PhantomCache: Obfuscating Cache Conflicts with Localized Randomization

Qinhan Tan*
Zhejiang University
tanqinhan@zju.edu.cn

Zhihua Zeng*
Zhejiang University
zengzhihua@zju.edu.cn

Kai Bu*
Zhejiang University
kaibu@zju.edu.cn

Kui Ren
Zhejiang University
kuiren@zju.edu.cn

***Abstract*—Cache conflicts due to deterministic memory-to-cache mapping have long been exploited to leak sensitive information such as secret keys. While randomized mapping is fully investigated for L1 caches, it still remains unresolved about how to secure a much larger last-level cache (LLC). Recent solutions periodically change the mapping strategy to disrupt the crafting of conflicted addresses, which is a critical attack procedure to exploit cache conflicts. Remapping, however, increases both miss rate and access latency. We present PhantomCache for securing an LLC with remapping-free randomized mapping. We propose a localized randomization technique to bound randomized mapping of a memory address within only a limited number of cache sets. The small randomization space offers fast set search over an LLC in a memory access. The intrinsic randomness still suffices to obfuscate conflicts and disrupt efficient exploitation of conflicted addresses. We evaluate PhantomCache against an attacker exploring the state-of-the-art attack with linear-complexity. To secure an 8-bank 16 MB 16-way LLC, PhantomCache confines randomization space of an address within 8 sets and brings only 1.20% performance degradation on individual benchmarks, 0.50% performance degradation on mixed workloads, and 0.50% storage overhead per cache line, which are 2x and 9x more efficient than the state-of-the-art solutions. Moreover, PhantomCache is solely an architectural solution and requires no software change.**

I. Introduction

Cache conflicts have long been exploited to leak sensitive information such as secret keys. Although memory isolation among processes can be enforced for the sake of security, caches are still shared among different processes. A cache access of a process may cause a cache hit or miss to another process accessing the cache. A cache hit and a cache miss differ in access latency, the timing of which can be observed by an attacker so as to infer the access behavior of a victim process. Consider the AES algorithm for example. A single bit of the secret key may determine whether a branch is executed. This leads to different cache access behaviors and therefore different cache timings. Through a timing side channel attack, an attacker can infer secret AES keys in use [5], [19], [26], [33], [44], [48].

Last level caches (LLCs) are the main target of conflict-based cache timing attacks. This is because an LLC is shared among all cores and it is difficult to prevent cache conflicts between attacker processes and victim processes in the LLC. This sharing feature of LLCs also brings severe risk of information leakage in web applications and cloud services. For example, an attacker is able to use unprivileged Javascript code to obtain secret information of a victim [33]. When the victim's browser runs the Javascript code, the attack is launched and the victim's cache access behaviors are monitored. The snooped access behaviors then can be used to infer secret information. Moreover, cache conflicts in an LLC also allow attackers to build robust cache covert channels in cloud environment. Maurice *et al.* [30] succeeded in building LLC-based covert communication channels between processes on Amazon EC2. The covert channel supports a transmission rate of more than 45 KBps and a 0% error rate even in the presence of high system activity.

Most countermeasures against conflict-based cache timing attacks [6], [15], [23], [28], [39], [41], [42], [46] either fall short of strong security or sacrifice system functionality. For example, detection-based solutions [8], [9], [13], [46], [47] may simply consider frequent cache misses of specific addresses as suspicious and trigger an event alarm. Such solutions, however, are usually threshold based and vulnerable to false negatives. To prevent conflict-based attacks, cache partition [23], [41], [42] and fuzzy time [39] break the premises of cache sharing and time measurements, respectively. While throttling attacks, they affect system functionality in terms of lower cache space utilization and inaccurate time measurements for normal processes.

As a fundamental countermeasure, randomized mapping aims to break deterministic cache conflicts by randomly mapping a memory block to a cache location [25], [42], [43]. This method does not reduce cache conflicts. Instead, it improves the difficulty of observing and exploiting cache conflicts. Once the placement policy is randomized, the memory-to-cache mapping for any address is not fixed. In this scenario, the attacker can hardly find and exploit addresses that may conflict with the victim addresses. This is because with the existence of randomness, the attacker cannot create cache conflicts with certainty. While efficient randomized-mapping on small L1 caches have been fully investigated [24], [25], [42], [43], it still awaits efficient implementation on a much larger last-level cache (LLC). The major challenge is how to

*Qinhan Tan and Zhihua Zeng have contributed equally and are considered to be co-first authors.

⋆Kai Bu is the corresponding author.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24086
www.ndss-symposium.org

guarantee fast lookup for a block that may be randomly mapped to anywhere in a large LLC. It is impractical to enforce a full-scale search across the entire LLC. Previous solutions for LLCs resort to indirect randomized mapping. They first introduce implicit mapping to lengthen the time for finding conflicted addresses. Then they conduct dynamic remapping to change the mapping strategy from time to time so as to bring randomness. They, however, have recently been found insecure against the state-of-the-art attacking algorithm with linear complexity [35], [40], [45]. To prevent the state-of-the-art attack, these countermeasures need an extremely frequent remapping, which will cause unacceptable performance overhead [35].

Skewed-cache based designs provide a partial solution to the above problem. Such designs scatter the cache lines in a cache set into different cache partitions, and adopt random placement among these partitions [35], [45]. The uncertainty in memory-to-cache mapping increase the cost of the attack, because an attacker needs to repeat the old procedures for many times to succeed with high probability. However, since the total number of possible cache locations of a physical address is not enhanced, these designs still need the inefficient dynamic remapping. To secure a 2 MB 16 way LLC against the linear-complexity attack, the state-of-the-art ScatterCache [45] brings 2% slowdown and 5% storage overhead per cache line, requiring both hardware and software changes.

In this paper, we present PhantomCache, an LLC-favorable countermeasure against conflict-based cache timing attacks without inefficient dynamic remapping. It leverages our newly proposed localized randomization to mitigate conflict-based cache timing attacks. In contrast with global randomization, localized randomization restricts the randomization space to a small range of cache locations. Each time a memory block enters the cache, it is randomly placed at a location from its fixed mapping range. Because the mapping range is small, all of the locations in it can be checked in parallel during a cache access to search for the needed memory block. This is more practical and efficient than global randomized mapping. Although the randomness is limited, PhantomCache is still sufficient to prevent conflict-based cache timing attacks.

On a 16 MB 16-way LLC where PhantomCache maps an address to one of 8 random locations, our analysis shows that an attacker using the linear-complexity attack algorithm needs 500+ years to succeed. PhantomCache achieves such strong security without dynamic remapping. This is because PhantomCache preserves a sufficiently large randomization space by randomly mapping an address across several sets from the entire LLC. In contrast, the number of possible mapping locations for an address by skewed-cache based solutions is equal to the number of locations in only one set of the undivided cache. This is why they need dynamic remapping to preserve security at the cost of efficiency [35], [45].

