be made sufficiently safe against existing conflict-based cache side-channel attacks. The full elimination of associativity evictions might be an overkill with unnecessary performance overhead. What actually required is to sufficiently raise the bar for existing attacks to a level that is unviable in practice. As shown by our experiments using actual attacks running on hardware implemented processors in Section 6, dynamically remapping a randomized set-associative LLC using a combination of detectors successfully thwarts all existing search algorithms with a marginal performance overhead.

4 Randomized Set-Associative Caches

This section briefly re-describes the randomized set-associative cache [10] and proposes a single-cycle hash logic for efficiently randomizing cache set indices in hardware.

4.1 Threat Model

As the purpose of adopting cache randomization is to deprive attackers from usable eviction sets, we consider finding a usable eviction set targeting a specific address as a successful attack. Only conflict-based cache side-channel attacks targeting the LLC are considered in this paper. In order to examine the effectiveness of defenses under hostile scenarios, we give attackers the following set of generous but still reasonable capabilities:

- She has fully reverse-engineered the virtual to physical address mapping.
- She can access unlimited number of random addresses, make arbitrary memory accesses to her own data and accurately infer cache hit/miss status by measuring the access latency.
- She can accurately trick the victim into running a single memory access, and there is no other active process during the attack.
- She has the full design details of the hardware.

4.2 Randomizing Cache Set Indices

In a randomized cache, the mapping from addresses to cache set indices is randomized. The structure of such a cache is depicted in Fig. 2. Assuming a 32-bit address system, all the higher 26 address bits, except for the lower 6-bit cache block offset, are used to produce a randomized cache set index using a hasher highlighted in red.

Instead of using a multi-cycle cryptographic cipher as in other randomized caches, such as QARMA [9] and PRINCE [11], we prefer a single-cycle hasher utilizing two non-cryptographic hash functions and a key table. The major concern here is the extra latency introduced by the cryptographic ciphers. A 2-cycle linear block cipher was used in [7] but found vulnerable to a shortcuts attack [20], [25]. Consequently, all the following designs adopt the even slower non-linear cryptographic ciphers. However, as correctly pointed out by [25], even the lightweight (fast) cryptographic cipher leads to considerable overhead. This

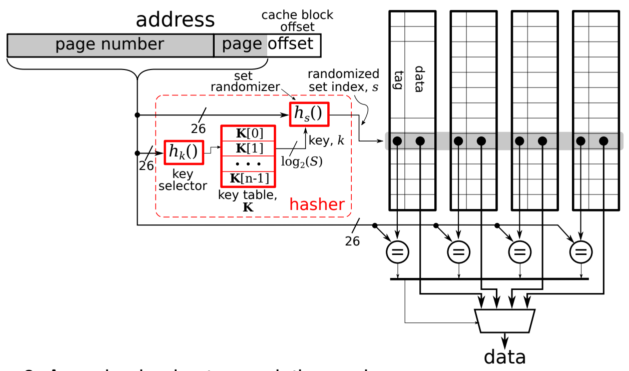


Fig. 2. A randomized set-associative cache.

is especially true for the less powerful processors using in-order cores and shared level-two (L2) caches, such as the Rocket-Chip. In our own estimation, *every extra cycle incurred by the cipher leads to 0.4∼0.8% increase in CPI.*

In layman's terms, the shortcuts attack relies on the existence of a factorization in the form of Equation 1 [25]:

$$s \leftarrow \mathcal{F}(a, k) = \mathcal{C}(a) \oplus \mathcal{D}(k) \tag{1}$$

where (a, k) is the address and key pair used by the cipher, $s \leftarrow \mathcal{F}()$ is the cipher function, which can be factored into two functions $\mathcal{C}()$ and $\mathcal{D}()$. As a result, when two addresses a_0 and a_1 are found congruent, the congruent relation is irrelevant to k. Replacing k does not invalidate the congruent relation between a_0 and a_1.

$$\mathcal{F}(a_0, k) = \mathcal{F}(a_1, k) \tag{2}$$
$$\Rightarrow\ \mathcal{C}(a_0) \oplus \mathcal{D}(k) = \mathcal{C}(a_1) \oplus \mathcal{D}(k) \tag{3}$$
$$\Rightarrow\ \mathcal{C}(a_0) = \mathcal{C}(a_1) \tag{4}$$

To resolve this problem, the possible factorization described in Equation 1 must be eliminated, such as using a non-linear $\mathcal{F}()$. A cryptographic cipher certainly works, but its 3∼5 cycle latency [11] incur substantial performance loss. We would like to propose a non-cryptographic but arguably strong enough single-cycle hasher.

As depicted in Fig. 2, our solution is a combination of two non-cryptographic hash functions: a simple linear hasher $h_k()$, a non-cryptographic non-linear hasher $h_s()$, and a table $\mathbf{K}$ storing an array of randomly generated keys. The overall function can be described as:

$$s \leftarrow \mathcal{F}(a, \mathbf{K}) = h_s(a, \mathbf{K}[h_k(a)]) \tag{5}$$

The $h_s()$, namely the *set randomizer*, is the main hash function $\mathcal{F}()$ used to produce the randomized cache set index. It adopts a non-linear design to eliminate the possible factorization described in Equation 1. Although $h_s()$ alone has achieved the goal of randomizing cache set indices, it may suffer from brutal-force attacks trying to decipher the key as its implementation is considered publicly available. Therefore, the *key selector* $h_k()$ and the *key table* $\mathbf{K}$ are introduced as an extra layer of protection. Instead of using the same key, $h_k()$ randomly chooses a key from $\mathbf{K}$ indexed by hashing the input address a: $k \leftarrow \mathbf{K}[h_k(a)]$. During a remap, both the key selector $h_k()$ and the key table $\mathbf{K}$ are regenerated. In this way, it becomes almost impossible for an attacker to tell whether two congruent addresses share the same key, not to mention breaking the hasher.[1]

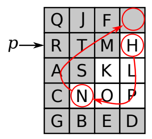


Fig. 3. Remapping cache block H using the multi-step relocation.

The whole hasher is implemented in combinational circuit finishing in just one cycle. As $\mathbf{K}$ is small, it is stored in a register array. The key width is set to $\log_2(S)$ where S is the number of sets. This is the minimal width to ensure that an address can be randomly mapped to all available sets. The detailed hardware implementation of these hash functions is described in Section 5.2.

4.3 Periodically Remapping

Cache remapping [7] is used to limit the time window available to attackers. Using the fast search algorithms described in Section 2, attackers may collect a number of congruent addresses forming an eviction set. To prevent attackers from constructing and further utilizing eviction sets, a remap is periodically triggered to randomly rearrange the mapping of addresses to cache sets, which effectively nullifies the congruent addresses found by the attacker.

During a remap, all cache blocks are sequentially relocated to new cache sets using the regenerated hash functions. Unavoidably, this relocation process causes cache conflicts leading to cache blocks being evicted from overcrowded sets while some sets are left with unoccupied ways. Existing research estimated that around 40∼50% of cache blocks are evicted in each remap [10]. To reduce the performance loss asserted by cache remapping, the cache remap period should be carefully selected to be just short enough to thwart most attacks using the fast search algorithms. For non-skewed set-associate caches, a remap is needed for every 10 evictions per cache block on average [10].

A further optimization is to adopt a multi-step relocation method in the remap process. When a cache block is relocated to a full cache set, instead of evicting one cache block to make a room, the remap tries to swap the block with a block still mapped using the old hash functions and further relocate the swapped block. A block is evicted only when all the blocks are already mapped using the new hash functions. Fig. 3 demonstrates an example of remapping cache block H (indicated by the remap pointer p) in a 5-set 4-way cache using the multi-step relocation, where all remapped blocks are shadowed in gray. H is relocated to set 4, which still contains blocks waiting to be remapped. Using the multi-step relocation, block N is swapped with H, and it is further relocated to set 1 which has an empty block. As a result, no block is evicted. Multi-step relocation significantly reduces the ratio of evicted blocks from 40% to just 10% [10].

1. Unlike in a traditional cryptanalysis where the ciphertext is observable, the randomized cache set index is unavailable to attackers. By searching for congruent addresses, an attacker might extract time-invariance of $h_s()$ and eventually break it. However, the use of $h_k()$ makes such analysis almost infeasible, and searching congruent addresses triggers remaps which in turn nullify the analysis. Both effects make the hasher resistant to brutal-force attacks.

4.4 Attack Detection

Although remap at a high frequency thwarts most side-channel attacks, it incurs observable performance loss due to the evicted cache blocks. Instead, moderately reducing the remap frequency while triggering extra remaps when attacks using fast search algorithms are caught in action can reduce the performance loss. Existing research has shown that both PPP and GE algorithms can be reliably detected [10], because they need to prune a large set of random addresses into a minimal eviction set and exceptional number of evictions are incurred on the targeted cache set during the prune process. An active attack can be detected accordingly by constantly monitoring the distribution of evictions among cache sets.