In summary, we make the following contributions to mitigating conflict-based cache timing attacks.

- We propose the localized randomization technique that can protect large caches (such as LLCs) against conflict-based cache timing attacks without inefficient dynamic remapping (Section III).
- We explore a series of hardware-efficient design strategies to realize localized randomization in PhantomCache. The mapping function to realize localized randomization only imposes one clock cycle latency per cache access (Section IV). Our analysis shows that these efficient design strategies do not compromise the security (Section V).
- We propose an efficient design to integrate PhantomCache into the multi-banked LLC architecture (Section IV-E). This improves parallelism and reduces the overhead of cache access.
- We implement PhantomCache using the ChampSim simulator and validate its performance using extensive SPEC CPU 2017 benchmarks. We evaluate PhantomCache implementation on an 8-bank 16 MB 16 way LLC. It brings only 1.20% performance degradation on individual benchmarks, 0.50% performance degradation on mixed workloads, and 0.50% storage overhead per cache line, which are 2x and 9x more efficient than the state-of-the-art skewed-cache based solutions. Moreover, PhantomCache requires no software change (Section VII-F).

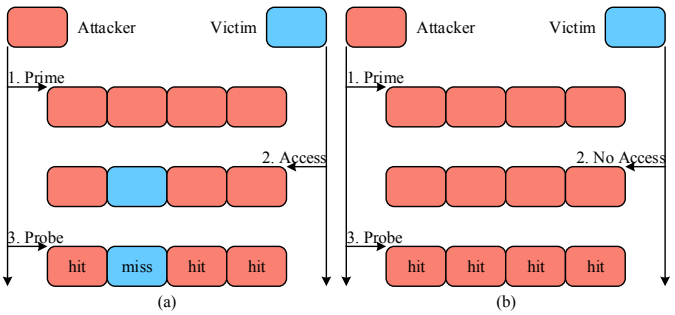

Fig. 1. Prime+Probe attack [26], [30] infers (a) the victim access upon a cache miss and (b) the victim no-access upon all cache hits in a 4-way cache set.

II. Problem

In this section, we review the basics of conflict-based cache timing attacks and underline the inefficiency of prior countermeasures based on randomized mapping.

A. Conflict-based Cache Timing Attack

Conflict-based cache timing attacks exploit cache conflicts to reveal the cache access behavior of processes [40]. To infer whether the victim has accessed a memory address, the attacker need craft another memory address that maps to the same cache line with the memory address of interest. The attacker then periodically accesses the crafted address. The first access makes the corresponding data block cached. For the next access, a cache hit indicates that the cached block is not replaced. The attacker makes sure that the victim has not accessed the memory address of interest in between its two accesses. Otherwise, a cache miss occurs and it reveals the memory access behavior of the victim. Since modern processors use set associative caches, the attacker can only deduce which set a memory address maps to rather than the exact cache line. Therefore, the attacker has to craft a set of addresses that map to an entire cache set, which is called an eviction set. Periodical access and timing inference then involve all or most of the addresses in the eviction set.

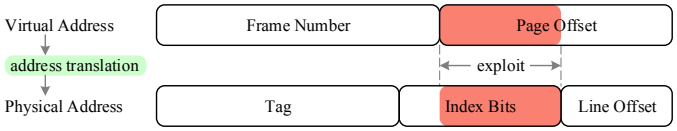

Fig. 2. Memory address format: virtual address and physical address.

Prime+Probe [26], [30], a representative conflict-based attack, exercises eviction addresses through first priming and then probing a cache set. During the wait interval in between, the victim may or may not access the cache set. We enumerate both cases in Figure 1. In Figure 1(a), the attacker first primes the entire cache set by accessing addresses in the eviction set. While the attacker waits, the victim accesses a memory address that maps to the primed cache set. This leads to a cache replacement over, for example, the second cache line in Figure 1(a). When the wait interval terminates, the attacker probes the cache set by accessing eviction addresses again. Since the second cache line is replaced by the victim, the attacker encounters a cache miss there and infers the access action of the victim. In contrast, if the victim has not accessed the cache set (Figure 1(b)), the attacker experiences all cache hits upon probing and infers the victim's non-access action. Note that Evict+Reload [16], [22] jointly exploits cache conflicts and shared blocks between the attacker and victim processes. It is therefore usually handled by countermeasures involving shared memory, say forbidding shared memory between Virtual Machines (VMs) [38].

B. Minimal Eviction Set

A successful conflict-based attack relies on a minimal eviction set that satisfies two properties—identical mapping and minimal cardinality [26], [40]. First, identical mapping requires that all eviction addresses map to the same cache set. As shown in Figure 1, if some eviction addresses map to other cache sets, they may still cause cache misses to the attacker even though the victim does not access the example cache set. This fails the Prime+Probe attack. Second, minimal cardinality requires that the number of eviction addresses be minimized to set associativity. An eviction-set cardinality less than set associativity is insufficient for priming an entire cache set. On the other hand, if the eviction set contains more addresses than a cache set can hold, they lead to cache conflicts among themselves and cause cache misses irrelevant to the victim access. This fails the Prime+Probe attack as well. To satisfy identical mapping and minimal cardinality, the algorithmic essence for finding a minimal eviction set features two respective core building blocks [40]. One is to initialize an eviction set with candidate addresses that likely map to the same cache set. The other is to iteratively remove addresses whose absence will not make the remaining eviction set ineffective.

Candidate address sampling. For sampling initial candidate addresses, the attacker exploits the deterministic memory-to-cache mapping strategy on modern processors. That is, the index bits of a physical address are directly used as the index of the cache set that the address maps to. What makes it exploitable is that part or even all of the index bits can be overlapped by a virtual address and its corresponding physical address. As shown in Figure 2, a virtual address consists of a frame number and a page offset while its corresponding physical address consists of a tag, index bits, and a line offset. Using system configuration parameters, it is straightforward to calculate the length of each field. The length difference of the page offset and the line offset decides how many index bits are shared by a virtual address and its corresponding physical address. The longer the page offset the more likely it facilitates inferring which cache set a virtual address can finally access. In particular, microarchitecture with large pages is more vulnerable to this exploit. For example, a Sandy Bridge processor released by Intel uses the 6th to 16th least significant bits of a physical address as index bits; a 2 MB page in use results in a 21-bit page offset, which covers the complete index bits [26]. This reveals to the attacker exactly which cache set a virtual address maps to.

Address removal. Algorithm 1 presents a typical iterative framework for finding a minimal eviction set with $\mathcal{O}(|E|^2)$ complexity [33], where $|E|$ denotes the cardinality of the initial set E. In each iteration, the attacker determines whether a candidate address e in the initial set E can be removed (lines 1-11). Removing e should not make the remaining E insufficient for evicting x from the cache. Consider, for example, when the current E is sufficient for priming the cache set that x maps to. After priming the cache set (line 2), accessing x encounters a cache miss with a relatively large delay t_1 (line 3). Now we remove a candidate address e at random and access all the remaining candidate addresses (lines 5-6). If they can no longer evict x, accessing x again will return a cache hit with a relatively low delay t_2 (lines 7-8). Such a miss-then-hit sequence of accessing x can be determined by a noticeable timing difference (line 8). In this case, the removed candidate address e should be added back to E (line 9). Otherwise, the two accesses of x in the same iteration show comparative timings, the attacker can remove the selected candidate address e. The iteration ceases upon the minimal eviction set contains as many candidate addresses as set associativity.

The state-of-the-art eviction set minimization algorithm [35], [40] achieves $\mathcal{O}(|E|)$ complexity using group testing [11]. Consider an m-way associative cache that requires an minimal eviction set of size $|E| = m$ to attack. In each iteration, the algorithm first splits the eviction set into $m + 1$ groups. It removes addresses group wise and tests whether accessing the remaining groups of addresses can still evict the target cache line. If yes, the group of addresses are removed from the eviction set. Otherwise, they are brought back to the eviction set. Since the minimal eviction set requires m addresses, these m addresses reside in at most m groups. Therefore, at least one of the $m + 1$ groups addresses can be removed in each iteration. This makes the minimization process converge much faster than the $\mathcal{O}(|E|^2)$ algorithm that removes addresses one by one.

C. (Inefficient) Randomized Mapping for LLC

Randomized memory-to-cache mapping has been explored as a fundamental countermeasure against conflict-based cache timing attacks. Ideally, randomized mapping obfuscates cache conflicts by mapping a physical address to a random cache line [25], [43]. Even for the same address, its consecutive placements will highly likely occupy different cache lines. This makes it hard to find a set of addresses that always map to the same cache set. Randomized mapping thus prevents the

Algorithm 1 Minimal Eviction Set Construction [33]

Input: Initial eviction set E; Target address x;
Output: Minimal Eviction Set E to Evict x;

while $|E| > SetAssociativity$ **do**
  Access all candidate addresses of E;
  $t_1 \leftarrow$ measure the time it takes to access x;
  $e \leftarrow$ select a candidate address at random from E;
  $E = E \setminus \{e\}$;
  Access all candidate addresses of E;
  $t_2 \leftarrow$ measure the time it takes to access x;
  if $t_1 - t_2 > threshold$ **then**
    $E = E \cup e$;
  end if
end while
// iteration terminates upon $|E| = SetAssociativity$;
return E;

TABLE I. COMPARISON OF PHANTOMCACHE WITH EXISTING RANDOMIZED MAPPING SOLUTIONS FOR LLC.
NOTES: *RPCACHE, CEASER, CEASER-S, AND SCATTERCACHE ACHIEVE RANDOMIZED MAPPING THROUGH DYNAMIC REMAPPING. WITHIN A REMAPPING INTERVAL, MEMORY-TO-CACHE MAPPING IS ESSENTIALLY DETERMINISTIC. NOTE THAT DYNAMIC REMAPPING IS ALSO REFERRED TO AS RE-KEYING [45].
#RPCACHE SUPPORTS DIRECT CACHE SEARCH BY QUERYING THE ADDRESS MAPPING IN PERMUTATION TABLES. STORAGE OVERHEAD INDUCED BY LARGE PERMUTATION TABLES MAY LIMIT ITS USAGE IN LLC [34].

Legend: CF: Crypto-Free; DS: Direct Search; HO: Hardware Only; LA: LLC Applicability; DRF: Dynamic-Remapping Free

Solution	CF	DRF	DS	HO	LA
RPcache* [42]	✓	✗	✓	✗	✓#
CEASER* [34]	✗	✗	✓	✓	✓
CEASER-S* [35]	✗	✗	✓	✓	✓
ScatterCache* [45]	✗	✗	✓	✗	✓
PhantomCache	✓	✓	✓	✓	✓

attacker from finding minimal eviction sets, the foundation of conflict-based cache timing attacks. While efficient randomized mapping solutions for L1 caches have been proposed [24], [25], [43], solutions for securing LLCs still suffer from practical inefficiency [34], [35], [42], [45].

Random Replacement on L1 caches. It aims to fully achieve randomized mapping. The pioneering solution, NewCache [25], [43], randomly selects a cache line and replaces it with the accessed memory block upon a cache miss. Then an access enforces a search through all cache lines. Because of the small size of L1 caches, NewCache can guarantee fast search. Built upon NewCache, Random Fill Cache [24] further randomizes the selection of which block to put in cache. Specifically, it may not cache a requested block. Instead, the requested block is directly sent to the processor while a randomly fetched block adjacent to the demanded one from memory will be cached. Random Fill Cache requires both hardware and software changes. It suits better to applications that have random memory access patterns.

Dynamic Remapping on LLC caches. Since it is hard to enforce global search over a large LLC, solutions for protecting LLCs are double-edged in that 1) they first introduce implicit mapping to lengthen the time for finding eviction sets, and 2) then they conduct dynamic remapping to change the mapping strategy for defeating the attacker's accumulated inference. Implicit memory-to-cache mapping can be achieved by either permutation tables [42] or encryption units [34], [35], [45]. RPcache [42] uses permutation tables indexed by the index bits of an address. The indexed entry features the cache set index the address maps to. To address the storage overhead of permutation tables, CEASER [34] hides memory-to-cache mapping using encryption. It encrypts a physical address and uses the encryption result as the cache set index. However, once permutation rules [42] or encryption keys [34] are fixed, an address will always map to the same cache set. This deterministic mapping again leaves the door open for cache timing attacks. Toward a double defense over implicit mapping, dynamic remapping periodically changes the memory-to-cache mapping strategy. Remapping frequency should be sufficiently high such that the attacker cannot get enough time to find a minimal eviction set targeting a specific mapping. For example, to secure an 8 MB 16-way LLC against the $\mathcal{O}(|E|^2)$ attack with 0.50% performance overhead, where E is the average cardinality of initial sample addresses, CEASER needs to remap a line every 100 accesses.

As the $\mathcal{O}(|E|)$ attack emerges [35], [40], dynamic remapping has been augmented with skewed cache design to control overhead [35], [45]. A skewed cache divides the cache space into partitions [36]. Each partition contains a number of consecutive ways from all cache sets. For example, a two-way skewed-associative cache over a traditional 16-way cache divides the cache into two partitions, one contains the first 8 ways from all sets while the other contains the second 8 ways from all sets. The property for security leverage is that each partition has a different address mapping function. To evict an address, an attacker now needs to fill sets the address may map to on all partitions. Since a fixed mapping strategy of each partition is vulnerable, dynamic remapping is still needed. CEASER-S [35] extends CEASER under a skewed cache. Against the attack that requires that each eviction address should map to the same set with the victim address on all partitions, it leads to only 1% slowdown when remapping a line every 100 accesses. However, ScatterCache [45] finds that an attacker can more easily find eviction sets with partially-congruent addresses. In other words, each eviction address can map to the same set with the victim address on one or more partitions while all eviction addresses jointly cover all possible sets of the victim address. Apparently, this attack is more challenging to defend. The state-of-the-art ScatterCache can secure 2 MB 16 way LLC with 2% slowdown and 5% storage overhead per cache line while requiring both hardware and software changes [45].