The proposed detection utilizes the Z-Score standardization [26], which is a standard way of measuring imbalance across samples. The Z-Score of a cache set is measured as:

$$z_i = \frac{e_i}{\sqrt{\frac{\sum e^2}{S-1}}} \tag{6}$$

where e_i is the number of evictions occurred in a monitoring period on cache set i and z_i is the calculated score for the set. z_i approaches to $\sqrt{S}$ when all evictions in the monitoring period occur on the i-th set (potentially targeted). However, this might lead to false-positive errors when the cache miss rate is extremely low and only one eviction happens on a benign set. Consequently, the detector relies on an accumulated and weighted score (y) calculated as:

$$y_i[t] = (1-\alpha)\cdot y_i[t-1] + \alpha\cdot(e_i - \bar{e})\cdot z_i \tag{7}$$

where $(e_i - \bar{e})\cdot z_i$ is the weighted score to represent the number of evictions and α is the discount factor (empirically set to $1/32$) of the exponential moving average (EMA) [27] applied on the weighted score. This y_i ranges between 0 and W and approaches W when W extra evictions unevenly occur on the i-th set in a short period of time. Since attackers might hide themselves by prolonging the search algorithm, a threshold (h) shall be carefully selected to detect most attacks while leaving benign applications unreported. It is estimated that the success rates of PPP and GE reduce to almost nil if $h = 5$ [10].

5 Hardware Design

The randomized set-associative cache described in Section 4 has been implemented into the shared L2 cache [28] (acting as the LLC) of the Rocket-Chip processor [17]. This section describes the hardware design of the randomized cache along with the extensions made to Rocket-Chip.

5.1 Overall Structure

The overall structure of the randomized LLC cache is depicted in Fig. 4 with the newly added modules highlighted in red. Adopting the SiFive Tilelink protocol, five channels (A to E) are used for the communication with the inner L1 cache and the outer memory. Table 1 provides a functional description of these channels, along with the special channel X for SiFive's extension of software-controlled cache flush operations and the newly added channel P for our extension of software accessible performance counters (PFCs). The

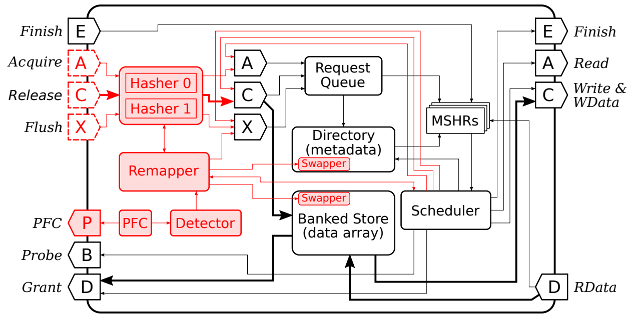


Fig. 4. Overall structure of the randomized LLC in the Rocket-Chip.

TABLE 1
Function of the extended TileLink channels

	L1 side (left)	Memory side (right)
A	*Acquire*: L1 asks for a block or a permission upgrade.	*Read*: LLC reads a block from memory.
B	*Probe*: Coherence probes from LLC to L1 caches.	Not used.
C	*Release*: L1 writes back a block or answers a *probe*.	*Write & WData*: LLC writes back a block to memory.
D	*Grant*: LLC's response to a *release* or an *acquire* (may with data).	*RData*: LLC gets the block asked by a previous *read*.
E	*Finish*: L1 denotes the end of an *acquire* transaction.	*Finish*: LLC denotes the end of a *read* transaction.
X	*Flush*: A processing core requests to flush a block.	Not used.
P	*PFC*: LLC sends PFC counts to processing cores.	Not used.

L1 side of channels support coherent cache transactions while only normal (uncoherent) memory transactions are supported on the memory side. An exemplary transaction is depicted in Fig. 5. It starts with an L1 sending an *Acquire* on channel A to fetch a missing cache block B_0. Unfortunately, B_0 also misses in the LLC and the cache set is full. Block B_1 is chosen by the replacer to make a room. As B_1 is in a modified status (assuming an MSI protocol), LLC *probes* all L1 caches for the latest copy of B_1 through channel B and later receives it through a *Release* message on channel C (which is immediately acknowledged by a *Grant* message on channel D). Block B_1 is then written back to memory through a *Write & WData* message on channel C. Almost at the same time, LLC requests the memory for the missing B_0 by a *Read* message on channel A. The memory then sends back B_0 (*RData*) on channel D, which is quickly forwarded to the requesting L1 by the LLC using a *Grant* on channel D. Both the L1 and the LLC send back *Finish* messages to finish their (*Acquire*/*Read*) transactions, respectively.

A number of modules have been added into the LLC to support dynamic randomization. For all L1 side channels requesting to access a specific block, i.e., channel A (*Acquire*), C (*Release*) and X (*Flush*), the channel input module is pushed further into the LLC to allow the added hashers

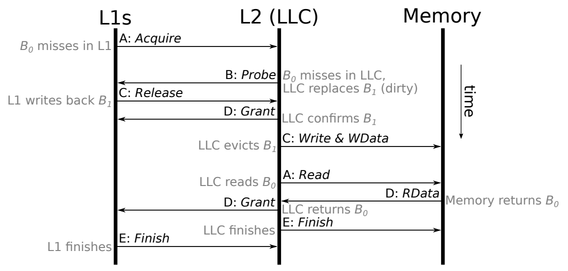


Fig. 5. An exemplary LLC transaction using the Tilelink channels.

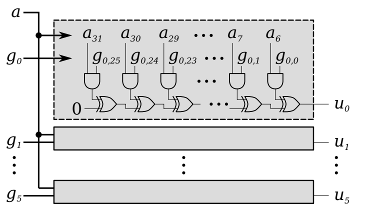


Fig. 6. Implementation of the key selector $h_k()$.

(*hasher* 0 and 1 in Fig. 4) to produce the randomized cache set indices beforehand. During a remap, the functions of these hashers are regenerated by the *remapper*, which also schedules the relocation of all cache blocks with the help of two *swappers* added in the *directory* and the *banked store*, respectively. A *detector* is responsible for triggering remaps based on cache states collected by the *PFC*. Both periodical and attack triggered remaps are implemented.

5.2 Hash Function

The internal structure of the proposed hasher, comprising a *key selector*, a *set randomizer*, and a *key table*, has already been depicted in Fig. 2 (Section 4.2). In each cycle, it produces a randomized cache set index s according to the input address a using a key table storing 64 10-bit keys (assuming a 1024-set LLC). As shown in Fig. 6, similar to Intel's complex addressing scheme [29], the hash function $h_k()$ can be described as:

$$u = h_k(a, \mathbf{G}) \tag{8}$$

where u is the 6-bit index for choosing a random key in the key table and $\mathbf{G}$ is a vector containing six independent generation factors $\{g_0 \ldots g_5\}$. The i-th bit of u is generated from a using g_i as follows:

$$u_i = (a_6 \cdot g_{i,0}) \oplus (a_7 \cdot g_{i,1}) \oplus \cdots \oplus (a_{31} \cdot g_{i,25}) \tag{9}$$

Only the higher 26 bits of a are used in the generation omitting the lower 6-bit block offset. The random generation factor g_i is also 26 bits wide. Implemented in combinational circuit, the key selector consumes a tiny portion of a cycle.

Using a key randomly chosen by the key selector from the register-implemented key table $\mathbf{K}$, the randomized cache set index s is produced by the set randomizer $h_s()$ as depicted in Fig. 7. Following the methodology described in [30] for designing high-performance non-cryptographic hash functions, this hasher is composed of multiple linear XOR stages and non-linear S-box stages connected by randomly shuffled interconnects. The final 10-bit cache set index (s) is produced by overlapping the 36-bit output (o): $s_i = o_i \oplus o_{i+10} \oplus o_{i+20} \oplus o_{i+30}$. Each XOR stage implements a sparse and invertible matrix multiplication using 2-input and 3-input XOR gates. The non-linear S-box stage includes an array of AOI222 gates each mixing three consecutive bits. Denoting an S-box stage as "*t*" and three consecutive XOR stages as "*x3*", a 9-stage *tx3tx3t* hash function is found performing sufficiently effective in the generalized uniformity test, the avalanche test and the universality test [30]. In layman's words, this hash function statistically ensures that the hashed outputs are uniformly distributed, one flip on any input bit leads to random flips on ∼50% output bits, and changing the key k results in a fresh new hash function.

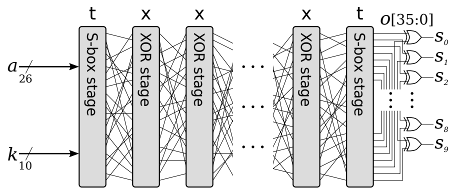


Fig. 7. Implementation of the set randomizer $h_s()$.