In comparison, our PhantomCache does not require dynamic remapping or software change. To secure an 8-bank 16 MB 16-way LLC, it brings only 0.50% slowdown and 0.50% storage overhead per cache line. The shake-off of dynamic remapping is because PhantomCache randomly maps an address across several sets across the entire LLC. We efficiently integrate PhantomCache design into the multi-banked LLC architecture (Section IV-E). In contrast, the number of possible mapping location for an address by skewed-cache based solutions is equal to the number of locations in only one set of the undivided cache. This is why they need dynamic remapping to preserve security at the cost of efficiency [35], [45].

III. Overview

In this section, we present PhantomCache, an LLC-favorable countermeasure against conflict-based cache timing attacks without inefficient dynamic remapping (Table I). It guarantees efficiency by our newly proposed localized randomization technique. Localized randomization bounds randomized memory-to-cache mapping of an address within only a limited number of randomly selected cache sets. The intrinsic mapping randomization enables PhantomCache as effective against eviction set minimization as the fully randomized NewCache [25], [43]. Meanwhile, searching a block touches also the limited cache sets and can be efficiently implemented by a practical hardware supporting parallel access. This makes PhantomCache as LLC favorable as CEASER [34], CEASER-S [35], and ScatterCache [45]. However, PhantomCache relieves from frequent, inefficient dynamic remapping as it does not involve deterministic mapping.

A. Motivation

Essentially, if we could limit randomization space without sacrificing security of randomized mapping, we can achieve fast cache search. This instantly motivates us to explore localized randomization. That is, we randomize the mapping of an address within a limited number of cache sets, instead of across the entire cache. The intrinsic property of randomness hinders the attacker from minimizing an eviction set. Traditionally, the criterion for removing an eviction address is that the remaining eviction set can still prime the target cache set to evict the target address [33], [40]. It leverages the fact that, in traditional caches, an eviction set including at least m (i.e., set associativity) relevant addresses must be sufficient for priming the target cache set. However, localized randomization makes this criterion non-deterministic and hard to exploit. Consider n addresses, each of which maps to r cache sets but all share a common cache set. Given set associativity m, we derive the probability of priming the common cache set by accessing the n addresses as the following.

$$Pr_{\text{prime}}(n) = \sum_{i=m}^{n} C_n^i \times (\frac{1}{r})^i \times (\frac{r-1}{r})^{n-i}. \quad (1)$$

In this scenario, even if the eviction set includes m relevant addresses, accessing all its addresses cannot necessarily prime the target set. Thus, the attacker needs to repeat accessing the whole eviction set r^m times on average to figure out whether it includes enough relevant addresses. This increases the attacker's cost rapidly with the growth of r, which motivates us to explore more about localized randomization.

B. Methodology

Toward practically efficient obfuscation of cache conflicts, the key idea of localized randomization consists of two phases of randomness. First, for each address, we randomly select a predefined number of candidate sets for it. The selection uses the address and a random mapping function to compute cache set indices. Second, we randomly select one candidate set for mapping the address. As shown in Figure 3, PhantomCache enforces localized randomization through modified placement and search policies while it does not touch the replacement policy.

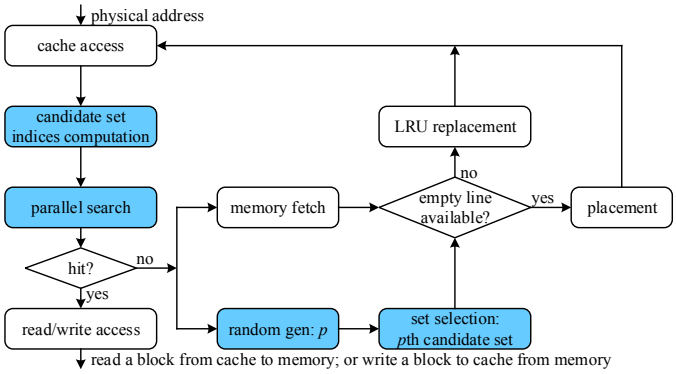

Fig. 3. Cache access handling of PhantomCache.

Placement policy. PhantomCache places a block into a randomly selected cache set among several candidate sets. The indices of the candidate sets are computed using the block's address and random salts. Given r candidate cache sets to use, we introduce r random salts for cache configuration. These random salts are initialized using an on-chip pseudo-random number generation (PRNG) upon machine booting. We can compute the candidate set indices for an address as:

$$C = \{c_i \mid c_i = F(address,\ salt_i) \text{ for } \forall i \in [0,\ r-1]\}. \quad (2)$$

We require that the memory-to-cache mapping function F should generate candidate sets randomly and independently among different addresses. We, however, do not simply duplicate the block into all of the candidate cache sets. That would lead to an impractical cache fatigue. We select only one candidate set at random for placement as the following.

$$SelectedSetIndex\ =\ c_i\ for\ i = PRNG(r), \quad (3)$$

where $PRNG(r)$ generates a random number ranging from 0 to $r-1$.

Search policy. PhantomCache needs to search a block in all its candidate sets. Upon a cache access, we first compute the indices of all candidate sets by Equation 2. We then compare the address's tag field against those cached in each set. A matching indicates a cache hit. Otherwise, a cache miss occurs; the CPU needs to fetch the block from memory and place it in cache. Since a small number of candidate sets are sufficient for security guarantee (Section III-A and Section V), it is feasible to implement parallel search in hardware. Although fully parallelism is difficult to realize because that needs a multi-port cache, we can use a multi-banked cache to improve parallelism.

Replacement policy. PhantomCache imposes no modification on the replacement policy. When placing a block into a cache set, if there is no available cache line, one cached block needs to be replaced. We simply follow the replacement policy in use such as the commonly used LRU policy.

IV. Design

In this section, we detail the PhantomCache design. PhantomCache logics only reside in the LLC management module, serving as a transparent layer between the L2 cache and LLC. The key challenge is how to optimize the extra access latency while implementing randomized localization. We explore a

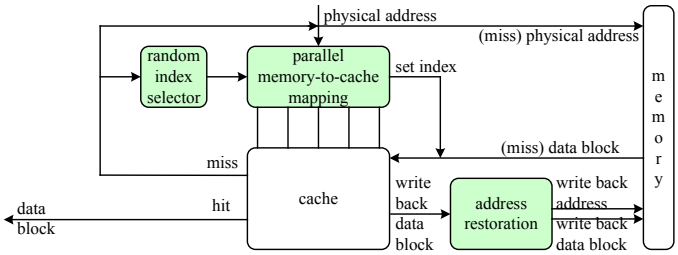

Fig. 4. PhantomCache architecture.

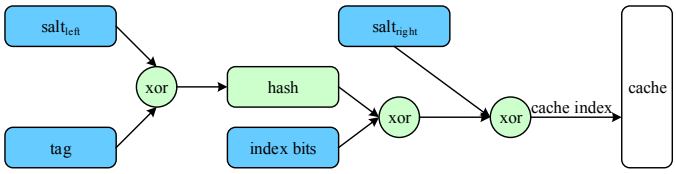

Fig. 5. Memory-to-cache mapping of PhantomCache.

series of design strategies toward only a single clock cycle overhead per access and a $\lceil \log_2 r \rceil$-bit storage overhead per cache line, where r is the number of candidate sets for randomly mapping an address. PhantomCache can be efficiently integrated into the multi-banked LLC architecture.

A. Architecture: Localized Randomization

As with existing randomized mapping solutions, we implement PhantomCache logics mainly through a random memory-to-cache mapping function. As shown in Figure 4, it functions transparently to the cache and memory. The cache still passively accepts an access request and returns the requested data block upon a read hit or continues with caching in the data block upon a write hit. Upon a cache miss, however, the access request is directed to memory[1]. The corresponding data block is then fetched from memory and written to a cache set. The replacement policy (i.e., LRU) decides which cache line to use in the cache set. The placement policy decides which cache set to use for caching a data block with a specific physical address. This is the part that localized randomization shines forth. As discussed in Section III-B, our newly proposed localized randomization technique bounds randomized mapping within only several candidate cache sets across the entire cache. Since any candidate set is likely to be selected, searching a data block needs to walk through all its candidate sets. Moreover, since the memory-to-cache mapping function is modified, we accordingly modify the address restoration process as well. When a dirty data block is written back from cache to memory, the cached metadata should be sufficient for calculating the original memory address.

B. Memory-to-Cache Mapping

Randomness is the ultimate design goal of the memory-to-cache mapping function. Toward randomized mapping, the part of an address used for calculating the cache set index should guarantee uniqueness. In current memory hierarchy, addresses within the same data block always map to the same cache line. We therefore need only a block-wise address uniqueness using the tag and index bits (Figure 2). Since these two fields alone always generate the same cache set index, we can introduce r random salts for r-degree PhantomCache. Specifically, we calculate the indices of r candidate sets using the address's tag and index bits as well as one of the r salts after another. Toward localized randomization of PhantomCache, a random selector is used for selecting one of the r candidate sets for placement.

PhantomCache leverages fast built-in hardware random number generators (HRNGs) on modern CPUs to generate random salts. For example, an Intel CPU [2] can use an entropy source to generate a random stream of bits at a high rate of 3 Gbps. A pair of 256-bit sequences from the entropy source is used as seeds to generate up to 1,022 128-bit random numbers, which are stored in a random number pool in hardware. As with existing hardware-level randomization schemes [35], [45], PhantomCache can directly request r random numbers from the random number pool. This avoids the delay of random number generation upon requests. Furthermore, we need another step of randomness to select one of r candidate sets for block placement. The scale of r is decisive for security. Consider an extreme case when $r = 1$. In this case, PhantomCache degenerates to deterministic mapping that is vulnerable to conflict-based cache timing attacks. We thus require a sufficiently large r to secure PhantomCache. However, a larger r imposes a higher performance overhead due to searching across all the r candidate sets for every data access. The analytical results in Section V and experimental results in Section VII show that PhantomCache can secure a 16 MB 16-way LLC with a small $r = 8$ and only 0.50% performance degradation. To select one from 8 candidate sets for data placement upon each LLC miss, PhantomCache needs only 3 random bits. This should not incur observable latency to the memory-to-cache mapping process.

While preserving mapping randomness, we need to minimize the so-caused cache overhead. The address portion in a cache line should support both data search and address restoration. Traditionally, the tag of a memory address is cached. Since we use an address's tag and index bits and a salt to calculate the cache set index, a straightforward solution needs to at least store the tag and index bits in a cache line. This necessitates a wider cache. We optimize cache overhead such that index bits are not cached. The optimization leverages the fact that addresses with the same index bits (or tag) must have different tags (or index bits). The mapping randomness can still hold if we feed the tag and index bits separately into the mapping function. Besides, to guarantee mapping invisibility to attackers, a salt is divided into $salt_{left}$ and $salt_{right}$, which are used respectively at the beginning and ending of the mapping function. Because a physical address is split into the tag and the index bits, we respectively XOR $salt_{left}$ and $salt_{right}$ with them after they are input to the mapping function to minimize the attacker's control over the mapping result. This idea comes from the key whitening technique that is commonly used in block cipher algorithms such as AES [10].

As shown in Figure 5, we first compute over the tag and $salt_{left}$. The result then goes through another computation

[1]Note that a unit called memory controller coordinates the transmission of control messages and data blocks between the cache and memory. Since PhantomCache does not modify the memory access principle, we omit the memory controller in Figure 4 for simplicity.

Algorithm 2 LFSR based Toeplitz Hash [20]

Input: $message$
Output: $result$
1: $result := 0$;
2: $state :=$ LFSR's initial state;
3: **for** each bit b of $message$ from LSB to MSB **do**
4: // LSB: Least Significant Bit; MSB: Most Significant Bit;
5: **if** $b == 1$ **then**
6: $result := result \oplus state$;
7: **end if**
8: $state :=$ LFSR's next state;
9: **end for**

together with index bits and $salt_{right}$. Note that the second round of computation essentially involves XORing the index bits with $salt_{right}$. The new result is taken as the cache set index. Then we only need to cache the tag and the random number for specifying the adopted salt. This way, cache overhead is minimized to only a $\lceil \log_2 r \rceil$-bit random number per cache line. When we use $r = 8$ to guarantee a strong security for an 16 MB 16-way LLC (Section V), PhantomCache introduces only 0.50% storage overhead per cache line. Furthermore, the length of salts decides security. The longer the salts are, the more robust they are against brute-force attacks. Current PhantomCache makes salt length exactly equal to the length of the tag together with index bits. It incurs only insignificant modification to increase salt length. For example, we can add a third part to the salt and concatenate it with the input of the hash function in Figure 5. Our security analysis in Section V shows that that the current two-part salt is already sufficient to withstand brute-force attacks.

Any random hash function suffices to guarantee randomness of the mapping function. Since PhantomCache requires r candidate sets, we need implement r such mapping functions to support parallel mapping. We adapt the LFSR based Toeplitz hash [12], [20] toward a single–clock-cycle hash function (Section IV-C) with affordable hardware complexity (Section VII-I).