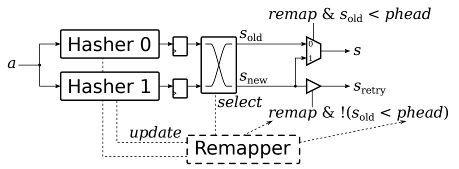


Fig. 8. Generating the randomized cache set index using two hashers.

The whole hasher is implemented in pure combinational circuit and fast enough to finish in one cycle.

As a part of a remap, the whole hasher is regenerated to produce a new mapping for cache set indices. To be specific, the vector of generation factors $\mathbf{G}$ controlling the key selector $h_k()$ and the key table $\mathbf{K}$ are refilled with independently generated random numbers. The update of $\mathbf{G}$ ensures that any pair of addresses using the same key in the previous remap period would likely use different keys, while the update of $\mathbf{K}$ replaces $h_s()$ with a new one.

5.3 Cache Set Randomization

This section explains how cache blocks are addressed in the LLC using the hashers described in Section 5.2. A pair of hashers are shared by all requesting channels on the L1 side, i.e. channels A, C and X, as shown in Fig. 4. They are placed before the channel input modules; therefore, the randomized cache set index of an incoming message is made available and stored inside the message before it is processed by the channel input module. This also asserts the minimum modification to the original message processing logic as most of it is unaware of the change in cache indices.

Instead of one, two hashers are used to support uninterrupted operation during remaps, as explained in Fig. 8. In normal operation when no remap is active (*remap* = *false*), one of the hashers, *selected* by the remapper, generates the randomized cache index using the current (old) mapping ($s \leftarrow s_{\text{old}}$). Meanwhile, the unused hasher is proactively *updated* by the remapper to prepare a new mapping for the next remap period. When a remap is triggered (*remap* = *true*), two indices are generated by both hashers simultaneously: s_{old} using the old mapping and s_{new} using the new mapping. s_{new} is chosen when the cache set pointed by s_{old} is already remapped (s_{old} < *phead*, see Section 5.4), indicating s_{old} is invalid. Otherwise, s_{old} is chosen by default. As explained in Section 5.4, cache blocks in unremapped sets may have already been relocated using the new mapping due to the multi-step relocation. If a block is found missing using s_{old}, it might be relocated and should be found again using s_{new}. For this reason, s_{new} is also stored in the message as s_{retry}, allowing the *scheduler* to reissue a failed request back to the input module using s_{retry} as index. The detailed reasoning of these conditions will be revisited in Section 5.4 after the remap logic is explained.

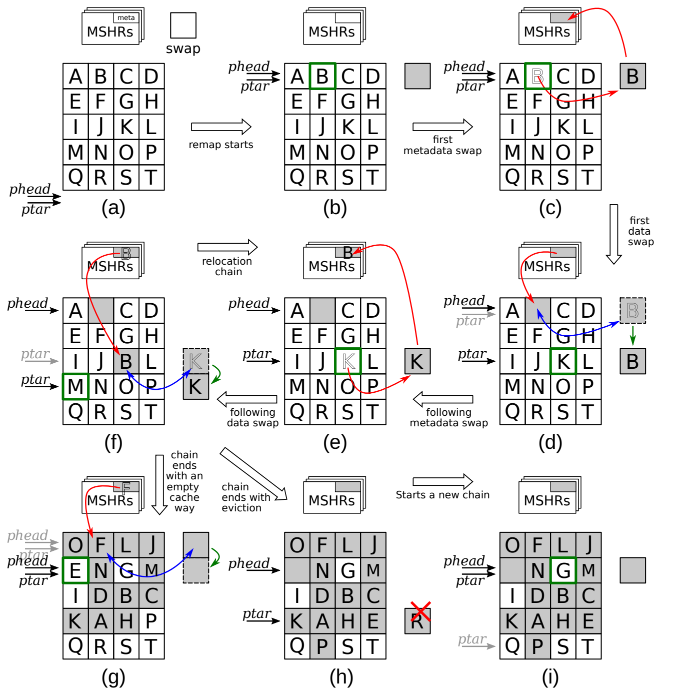


Fig. 9. General procedure of remapping a 5-set 4-way cache: (a) normal operation when remap is inactive, (b) starting a remap, (c) the first metadata swap, (d) the first data swap, the following (e) metadata swap and (f) data swap on a relocation chain, a relocation chain ending by (g) swapping the final block with an empty cache way or (h) evicting the final block as it is relocated to a fully occupied and remapped set, (i) starting a new relocation chain. The move of metadata (data) is colored in red (blue) arrows. The block chosen to be relocated next is highlighted in green. All remapped blocks are shadowed in gray.

5.4 Cache Remapping

A remap is triggered by the *detector* and scheduled by the *remapper*. This section describes the internal logic of the *remapper* while the *detector* is discussed in Section 5.5.

5.4.1 General Description

A remap is a multi-cycle procedure adopting the multi-step relocation described in Section 4.3. All blocks are sequentially remapped (relocated to a new cache set) by multiple relocation chains, each of which is a chain of block swaps triggered by the relocation of an unremapped cache block. This starting block of a new relocation chain is always chosen from the unremapped cache set with the smallest index. A relocation chain finishes when one of two designated ending conditions is satisfied. We will shortly present the two conditions. A *swap* buffer, which is just capable of storing one cache block along with its metadata, is added to temporarily store the cache block being relocated. The buffered block can be addressed as an extra way added to the new cache set (similar to a victim cache). Several global variables are added as well, including *remap* denoting a remap is active, a *phead* pointer identifying the starting set of the current relocation chain and a *ptar* pointer labeling the set currently being relocated. The metadata associated with each cache block is extended to store the full (26 bits of) address because the cache set index is no longer a segment of the address. A *remap-id* bit is also added to the metadata denoting whether a block is remapped.

Fig. 9 describes the general procedure of a remap. If the remap is inactive, the cache operates normally, and both *phead* and *ptar* point to the end of the cache as depicted in Fig. 9a. Once a remap is triggered, as shown in Fig. 9b. both pointers are retargeted to point at cache set 0, as it will be the first to be remapped, and the (empty) block stored in the swap buffer is set as remapped by toggling its *remap-id* bit. The remap procedure proceeds in multiple relocation chains. The first relocation chain starts with a randomly selected unremapped block (block **B** in Fig. 9b) in cache set 0. This block is swapped with the empty block stored in the swap buffer in two steps: a metadata swap (Fig. 9c) and a data swap (Fig. 9d). Instead of letting the remapper write the directory directly (using an extra write port), a missing state handling register (MSHR) is borrowed for this operation. In the metadata swap, the metadata of the swap buffer is first moved to an MSHR acquired by the remapper and then the metadata of block **B** is copied into the swap buffer. Later in the data swap, shown in Fig. 9d, the MSHR writes the metadata of the original swap buffer (although empty for the first metadata swap of a relocation chain) into the place of block **B** as a normal metadata update operation. Meanwhile, the data of block **B** and the swap buffer are swapped. Assuming cache set 2 is the new set for block **B** identified by the remapper, the swap buffer is assigned to this set along with *ptar*. The relocation chain follows on with swapping block **B** with a randomly selected unremapped block in cache set 2. Let's say block **K**. This swap is similarly executed by a metadata swap (Fig. 9e) followed by a data swap (Fig. 9f). Afterwards, block **K** is temporarily stored in the swap buffer waiting to be relocated (to cache set 3) by the second swap of the current relocation chain.

This chain of swaps in the current relocation chain continues until one of two possible ending conditions appears: One ending condition occurs when the block in the swap buffer is relocated to an empty cache way in a fully remapped cache set. An example is illustrated in Fig. 9g, where block **F** is relocated to the remaining empty way of the fully remapped cache set 0. Consequently, the swap buffer becomes empty and the current relocation chain ends. Both *phead* and *ptar* progress to the next cache set, and a block is randomly chosen to start a new relocation chain. The other ending condition occurs when the block in the swap buffer is relocated to a fully remapped set with no empty way. An example is demonstrated in Fig. 9h, where block **R** is relocated to the fully remapped and occupied cache set 3. Instead of enforcing a swap, block **R** is evicted to clear up the swap buffer. Pointer *ptar* returns to the cache set pointed by *phead*, and a block is randomly chosen to start a new relocation chain, as in Fig. 9i. For both ending scenarios, if the cache set pointed by *ptar* is already fully remapped, both *phead* and *ptar* progress to the next cache set until all cache sets are remapped, when the remap is consequently finished.

Four optimizations have been applied to the design: The first one is to *maximize the reuse of existing logic*. In the swap of metadata, an MSHR is borrowed for writing metadata to the directory rather than introducing a new write port. When a block is evicted at the end of a relocation chain, the eviction is done by issuing a flush operation to channel X by the remapper. The second one is to *swap metadata and data separately*. This allows the data swap to operate asynchronously in the background, reducing the

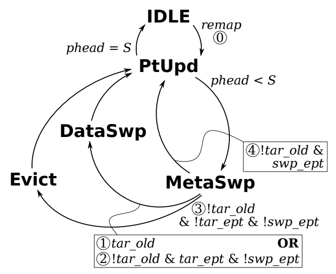


Fig. 10. State machine of the remapper

overall remap latency. The third one is to *make the swap buffer addressable*. The cache block temporarily stored in the swap buffer is considered remapped. It takes part in the way matching logic of the new (relocated) cache set as an extended way, so that cache accesses to this block from the inner L1 caches are not interrupted during the swap, reducing impact on normal cache accesses. The final one is to *deprioritize the remapper's requests to MSHR and channel X*, which also reduces impact on normal cache accesses.

Finally, let us return to the issue of choosing the cache set index during an active remap, as described in Section 5.3. Two indices, s_{old} and s_{new}, are simultaneously produced using both the old and the new mappings (Fig. 8). When $s_{\text{old}} < phead$, $s \leftarrow s_{\text{new}}$ because the cache set pointed by s_{old} has already been remapped and the requested block must be stored in the new cache set pointed by s_{new}. Otherwise, the requested block might still be stored in the cache set pointed by s_{old}. In this case, $s \leftarrow s_{\text{old}}$ as the old cache set should be checked first. However, some blocks in this set may have been relocated due to multi-step relocation, e.g., block **K** is relocated to from set 2 to set 3 as shown in Fig. 9g. This is why s_{new} is still provided as a retry index s_{retry} just in case that the new cache set needs to be checked as well.

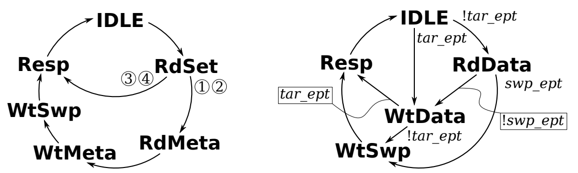
5.4.2 Remapper

As depicted in Fig. 4, the *remapper* is the central controller of the remap procedure while the actual swap of metadata and data is offloaded to the two swappers added in the *directory* (for metadata) and the *banked store* (for data), respectively.

The state machine of the remapper is described in Fig. 10. The state is **IDLE** when remap is inactive (Fig. 9a). When a remap is triggered (*remap* $\leftarrow$ $true$), the state transits to **PtUpd** where *phead* and *ptar* are updated. There are five different scenarios for updating these pointers. The initial entering from **IDLE** is scenario ⓪, where *phead* and *ptar* are both reset to 0 (Fig. 9b), starting the first relocation chain. The remaining four scenarios depend on the information collected in the next state **MetaSwp**.

A block relocation process starts with state **MetaSwp**, where the metadata swapper inspects the state of the cache set pointed by *ptar*, chooses an unremapped block (if any) in this set, and swaps the block's metadata with the one stored in the swap buffer. The detailed operation is soon explained in Section 5.4.3. Afterwards, states of both the cache set and the swap buffer are returned to the remapper for deciding the next operation. There are four different scenarios:

① $tar\_old = true$: **MetaSwp** $\rightarrow$ **DataSwp**. The cache set has unremapped blocks as indicated by *tar_old*. As an example shown in Fig. 9e, cache set 2 has unremapped blocks. The unremapped block **K** is chosen, whose metadata



(a) Metadata (b) Data

Fig. 11. State machines of (a) the metadata and (b) the data swappers.

is swapped with the block **B** stored in the swap buffer. The data swapper is consequently scheduled to swap the data.

② !*tar_old* & *tar_ept* & !*swp_ept*: **MetaSwp** $\rightarrow$ **DataSwp**. The cache set is fully remapped but with empty ways (*tar_ept*) while a block is stored in the swap buffer (!*swp_ept*). As an example shown in Fig. 9g, cache set 0 are remapped but there is an empty way. The block **F** stored in the swap buffer is therefore swapped into this way. Similar to ①, the data swapper is scheduled to swap the data.

③ !*tar_old* & !*tar_ept* & !*swp_ept*: **MetaSwp** $\rightarrow$ **Evict**. The cache set is fully remapped and fully occupied. The block stored in the swap buffer is therefore evicted. As an example shown in Fig. 9h, the block **R** in the swap buffer is relocated to cache set 3, which is fully remapped and fully occupied. A flush operation is issued to channel X to evict **R**.

④ !*tar_old* & *swp_ept*: **MetaSwp** $\rightarrow$ **PtUpd**. This is a corner case. The cache set if fully remapped while the swap buffer is empty. Nothing waits to be done on this set and both pointers (*phead* and *ptar*) progress to the next set.

These scenarios also decide how pointers are updated later in state **PtUpd**: For scenario ①, *ptar* $\leftarrow$ s_{new} as the relocation chain continues on relocating the block stored in the swap buffer. Scenarios ② and ③ are the two ending conditions for a relocation chain. *ptar* $\leftarrow$ *phead* to start a new chain. Finally for the corner case ④, both pointers progress to the next cache set by *phead*++; *ptar* $\leftarrow$ *phead*. When all cache sets are remapped (*phead* $= S$), the remap is finished. The state returns to **IDLE** and *remap* $\leftarrow$ *false*.

5.4.3 Metadata and Data Swappers

The metadata swapper follows a state machine depicted in Fig. 11a. On receiving a request from the remapper, the state transits from **IDLE** to **RdSet**, where the swapper acquires the state (unremapped blocks and empty ways) of the cache set pointed by *ptar*. Depending on the state, the metadata swapper proceeds differently according to the four scenarios described in Section 5.4.2. A metadata swap is needed for scenarios ① and ②. The metadata of a chosen block is first read and latched in the swapper in state **RdMeta**. An MSHR is then being acquired through the remapper in state **WtMeta** to schedule a write of the metadata stored in the swap buffer to the chosen block in the cache set. Once an MSHR is allocated, the previously latched metadata is then written to the swap buffer in state **WtSwp**. For scenarios ③ and ④, metadata swap is bypassed as it is not needed. In all scenarios, the swapper returns to **IDLE** after forwarding the state of the cache set to the remapper in state **Resp**.

Data swaps are fulfilled by the data swapper following the state machine depicted in Fig. 11b. A swap starts when a request is received from the remapper in state **IDLE**. If the block to be swapped is non-empty (!*tar_ept*), it is latched to a local store to make a room in state **RdData**. Then the block stored in the swap buffer is moved to this block in state **WtData** if the swap buffer is non-empty (!*swp_ept*). Finally, the locally latched block is written to the swap buffer in state **WtSwp**. The remapper is informed in state **Resp** and the swapper returns to **IDLE** afterwards.

5.5 Attack Detection

The *detector* in Fig. 4 monitors the cache states and triggers remaps when one of its detection criteria is met. The current detector implements two detection criteria but more could be added in the future. One is a passive criterion which triggers a remap when the number of evictions overpasses a predefined threshold. As described in Section 4.3, it limits the time window available to attackers for finding and exploiting eviction sets. Setting the threshold to 10 evictions per cache block (EV10) was found sufficient to thwart the CT algorithm [10]. This criterion is easy to implement. The number of evictions is constantly monitored by the *PFC* module. The *detector* notifies the *remapper* whenever enough evictions are recorded from the last remap.

The other one is an active criterion which triggers a remap when an attack is detected in action using the detection algorithm described in Section 4.4. The hardware implementation needs to resolve three issues: **(a)** monitoring the number of evictions on each cache set, **(b)** the calculation of the Z-Score for each cache set according to Equation 6, and **(c)** the calculation of the accumulated weighted score according to Equation 7.

Issue (a): The number of evictions on each cache set is recorded by two SRAMs $\mathbf{E}$ and $\mathbf{E}'$. Both SRAMs are initially zeroed, and $\mathbf{E}[i]$ increases by one when a block is evicted from cache set i. When the current monitoring period finishes with enough amount of cache accesses, $\mathbf{E}[i]$ is used in the Z-Score calculation as e_i. Since reading SRAM $\mathbf{E}$ takes time, evictions of the new monitoring period are recorded in $\mathbf{E}'$ to avoid interference. The two SRAMs swap places after Z-Scores are calculated and $\mathbf{E}$ is zeroed.

Issue (b): Instead of directly calculating the Z-Score, $\sum e$ and $\sum e^2$ are first accumulated by a scan of SRAM $\mathbf{E}$. $\sum e$ is averaged into $\bar{e}$ for Equation 7 while $\sum e^2$ is used for producing the reverted square-root-mean (q):

$$q = \frac{1}{\sqrt{\frac{\sum e^2}{S-1}}} \tag{10}$$

Since S is typically a 2's power and $S \gg 1$, $\sum e^2/(S-1)$ is approximated to right shifting $\sum e^2$ by $\log_2 S$ bits, e.g., 10 bits for a 1024-set LLC. The calculation of q needs a square root and a division operator, both of which are successively approximated by the multiplicative iteration [31] described in Algorithm 1. This is not the most advanced algorithm but adequate enough for our purpose. Since the detection algorithm can tolerate certain level of noise, the calculation of Z-Scores does not require full precision. The multiplicative iteration method allows us to easily control the width (`<j,k>` in Algorithm 1) of the calculated result. When q becomes available, Z-Scores can be calculated according to Equation 6 by scanning SRAM $\mathbf{E}$ for a second time: $z_i = e_i \cdot q$.