C. Single–Clock-Cycle Hash

The hash function for mapping should be hardware-efficient and guarantee strong randomness. We select the LFSR based Toeplitz hash that satisfies both requirements. An LFSR is a shift register that generates a new state using a linear function and the current state [20]. The LFSR based Toeplitz hash iteratively generates the hash result of an input message (Algorithm 2). Starting from the LSB, each iteration XORs the current state to the result if the message bit is one (lines 5-7). Then LFSR derives the next state for use in the next iteration (line 8). Realization of the LFSR based Toeplitz hash [12], [20], however, uses sequential logic. The message needs to be processed bit by bit, incurring a high latency when it is long.

We adapt the LFSR based Toeplitz hash toward a single–clock-cycle hash function using combinational logic. The state values are pre-computed and stored in registers at boot time. These state values can be directly input to the hash circuit without the delay of re-generation upon each hash computation. As shown in Figure 6, the combinational logic circuit requires

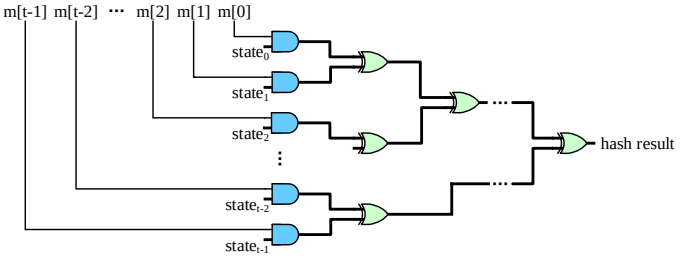

Fig. 6. Single–clock-cycle hash function by implementing the LFSR based Toeplitz hash [20] using combinational logic.

only AND gates and XOR gates. The hash input message is the XOR result of a t-bit tag and a salt. For each AND gate, a message bit with value one makes the corresponding LFSR state go through the XOR gates. Since the message is as long as the tag field, which should be shorter than the up to 64-bit memory address in modern architecture. The number of XOR gates in the critical path is at most $\log_2 64 = 6$. The number of gate delays supported in a clock cycle is determined by several factors such as circuit wiring, clock frequency, lithography, and energy restriction. Typically, modern processors can process 15∼20 gate operations in one clock cycle [34]. Our hash function in Figure 6 with a critical path of 7 gates (i.e., 1 AND gate and 6 XOR gates) thus brings only a single clock cycle latency in most cases. For some processors with an extremely high clock frequency, only 4∼5 gate operations may fit in a single clock cycle. In this case, our hash function may bring a latency of two clock cycles, on par with state-of-the-art cache protection schemes [35], [45]. Even if the mapping latency is two clock cycles, the performance degradation by PhantomCache is still only 1.34% (Section VII).

Randomness. Given an m-bit message M with an n-bit hash output, the randomness of the LFSR based Toeplitz hash is quantified by the following probability-inequality [20]:

$$\forall M \neq 0, c, \quad Pr(h(M) = c) \leq \frac{m}{2^{n-1}}, \tag{4}$$

where c denotes a certain hash output. In our case, consider an LLC with 2^{14} sets and 64-byte cache lines. Given 64-bit physical addresses, we have 14-bit index bits and tags of 64-14-$\log_2 64$=44 bits. PhantomCache uses tags as the hash input and index bits as the hash output. By Formula 4, the probability of a randomly picked tag being hashed to a given set is below $\frac{44}{2^{14-1}} = 0.5\%$, which shows no significant mapping bias.

Security. Admittedly, using a hardware-efficient hash function and simple XOR operations may not be cryptographically secure. However, the application scenario of PhantomCache is different from a typical cryptographic scenario. The attacker is assumed to know only the victim physical address and can only observe cache conflicts. Because of the adoption of random salts, the attacker can neither control the input of the hash function nor know the output of the hash function. This increases the difficulty to create hash collisions deliberately. Furthermore, we explore the LFSR based Toeplitz hash as a low-overhead choice. If this hash function is found to be insecure, we can replace it with other more secure ones with a longer latency than one clock cycle (e.g., the ones used in [34], [45]) as long as they can make the memory-to-cache mapping invisible. This does not affect the key idea of PhantomCache, that is, localized localization. In other words, the essential goal of our mapping

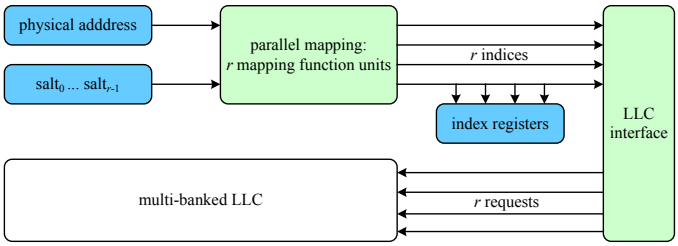

Fig. 7. Parallel search of PhantomCache.

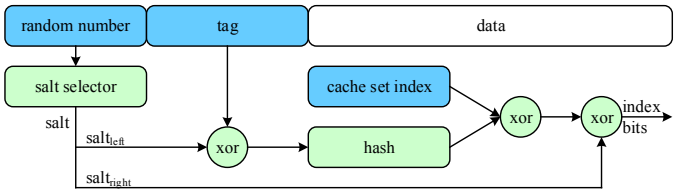

Fig. 8. Address restoration of PhantomCache.

function is to hide the explicit correlation between the physical address and the set index. Such techniques have been well studied in previous solutions [34], [35], [45]. We consider them independent of our localized-randomization technique. This is also why we analyze salt robustness against only brute-force attacks as with existing solutions [35], [45] (Section V).

D. Cache Access

Upon a cache access, PhantomCache enforces parallel search over all r candidate sets of the requested address (Figure 7). We need to check all candidate sets because localized randomization may have mapped the requested address to any of them. With r mapping function units (Figure 7), we first compute the indices of r candidate sets each using one of the r salts and the requested address. Tag fields of all cache lines in the r sets are selected to compare with the requested address's tag in parallel[2]. According to the mapping design (Figure 5), a matching of both the tag field and random number in a cache line yields a cache hit. Otherwise, a cache miss occurs and we need to fetch the requested block from memory to one of the r candidate sets at random. Since we have computed all the indices of candidate sets and stored them in index registers during search, we can directly generate a random number to select one therein. This avoids index re-computation and so caused overhead. Then we follow LRU to place the fetched block into the selected cache set. Meanwhile, the random number used for set selection should also be cached for the sake of address restoration (Section IV-F).

The total extra access latency brought by the mapping function is only one clock cycle. The critical path of the mapping function consists of 10 gates—7 gates of the hash function (Figure 6) and 3 other XOR gates (Figure 5). Given that modern processors can process 15˜20 gate operations per clock cycle [34], the mapping function brings an extra access latency of only one clock cycle.