Issue (C): The accumulated weighted scores y are calculated during the second round of scan of SRAM $\mathbf{E}$ as well.

```
Input: a: fix<n,m>; op: '/' or '√'.
Output: b: fix<j,k>.
function b ← op(a)
    b = 0
    t = (op == '/') ? 1.0 : a
    for i = j+k-1 : 0 do
        b[i] = 1'b1
        c = (op == '/') ? (b * a) : (b * b)
        b[i] = (c > t) ? 1'b0 : 1'b1
    end
    return b
end
```

Algorithm 1: Approximate square root or division (`fix` denotes fixed-point real number).

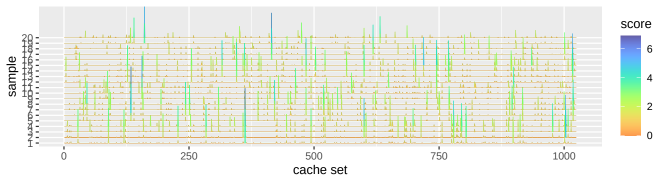
The score from the previous monitoring period $y[t-1]$ is stored in an SRAM $\mathbf{Y}$ (zeroed after a remap). $y_i[t-1]$ is fetched from $\mathbf{Y}[i]$ at the same time when z_i is calculated. When all variables on the right-hand side of Equation 7 become available, $y_i[t]$ is calculated and written back to $\mathbf{E}[i]$ for the next period. If $y_i[t]$ is larger than the pre-defined detection threshold h (set to 5 in [10]), an attack is detected and a remap is thus triggered.

Considering a 1024-set 16-way LLC, the precision of SRAM recorded variables is chosen as follows: e_i: `fix<5,0>`; y_i: `fix<3,10>`. A 5-bit unsigned integer is wide enough for e_i as the evictions recorded on one cache set is bounded by $2W$ using a proper monitoring period. As for y_i, it is capped to 8.0 as any value larger than 5.0 triggers a remap. A wide fractional part is kept for maintaining high precision in the EMA calculation (Equation 7). The precision of the intermediate variables is chosen as follows: $\bar{e}$: `fix<5,10>`; e^2: `fix<20,0>`; q: `fix<5,10>`; z_i: `fix<10,20>`; $(e_i - \bar{e})$: `signed fix<6,10>`. Full precision is maintained for $\bar{e}$, e^2 and z_i. The reverted square-root-mean q has both its integer and fractional parts truncated but only the fractional part suffers from marginal precision loss. Nearly all variables are positive and is recorded as unsigned except for $(e_i - \bar{e})$, which might be negative and is kept signed as a way for noise cancellation. The final accumulated weighted score y_i is almost always positive. When it is indeed negative, $|y_i|$ is tiny and approximated to 0. y_i is therefore recorded unsigned and truncated to `fix<3,10>`. The overall calculation latency for all cache sets is around $8S$ cycles, eight cycles per cache set on average.

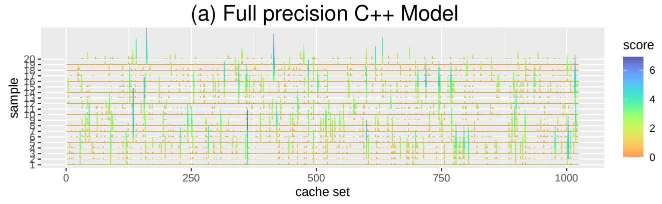
Fig. 12 presents a graphic illustration of the estimation error incurred by the hardware implementation of the active detector. Samples of the eviction distribution over 20 consecutive monitoring periods are collected by running the GE algorithm on a 1024-set 16-way LLC and fed to a perfect C++ model along with an RTL simulation of the hardware detector. The estimated accumulated weighted scores y_i are shown in Fig. 12a and 12b for the C++ model and the hardware implementation, respectively. When the score on a set rises near or pass the threshold (colored in dark blue), it is detected and a remap is triggered. The results presented in both figures are almost identical. Further proved by Fig. 12c, the (absolute) error between the two is less than 0.006, negligible as it is only 0.12% of the threshold.

6 Performance Evaluation

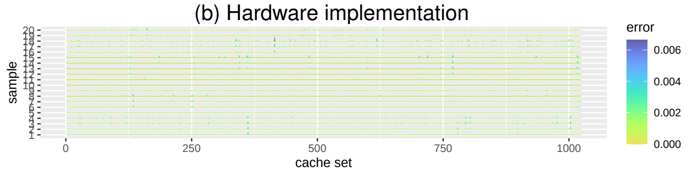
The randomized set-associative cache has been implemented into the shared LLC (L2) of the Rocket-Chip proces-



(a) Full precision C++ Model



(b) Hardware implementation



(c) Error incurred by hardware (height $\times 100$)

Fig. 12. Estimation error (c) of the accumulated weight score y_i between the C++ model (a) and the hardware implementation (b).

sor [17] open-sourced in the Chipyard project [18]. A dual-core Rocket-Chip (32KB PLRU L1-I and L1-D, 1MB 1024-set 16-way PLRU L2) has been ported to a Digilent Genesys-2 FPGA board. The processing cores run at 75MHz while the off-chip memory (1GB) runs at 900MHz. The system boots into a Linux kernel (ver. 5.11.0) using busybox and openSBI. Both the SPEC CPU 2006 benchmark and demonstrative attacks are compiled using GNU GCC (ver. 9.2.0). To evaluate the overhead in ASIC designs, the processor has also been synthesized by a commercial tool using the Nangate 45nm open cell library [32], with the area and power of RAMs estimated by CACTI 6.5 [33].

A number of remap related PFCs are added to the LLC along with a new PFC reading interface. The original Rocket-Chip supports a range of PFCs monitoring the processing core and the main coherent bus. However, only a small number (four on SiFive's U740 [34]) of events can be monitored in parallel. Since we need much more parallel PFCs to evaluate the performance of different randomization techniques, a separate PFC interconnect and counter system has been added to make all PFCs run in parallel and accessible by only three extra control registers [35].

6.1 Consistency Between Spike and FPGA Results

The performance of the randomized caches presented in [10] was collected using the Spike [36] instruction level simulator with a behavioral cache model [13], while it is collected from a Rocket-Chip processor running on FPGA in this paper. We would like to assess the consistency between the two as a way to show that the conclusions presented in [10] are reasonable. Fig. 13 depicts the misses per K instructions (MPKI) of running the SPEC CPU 2006 benchmark on both platforms.[2] MPKI of the statically randomized LLC (static) shows similar trends on both Spike and FPGA while it is smaller on the latter. Programs were run in a bare-metal mode on Spike without the full OS support (file I/O served by host) while a Linux kernel boots and runs in the background on FPGA. Since PFC's record of instruction counts kernel as well, the overall memory throughput is brought down, which is easily noticeable for the memory heavy benchmarks. The only significant discrepancy appears on 450.soplex, presenting a substantially low MPKI on FPGA. Further investigation shows that 450.soplex (194KB/s) is one of the most I/O heavy benchmarks [37], where the impact of the different ways in serving file accesses is the most visible. Periodically remap (EV10) results in marginal overhead for the average MPKI on both platforms. Considering the reasonable discrepancies, the results are consistent. Fig. 13 also demonstrates the MPKI of a Rocket-Chip processor utilizing a non-randomized LLC (non-random). Instead of incurring extra cache misses, cache randomization (static) actually reduces MPKI by $\sim$1%, as some benchmarks benefit slightly from the reduced conflict misses due to the randomization across cache sets.

2. Only benchmarks successful on both platforms are shown. On Spike, 21 out of the 29 benchmarks run successfully due to the limit support of syscalls of running SPEC in the bare-metal mode. As for FPGA, 429.mcf and 434.zeusmp fail due to lack of memory.

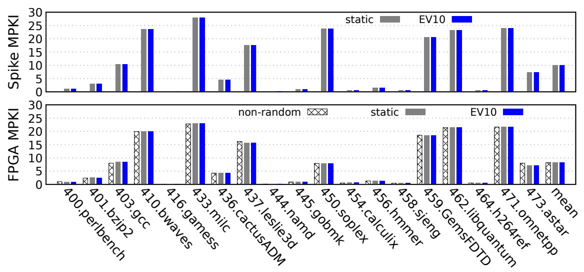


Fig. 13. Misses per K instructions (MPKI) of running SPEC CPU 2006 (100G instructions) on both Spike (static and EV10) and FPGA (non-random, static and EV10).

TABLE 2
Success rate of fast algorithms under different defenses (each test is repeated by 1000 times, detector threshold $h = 5.0$, DT1/4: calculate Z-Score after one or four accesses per cache set).

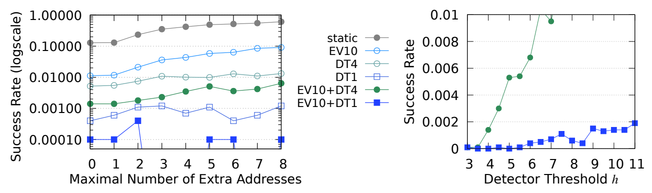
	Detector Combination					
Alg.	**Static**	**EV10**	**DT4**	**DT1**	**EV10+DT4**	**EV10+DT1**
GE	97.2%	$\sim$0.0%	$\sim$0.0%	$\sim$0.0%	$\sim$0.0%	$\sim$0.0%
PPP	20.4%	13.3%	$\sim$0.0%	$\sim$0.0%	$\sim$0.0%	$\sim$0.0%
CT	14.8%	$\sim$0.0%	$\sim$0.0%	$\sim$0.0%	$\sim$0.0%	$\sim$0.0%
CT-fast	13.9%	2.5%	1.0%	0.1%	0.3%	$\sim$0.0%
W+W	17.4%	11.7%	$\sim$0.0%	$\sim$0.0%	$\sim$0.0%	$\sim$0.0%

6.2 Strength of Defense

To demonstrate the strength of randomized caches against the fast eviction set search algorithms described in Section 2, we have ported all of them to Rocket-Chip except for the optimized PPP [20] as it targets only skewed caches.

The result is revealed in Table 2. When the LLC is statically randomized without dynamic remapping, all algorithms work as expected. GE presents the highest success rate thanks to its iterative structure and strong tolerance to noise. As described in Section 4.3 and [10], cyclic remapping per every 10 evictions per block (EV10) is specially designed to thwart CT and limit the window available for attackers to utilize the potentially found eviction sets. By applying EV10, GE and the original CT fail to work. The success rates of all other algorithms drop significantly but they still survive.

Both PPP and the optimized CT variants (CT-fast and W+W) incur unbalanced amount of evictions on the targeted cache set, which is prone to be observed by the active



(a) Threshold h = 5.0 (b) Eight extra addresses

Fig. 14. The success rate of CT-fast with various number of extra addresses (a) and different detect threshold (b). (each test is repeated by 10K times).

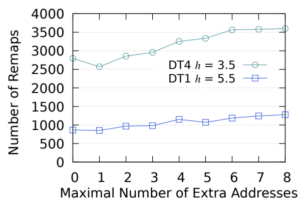


Fig. 15. Number of remaps triggered by running 1000 rounds of CT-fast.

TABLE 3
FPGA area breakdown at 75MHz.

		Slice	Percent	Overhead	RAM	Percent
Original	Rocket-Chip	24287			354.5	
	LLC	3114	12.8%		272.5	76.7%
	Channels	802	3.30%		0	
	MSHRs	823	3.39%		0	
	Request Queue	183	0.75%		0	
	Banked Store	1099	4.53%		256	71.9%
	Directory	392	1.61%		16.5	4.63%
Random	Rocket-Chip	28115		15.8%	356	
	LLC	5609	20.0%	80.1%	274	77.0%
	Channels	1076	3.83%	34.2%	0	
	MSHRs	910	3.24%	10.6%	0	
	Request Queue	311	1.11%	69.9%	0	
	Banked Store	1641	5.84%	49.3%	256	71.9%
	Directory	540	1.92%	37.8%	16.5	4.63%
	Hasher	526	1.87%		0	
	Detector	362	1.29%		1.5	0.42%
	Remapper	66	0.23%		0	
	PFC	485	1.73%		0	

detector described in Section 4.4. Nearly all algorithms cease to work when an active detector acts alone. CT-fast is the only survivor but its success rate drops to 1.0% when the monitoring period is four accesses per cache set (DT4) and further down to just 0.1% by reducing the period to one access per cache set (DT1). Although this result seems good enough, we believe EV10 is still necessary as it acts as a fail-safe way to limit the available time window even if eviction sets are still found by a future unknown algorithm. In addition, combining EV10 with active detection crashes the remaining survival chance of CT-fast. The success rate decreases to 0.3% with EV10+DT4 and nil with EV10+DT1.

CT-fast seems to be the most indomitable algorithm. The one analyzed in Table 2 is the basic form where each test collects only W addresses and fails if a postmortem test finding that they do not form an eviction set. As indicated by [21], extending the search for extra addresses when postmortem tests fail significantly increases the success rate. Fig. 14a demonstrates the strength of various detector combinations against the full-blown CT-fast algorithm with 0 (basic form) to 8 extra addresses. Without dynamic remapping (static), the success rate is raised to 61% by 8 extra addresses. EV10 alone only reduces the success rate but fails to stop it. The active detector ($h = 5$) is still effective, because the postmortem test can be considered as a utilization of the eviction set, which unavoidably asserts extra amount of eviction to the target cache set and makes the attack exposed. DT4 is able to reduce the success rate to ∼1.3% while applying EV10+DT1 presses it to ∼0.01%. Fig. 14b further reveals the drop of success rate when reducing the detector threshold h for EV10+DT4 and EV10+DT1. The result shows that EV10-DT1 is strong against the full-blown CT-fast algorithm. The success rate is merely 0.2% when h is set to as high as 11. When h is reduced to 5.5 for EV10+DT1, CT-fast ceases to work. Similarly, EV10-DT4 is also effective when $h = 3.5$. Non-skewed set-associative caches can be sufficiently strengthened against all fast algorithms, including the full-blown CT-fast.

To analyze the rate of false-positive and false-negative errors during active attacks, Fig. 15 reveals the number of remaps triggered by DT4 or DT1 during 1000 rounds of CT-fast with various number of extra addresses. It is shown that 850∼1250 remaps are triggered by DT1 while it is 2500∼3600 by DT4. DT4 leads to significant amount of false-positive errors. The monitoring window of DT4 is four times wide the window of DT1, which makes DT4 more sensitive to noise caused by non-representative but uneven cache evictions than DT1. However, a wider monitoring window makes it more difficult for any attack to evade detection by intentionally prolonging the search. Compared with false-positive errors, false-negative error is more important. It seems that 15% of the attacks evade from DT1. However, the success rate of CT-fast without remapping is only 13.9% (Table 2), most of the seemingly evasive attacks actually fail to find eviction sets and, therefore, do not trigger the DT1 detector. The real false-negative rate is far smaller.

6.3 Performance Overhead

The area overhead is illustrated in Table 3 and Table 4 for FPGA and ASIC, respectively. Unlike the 22% storage (RAM) overhead incurred by MIRAGE [11], only 1.9% of extra storage is introduced to the LLC in our (ASIC) implementation. Cache randomization leads to some observable overhead in logic area, which is not evaluated by any other designs due to the lack of hardware implementation and is therefore significantly under-estimated, as discussed in Section 3. In FPGA, the logic area of the dual-core Rocket-Chip rises by 15.8%, while the LLC is increased by 80.1%. When both logic and RAM area is considered together in ASIC, the area of the Rocket-Chip rises by 2.84%, where the area of the LLC increases by 3.34%. We argue this area overhead is reasonably small. The single-issue in-order Rocket core is tiny compared with a high-performance out-of-the-order (OoO) core. According to the area ratio and geometry information provided in [38], [39], the area ratio between a Rocket core and a six-issue BOOM OoO core is estimated around 1:17. The overall area overhead is therefore reduced to only 0.6%. The area of the LLC in the original Rocket is also under-estimated, because it is a shared L2 cache with only basic coherence support. There is no private L2 as in Intel processors, no performance counter and no hardware prefetcher. Considering all these reasons, the area overhead of randomizing a traditional set-associative LLC in high-performance OoO processors is marginal.

For the circuit speed, the original Rocket can run up to 79.4MHz and 162.1MHz on FPGA and ASIC, respectively,

TABLE 4
Breakdown of ASIC post-synthesis area ($k\mu m^2$) at 150MHz.

		Logic	RAM	Total	Percent	Overhead
Original	Rocket-Chip	755.9	4713	5469		
	LLC	106.2	4175	4282	78.3%	
	Channels	64.86	0	64.86	1.19%	
	MSHRs	8.287	0	8.287	0.15%	
	Request Queue	16.79	0	16.79	0.30%	
	Banked Store	7.544	3991	3999	73.1%	
	Directory	3.480	184.4	187.9	3.44%	
Random	Rocket-Chip	833.7	4790	5624		2.84%
	LLC	171.4	4253	4425	78.7%	3.34%
	Channels	68.39	0	68.39	1.22%	5.44%
	MSHRs	11.98	0	11.98	0.21%	44.60%
	Request Queue	22.74	0	22.74	0.40%	35.45%
	Banked Store	14.12	3991	4005	71.2%	0.16%
	Directory	6.154	240.9	247.0	4.39%	31.49%
	Hasher	16.71	0	16.71	0.30%	
	Detector	10.56	21.21	31.78	0.56%	
	Remapper	0.750	0	0.750	0.01%	
	PFC	9.528	0	9.528	0.17%	

while it is 80MHz and 160.8Mhz for the randomized one on FPGA and ASIC, respectively. The critical path in the original Rocket starts from the channel input modules in the LLC to the directory. According to Fig. 4, the hasher is added between the original input modules and the modules that are pushed forward. As a result, it acts as a manual retiming on the critical path which borrows a small amount of time from the newly added cycle by the hasher. The maximal frequency of the FPGA implementation is slightly improved. As on ASIC, the extra logic introduced into the LLC increases the wire density. The wire load model is therefore stressed, resulting in a tiny frequency drop. The hasher latency is around 4.91ns on ASIC when the clock period is set at 6.20ns.