E. Parallel Search

In order to realize parallel search, we need a multi-banked cache. However, because in PhantomCache any combination of sets may become the candidate sets of an address, accessing all of them in parallel yields a multi-banked cache with exactly one set per bank. Given the large number of sets on an LLC, a set-grained multi-banked LLC is power hungry and will raise manufacture challenges. Therefore, we further propose a parallel search strategy that leverages the existing multi-banked LLC architecture with only a few banks (e.g., 8). Each bank contains an equal number of sets. This design may partially sacrifices parallelism because more than one of the r candidate sets may map to the same bank. In other words, since we still randomly map an address to any r sets across the entire LLC, it is hard to always guarantee that r sets map to r banks, especially when the LLC has fewer than r banks. A design challenge is then how to synchronize all the r access requests corresponding to an address. Among the r accesses, a hit succeeds the original access request while all misses lead to a cache miss. Once they go to different banks, different queue lengths on each bank may make the r accesses complete asynchronously. This confuses the CPU as a miss comes after a hit of the same address request. Then the CPU needs to handle the cache miss by fetching the corresponding data block from memory even though it is already cached.

We modify the LLC queue management policy to support PhantomCache. For each address request, we inflate it to r requests. Each inflated request targets a possible set the address maps to. According to which bank a set belongs to, these inflated requests are scheduled to different bank queues. Once an inflated request is served, we use its result to update the status of the original request in the LLC queue. Specifically, we maintain a counter and a hit indicator for each original request in the LLC queue. If the inflated request is a hit, the counter is incremented, the hit indicator is enabled, and the cached data block is immediately sent to the L2 cache. Otherwise, we only increment the counter. For an original request in the LLC queue, if after all its r inflated requests get served and the hit indicator is enabled, it can be removed from the queue. If its counter becomes r and the hit indicator is not set, the original request encounters a cache miss and invoke memory access. Thanks to the pipelined optimization of modern caches [32], [37] where consecutive requests can be handled without multiplying the access latency[3], PhantomCache introduces a limited performance overhead. For example, given $r = 8$ that guarantees a strong security on an 8-bank 16 MB 16-way LLC (Section V), PhantomCache brings only 0.50% performance slowdown on average (Section VII-F).

F. Address Restoration

We need to restore memory addresses of blocks when writing them from cache back to memory. Address restoration takes place when dirty blocks are evicted from cache upon replacement or process termination. As aforementioned, address restoration requires both the tag and index bits of a memory address. The tag field is already stored in the cache line. We further restore index bits as in Figure 8. As aforementioned in

[2]Note that in traditional set-associative caches, cache lines in a cache set are also checked in parallel. With a multi-banked cache, we are able to check multiple cache set in parallel

[3]For example, LLC needs 20 clock cycles to handle an individual request, but it takes only 21 cycles to handle two consecutive requests because they are handled in pipeline.

Figure 5, index bits are used for determining the cache set index through an XOR with a hash output and $salt_{right}$. The hash input is the XOR result of the tag and $salt_{left}$. Since the random number for specifying the salt is also cached, we can find the salt and divide it into $salt_{left}$ and $salt_{right}$, XOR $salt_{left}$ with the tag, and hash the XOR result. Finally, XORing the hash output with the set index and $salt_{right}$ restores the index bits. With the tag and index bits, we can restore the memory address and write back the evicted block to the corresponding memory location.

V. Security

In this section, we analyze the security of PhantomCache. Localized randomization significantly obfuscates cache conflicts and costs the attacker an unreasonable long time to find a minimal eviction set or crack salts. For example, under PhantomCache with 8 candidate sets, the eviction set minimization process takes the attacker more than 500 years even if it uses the state-of-the-art $\mathcal{O}(|E|)$ algorithm. A number of 8 candidate sets can already enforce more than 136 years to crack a salt. We also randomly initialize the salts upon machine booting. This renders salt cracking more impractical.

A. Threat Model

Following related work [34], [35], [45], we are concerned with an attacker that knows the victim address. The attacker can launch a conflict-based attack as long as it obtains a minimal eviction set for the victim address. Therefore, we consider an attack successful if the attacker finds a minimal eviction set.

We make the following assumptions in favor of the attacker. These assumptions are commonly accepted in the literature [34], [35], [45].

- The attacker knows the exact physical address accessed by the victim process.
- The attacker possesses abundant initial addresses that contain sufficient addresses for forming a minimal eviction set. It may choose to run the classic $\mathcal{O}(|E|^2)$ algorithm or the state-of-the-art $\mathcal{O}(|E|)$ algorithm (Section II-B).
- The attacker can make memory accesses and measure access latency.
- The attacker is interfered with no noise during the attack. That is, it is the only entity that makes memory accesses until a minimal eviction set is found.
- Regarding PhantomCache specifics, the attacker can be aware of the design of the mapping function, such as the hash function. However, it cannot know the exact salts that are used for computing the exact cache set indices for an address. The attacker may try to crack the salts.

B. Security Goal

As with existing solutions [34], [35], [45], the security goal of PhantomCache is to prevent the attacker from finding a minimal eviction set within a reasonable time. Specifically, a defense is considered as providing strong security if it can hinder eviction set minimization for more than 100 years [34]. PhantomCache achieves the security goal via three subgoals.

Scarcity of eviction addresses. Sufficient eviction addresses are necessary for the attacker to form a minimal eviction set. PhantomCache significantly raises the bar for feasible eviction addresses. It no longer maps an address to a determined cache set. Instead, each address may be randomly mapped to one of its candidate sets. It becomes impractical for the attacker to simply find eviction addresses that share all the same candidate sets (Section V-C). The attacker then has to resort to eviction addresses that share only part of their candidate sets. However, which candidate set an address really maps to is random upon it is cached. This further boosts the difficulty of eviction set minimization.

Hardness of eviction set minimization. Given that there always have sufficient addresses mapping to the same cache set, one can hardly prevent the existence of minimal eviction sets. A countermeasure thus aims to obstruct the process of eviction set minimization. As with hardware-level randomization schemes [35], [45], PhantomCache essentially enforces such obstruction through randomizing the memory-to-cache mapping. The difference is that PhantomCache leverages our proposed localized randomization technique toward a more efficient protection (Section V-D).

Hardness of salt cracking. Salts play a critical role for defending against eviction set minimization. If the attacker knows the salts, it can greatly ease the eviction set minimization process. This is because that it can use the salts to easily compute the candidate set for any physical address. To test the validity of a salt, the attacker could use it to calculate a set of addresses with a common candidate set. Then it repeats accessing these addresses. If the common candidate set can be primed, the attacker can make sure that the salt is currently used by the system. Such a cracking process induces heavy memory accesses, which associate with a long access time. Our analysis shows that PhantomCache is robust against salt cracking with a sufficiently long endurance time (Section V-E).

We next detail how PhantomCache satisfies these security subgoals—scarcity of eviction addresses, hardness of eviction set minimization, and hardness of salt cracking—in Section V-C, Section V-D, and Section V-E, respectively.