Fig. 16 demonstrates the runtime performance overhead of running SPEC CPU 2006 benchmark on the randomized Rocket-Chip on FPGA. Note that all remaps triggered by the detectors are false-positive errors as there is no active attack. EV10+DT4 and EV10+DT1 incur marginal overhead on the average MPKI of 0.80% and 1.11%, respectively, and EV10 actually reduces MPKI by 0.11%. A remap is triggered by EV10 when high amount of evictions are caused by a memory access pattern showing low temporal locality. The active detector (DT4 and DT1) triggers a remap when the mapping of cache sets is uneven and one cache set suffers from high level of conflict misses. In both scenarios, a remap asserts low performance overhead and may help regenerate a balanced mapping capable of reducing conflict misses, e.g., high remapping rate does not raise MPKI for 403.gcc, 437.leslie3d, and 471.omnetpp. EV10, EV10+DT4 and EV10+DT1 lead to a tiny rise of average CPI by 0.49%, 0.70% and 0.89%, respectively. The major contributor here is the extra one cycle latency added to the LLC, which itself incurs a 0.4~0.8% CPI increase. Remaps do not always result in CPI overhead as they may produce in a good mapping which reduces cache miss rate. As an example, the CPI of 450.soplex actually reduces. Comparing between DT4 and DT1, although less remaps are triggered by DT1, it leads to slightly higher MPKI and CPI.

The power overhead is slightly higher than MPKI and CPI. For all benchmark cases and detector combinations, the power overhead against the non-randomized case ranges from 1% to 4%, as shown in Fig. 16, while the average overhead is around 2.1%. Although active attack detection and dynamic remapping are computation intense operations incurring extra power consumption, they are minor contributors for the overall power overhead. As an evidence, Fig. 17 reveals the runtime power curve when a Rocket core is constantly writing memory while the LLC is remapped at 10 Hz. The clearly visible power peaks are caused by the remap operations, which always finish in 2ms and incur extra energy of 200μJ. The average remap frequency is around 1.83Hz, resulting in a trivial power overhead of 0.023%. A power analysis has been done for the randomized LLC on ASIC. LLC traces are collected from running SPEC CPU 2006 on Spike and fed into a post-synthesis simulation, which provides the switching activities for power estimation. The power of the the non-randomized LLC is 1.34W, where logic and RAM consume 15mW and 1.32W, respectively. For the randomized LLC, the power of logic and RAM rises by 55% and 1.9%, respectively. The overall power overhead is small, only around 2.5%.

It is also important to evaluate the performance overhead of running parallel applications on multicore processors. Due to the limited resource, we can put only two cores on a single FPGA. Instead, we have configured the same Spike simulator used in [10] with four cores and simultaneously run four out of six representative cases chosen from the SPEC CPU 2006 benchmark. Using non-randomized Rocket as the baseline, the MPKI overhead is demonstrated in Fig. 18. For all detector combinations,

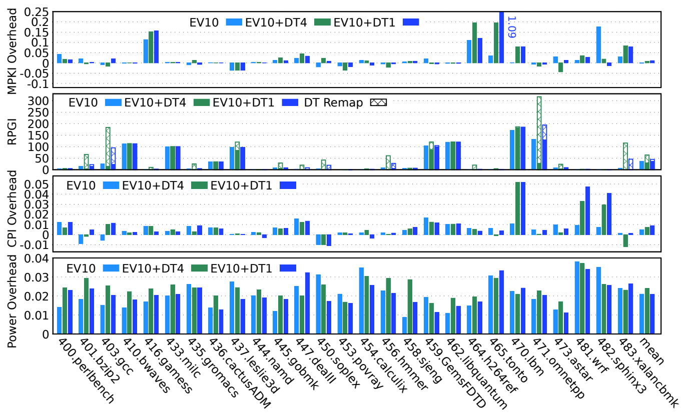


Fig. 16. Performance overhead of running SPEC CPU 2006 benchmark. MPKI overhead, CPI overhead and power overhead use the non-randomized Rocket as the baseline. All results, including the remap per G instructions (RPGI), are collected from running 10G instructions. The detector threshold is set to 3.5 and 5.5 for DT4 and DT1, respectively.

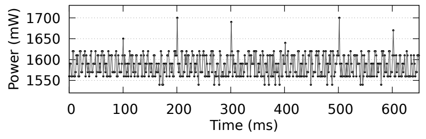


Fig. 17. The power peaks due to remap per 100ms.

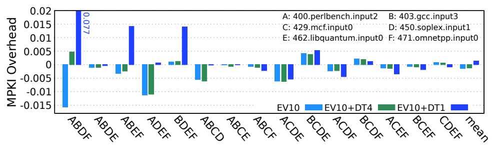


Fig. 18. MPKI overhead when running four SPEC CPU 2006 benchmarks simultaneously on a 4-core Spike simulator. The combinations of benchmarks are ordered by the average MPKI, from the lowest (ABDF, 3.54) on the left to the highest (CDEF, 25.22) on the right.

the average MPKI overhead is trivial (within $\pm 0.2\%$), and this overhead is more consistent for the memory heavier benchmark combinations on the right. According to the results presented in Fig. 16, we strongly believe that the performance overhead of cache randomization on multicore processors is still marginal, similar with the performance overhead on the dual-core Rocket-Chip tested on FPGA.

7 Discussion

New randomized cache designs: Since the introduction of randomized LLC (CEASER [7]), several randomized caches have been proposed by the research community. Most of them are randomized skewed caches while none has been adopted commercially. CEASER-S [8] and ScatterCache [9] significantly increases the difficulty of finding eviction sets by introducing cache skews, but still fail to stop attackers from finding small but still useful partial eviction sets [10], [14]. MIRAGE [11] eliminates the attacker-controlled associativity evictions by over-providing meta and multi-stepped Cuckoo relocation, but asserts heavy area and runtime performance overhead. As a trade-off, Chameleon cache [15] enforces a single relocation for all evicted blocks using a victim cache. The performance overhead could still be substantial as it is equivalent to remap using an intolerable short period of ACC1.

New cache attacks: Recent conflict-based cache attacks begin to adopt accurate manipulation of the cache replacement state. Prime+Scope [21] significantly raises the speed of traditional Prime+Probe attacks by presetting a selected block in an eviction set to the eviction candidate before triggering the victim to access the cache set. Consequently, the victim access can be detected by re-accessing the selected block rather than the whole eviction set. Similarly, Reload+Refresh [40] presets the victim's block to the eviction candidate; therefore, victim's access is stealthily observable by checking whether the victim's block is still the eviction candidate (not if accessed). During the manipulation of the cache replacement state, both attacks unavoidably assert a large amount of accesses (evictions) to the targeted cache set, which is exactly the pattern monitored by our active detector. Occupancy attacks [41] are a new type of cache attack which does not targeting a certain cache set but the whole LLC. A recent attack is able to build a stochastic channel [42] on randomized skewed caches by priming a small number of cache sets. Since these attacks control the amount rather than the location of the incurred cache conflicts, cache randomization is ineffective while cache partitioning is the preferred defense. Supporting way-based cache partitioning, such as Intel CAT, is inefficient due to the reduced number of ways in each partition. Instead, a recent addition to ScatterCache supports encryption based cache set partition [16]. As for our implementation, way-based cache partitioning is naturally supported as the set-associative layout is untouched.

Attack detection: Runtime detection of cache side-channel attacks using PFCs [43], [44] has shown to be effective to detect persistent attacks. The concentration of cache accesses on the target cache sets during the exploitation phase has long been discovered [43]. Recent hardware detectors with set level granularity begins to utilize this pattern [45]. To the best of our knowledge, we are the first to exploit the unique set distribution of cache evictions during the preparation phase of an attack (the search for eviction sets).

8 Conclusion

For the first time, a dynamically randomized set-associative cache has been implemented in the LLC of a Linux capable multicore processor. By randomizing the mapping of addresses to cache sets, periodically remapping the cache layout, and triggering extra remaps when cache attacks are detected in action, the set-associative LLC has been sufficiently strengthened to thwart all existing fast algorithms for searching eviction sets. The added cache randomization incurs only marginal runtime performance overhead.

Acknowledgement

This work was partially supported by the National Natural Science Foundation of China under grant No. 61802402 and 62172406, the CAS Pioneer Hundred Talents Program. Any opinions, findings, conclusions, and recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks," in *Proc. Int'l Symp. Comput. Archit.*, 2017, pp. 347–360.

[2] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symp. Security and Privacy*, May 2015.

[3] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES," in *Proc. Symp. Security and Privacy*, May 2015, pp. 591–604.

[4] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *Proc. Int'l Conf. Applied Cryptography and Network Security*, 2018, pp. 83–102.

[5] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript," in *Proc. Int'l Conf. Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016, pp. 300–321.

[6] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *Proc. USENIX Annu. Technical Conf.*, 2017, pp. 299–312.

[7] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proc. IEEE/ACM Int'l Symp. Microarch.*, 2018, pp. 775–787.

[8] ——, "New attacks and defense for encrypted-address cache," in *Proc. Int'l Symp. Comput. Archit.*, 2019, pp. 360–371.

[9] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *Proc. Security Symp.*, 2019, pp. 675–692.

[10] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! But we can fix it," in *Proc. IEEE Symp. Security and Privacy*, May 2021, pp. 955–969.

[11] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *Proc. USENIX Security Symp.*, Aug. 2021, pp. 1379–1396.

[12] P. Vila, B. Köpf, and J. Morales, "Theory and practice of finding eviction sets," in *Proc. IEEE Symp. Security and Privacy*, 2019.

[13] W. Song and P. Liu, "Dynamically finding minimal eviction sets can be quicker than you think for side-channel attacks against the LLC," in *Proc. Int'l Symp. Research in Attacks, Intrusions and Defenses*, 2019, pp. 427–442.

[14] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "CaSA: End-to-end quantitative security analysis of randomly mapped caches," in *Proc. IEEE/ACM Int'l Symp. Microarch.*, Oct. 2020.

[15] T. Unterluggauer, A. Harris, S. Constable, F. Liu, and C. Rozas, "Chameleon cache: Approximating fully associative caches with random replacement to prevent contention-based cache attacks," in *Proc. Int'l Symp. Secure and Private Exec. Environ. Design*, 2022.

[16] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer *et al.*, "Scatter and split securely: Defeating cache contention and occupancy attacks," in *Proc. Symp. on Security and Privacy*, 2023.

[17] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin *et al.*, "The Rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[18] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[19] A. Purnal and I. Verbauwhede, "Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache," *CoRR*, 2019, abs/1908.03383.

[20] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *Proc. IEEE Symp. Security and Privacy*, May 2021, pp. 987–1002.

[21] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks," in *Proc. ACM SIGSAC Conf. Comput. and Commun. Security*, Nov. 2021, pp. 2906–2920.

[22] J. P. Thoma and T. Güneysu, "Write me and I'll tell you secrets — write-after-write effects on Intel CPUs," in *Proc. Int'l Symp. on Research in Attacks, Intrusions and Defenses*, Oct. 2022.

[23] A. Seznec, "A case for two-way skewed-associative caches," in *Proc. Annu. Int'l Symp. Comput. Archit.*, May 1993, pp. 169–178.

[24] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. S. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. Int'l Symp. Comput. Archit.*, Jun. 2010, pp. 60–71.

[25] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro, "BRUTUS: Refuting the security claims of the cache timing randomization countermeasure proposed in CEASER," *IEEE Comput. Archit. Letters*, vol. 19, no. 1, pp. 9–12, Jan. 2020.

[26] M. Sugiyama, "2.5: Transformation of random variables," in *Introduction to Statistical Machine Learning*, Boston, 2016, pp. 22–23.

[27] J. S. Hunter, "The exponentially weighted moving average," *Journal of Quality Technology*, vol. 18, no. 4, pp. 203–210, 1986.

[28] SiFive, Inc., "block-inclusivecache-sifive," Apr. 2021, https://github.com/sifive/block-inclusivecache-sifive.

[29] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *Proc. Int'l Symp. Research in Attacks, Intrusions, and Defenses*, Nov. 2015, pp. 48–65.

[30] N. Hua, E. Norige, S. Kumar, and B. Lynch, "Non-crypto hardware hash functions for high performance networking ASICs," in *Proc. ACM/IEEE Symp. Archit. for Networking and Commun. Systems*, Oct. 2011, pp. 156–166.

[31] G. Metze, "Minimal square rooting," *IEEE Trans. Elec. Computers*, vol. EC-14, no. 2, pp. 181–185, 1965.

[32] Nangate, "45nm open cell library," California, USA, 2008.

[33] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. IEEE/ACM Int'l Symp. Microarch.*, Dec. 2007, pp. 3–14.

[34] SiFive, Inc., "SiFive FU740-C000 manual v1p6," Jan. 2022.

[35] Z. Xue, D. Xie, and W. Song, "Hardware performance counter based on RISC-V," *Computer Systems & Applications*, vol. 30, no. 11, pp. 3–10, 2021, (in Chinese).

[36] A. Waterman, S. Johnson, C.-M. Chao, Y. Chen, Y. Lee *et al.*, "Spike RISC-V ISA simulator," https://github.com/riscv/riscv-isa-sim.

[37] D. Ye, J. Ray, and D. R. Kaeli, "Characterization of file I/O activity for SPEC CPU2006," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 112–117, 2007.

[38] C. Celio, D. A. Patterson, and K. Asanović, "The berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor," EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2015-167, Jun. 2015.

[39] Y. Xu, Z. Yu, D. Tang, G. Chen, L. Chen *et al.*, "Towards developing high performance RISC-V processors using agile methodology," in *Proc. IEEE/ACM Int'l Symp. Microarch.*, Oct. 2022, pp. 1178–1199.

[40] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: abusing cache replacement policies to perform stealthy cache attacks," in *Proc. USENIX Security Symp.*, Aug. 2020, pp. 1967–1984.

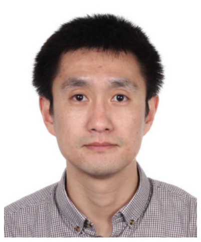
[41] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal *et al.*, "Robust website fingerprinting through the cache occupancy channel," in *Proc. USENIX Security Symp.*, Aug. 2019, pp. 639–656.

[42] T. Verma, A. Anastasopoulos, and T. M. Austin, "These aren't the caches you're looking for: Stochastic channels on randomized caches," in *Proc. IEEE Int'l Symp. Secure and Private Execution Environ. Design*, Sep. 2022, pp. 37–48.

[43] Y. Zhang and M. K. Reiter, "Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proc. Conf. Comput. and Commun. Security*, 2013, pp. 827—-838.

[44] J. Chen and G. Venkataramani, "CC-Hunter: Uncovering covert timing channels on shared processor hardware," in *Proc. IEEE/ACM Int'l Symp. Microarch.*, 2014, pp. 216–228.

[45] A. Harris, S. Wei, P. Sahu, P. Kumar, T. M. Austin, and M. Tiwari, "Cyclone: Detecting contention-based cache information leaks through cyclic interference," in *Proc. IEEE/ACM Int'l Symp. Microarch.*, 2019, pp. 57–72.

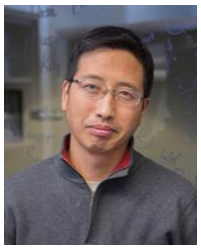


Wei Song is an Associate Professor at the Institute of Information Engineering, CAS. He received his Ph.D. from the University of Manchester, and had worked as a postdoctoral researcher in the University of Manchester and the University of Cambridge. His current research focuses on the security enhancement of computer architectures, such as the defenses for cache side channel and control-flow hijacking attacks.

Zihan Xue is a Ph.D. student at the Institute of Information Engineering, CAS. He received BE and ME degree from Southwest Jiaotong University and had worked as assistant engineer at CRSC Research & Design Institute Group Co., Ltd. His current research focuses on secure computer architectures.

Jinchi Han received his B.S. degree from Shandong University in 2022. He is currently pursuing a Master degree at the Institute of Information Engineering, CAS. His current research focuses on secure computer architectures.

Zhenzhen Li received her B.S. in Electronic Science and Technology from University of Information Engineering, China and received her M.S. in Electronic and Information Engineering from Beijing University of Posts and Telecommunications, China. She is currently working toward a Ph.D. degree in computer architecture at the Institute of Information Engineering, CAS. Her interests focus on computer security, such as the security analysis of cache side channel attacks.



Peng Liu received his BS and MS degrees from the University of Science and Technology of China, and his PhD from George Mason University in 1999. Dr. Liu is the Raymond G. Tronzo, MD Professor of Cybersecurity, and Director of the Cyber Security Lab at Penn State University. His research interests are in all areas of computer security. He has published over 350 technical papers. He has served on over 100 program committees and reviewed papers for numerous journals. He is currently the Co-Editor-in-Chief of Journal of Computer Security.