C. Scarcity of Fully-Congruent Addresses

The most intuitive way to launch a conflict-based cache timing attack in PhantomCache is to utilize addresses whose candidate sets are exactly the same. Such addresses are called fully-congruent addresses [45]. Given $m \times r$ fully congruent addresses, the attacker can use them to form an eviction set. To prime all the r candidate sets of the victim address, the attacker keeps accessing the whole eviction set and measures the latency. Note that if all the r candidate sets are primed, the attacker will not observe any cache conflicts in his access because all the $m \times r$ memory blocks are already in the cache. After an iteration of access without any cache conflicts, the attacker will stop the priming phase and trigger a victim access, after which the attacker will start the probe phase. However, we will next show that there may not be sufficient fully-congruent addresses in the system.

Consider the Intel HasWell core i7-3720QM processor, where the LLC includes 8,192 sets and each set holds 12 lines of 64-byte data [33]. Assume that in an r-degree PhantomCache where a physical address has r candidate sets, the attacker needs to find $12 \times r$ fully-congruent addresses to form an eviction set. Because of randomized mapping, the r possible cache sets are independent of each other. The possibility that two randomly selected addresses are fully-congruent is $\frac{1}{8192^r}$. Thus, finding $12 \times r$ fully-congruent addresses needs $12 \times r \times 8192^r \times 64 = \frac{3 \times r \times 8192^r}{4}$ KB on average. This result shows that the space cost grows exponentially with r, where the base is the number of sets in the cache. Note that even if r is only set to 2, the needed space still grows to an enormous scale: 96 GB. In any system, the maximum memory space of a process is limited, which indicates that when r is large enough, the attacker can never find enough fully-congruent addresses because they do not exist in the available memory space. Assume that the maximum memory space is M, cache associativity is m, the degree of PhantomCache is r, the capacity of data in a cache line is c, and the number of sets in the cache is s. Then, the minimum r to guarantee the scarcity of fully-congruent addresses is the minimum r satisfying the following constraint:

$$m \times r \times s^r \times c \geq M. \tag{5}$$

Because of the scarcity of fully-congruent addresses, the attacker has to resort to partially-congruent addresses, whose candidate sets overlap with the victim address's. Such addresses are abundant in the memory space. For example, if the cache has c cache sets in total, the attacker can always find 2 addresses that have a candidate set in common out of $\frac{c}{r}$ addresses. Partially-congruent addresses can be used to prime one or more candidate set of the victim address. Thus, with multiple groups of partially-congruent addresses, the attacker may be able to prime all the candidate sets of the victim address. We refer to such a group of partially-congruent addresses as an eviction set for the common candidate cache set. However, we find that even if we have loosed the requirement for eviction sets, it is still extremely difficult to find a minimal eviction set.

D. Hardness of Eviction Set Minimization

To obtain a minimal eviction set for the target address (i.e., victim address), the attacker must go through a minimization process. This is natural because according to the LRU replacement policy, cache conflicts only occur when some cache sets are filled up. We have assumed that the attacker is the only entity making memory accesses during the attack. If the attacker accesses a small group of random addresses, it can hardly fill up any cache set and observe cache conflicts. Therefore, the attacker needs to begin with a large number of addresses and manage to minimize them into a minimal eviction set.

As demonstrated in Algorithm 1 [26], [30], the minimization process starts with an initial set E of candidate addresses and proceeds by removing unnecessary addresses therein. Consider a cache with c cache sets in total. If the attacker randomly picks an address, the possibility that the address's candidate sets includes s_x, a certain candidate set of the target address x, is $\frac{r}{c}$. To form an initial E including m relevant addresses, the attacker needs $\frac{c}{r} \times m$ addresses on average [4].

TABLE II. TIME FOR EVICTION SET MINIMIZATION UNDER PHANTOMCACHE USING r CANDIDATE CACHE SETS.

r	$\mathcal{O}(\lvert E\rvert^2)$ algorithm	$\mathcal{O}(\lvert E\rvert)$ algorithm
2	13 days	0.7 seconds
4	584 years	6.5 days
6	170,682 years	7.8 years
8	9,583,986 years	584 years

Before removing one or more addresses from the initial set E, the attacker needs to test if the remaining addresses are still sufficient to fill up s_x. In other words, there are still at least m addresses remained whose candidate sets include s_x. In traditional caches, the attacker only needs to test once. Specifically, it first accesses x together with all the remaining addresses. Then it reloads x and measures latency. If x is observed to be evicted, then the remaining addresses are sufficient to fill up s_x.

PhantomCache drastically increases the difficulty of the preceding tests of address removal. Because every address may map to one of r candidate sets, the probability that all the m partially-congruent addresses map to the same target cache set s_x is only $\frac{1}{r^m}$. If one or more of them do not map to the target cache set, x cannot be evicted. This means that the attacker needs to repeat the test for at least r^m times to avoid removing addresses that should be retained. To summarize, PhantomCache brings an $\mathcal{O}(r^m)$ blowup to the complexity of existing minimization algorithms (both $\mathcal{O}(|E|^2)$ [33] and $\mathcal{O}(|E|)$ [35], [40]).

We now calculate the number of memory accesses by both types of algorithms to minimize an eviction set in PhantomCache. We start with the well-investigated and widely exploited minimization algorithm with an $\mathcal{O}(|E|^2)$ complexity [26], [30], [33], [40]. Given $\frac{c}{r} \times m$ addresses in the initial eviction set E, the attacker removes one address after accessing the remaining addresses for r^m times per iteration. The iterative address-removal process ceases upon the completion of the iteration with only m addresses left. We accordingly calculate the total number of memory accesses in the $\frac{c}{r} \times m - m + 1$ iterations as follows.

$$\sum_{i=m}^{\frac{c}{r} \times m} r^m \times i = \frac{1}{2} \times (\frac{c \times m}{r} + m) \times (\frac{c \times m}{r} - m + 1) \times r^m. \tag{6}$$

For the $\mathcal{O}(|E|)$ algorithm, although the complexity is linear with respect to $|E|$, the constant coefficient varies. So far, the best result is achieved in [35], that is, $37 \times |E|$. In our calculation, we consider an extreme case with the coefficient as only one in favor of the attacker. Then the total number of memory accesses approximates as the follows.

$$\frac{c \times m}{r} \times r^m. \tag{7}$$

Table II provides the time cost for the attacker to find a minimal eviction set on an Intel Xeon E5-4620 CPU with a 16-way 16 MB LLC. In this case we have $m = 16$ and

[4]The attacker can also start with more addresses, but that will only take it more time to minimize the eviction set because finally only m addresses are needed.

$c = 16{,}384$. We lower bound the memory access time by L1-cache access time, that is, approximately 2 ns. When we set r as 4, the $\mathcal{O}(|E|^2)$ algorithm costs the attacker more than 500 years. For the $\mathcal{O}(|E|)$ algorithm, $r = 8$ suffices to take the attacker 500+ years for finding a minimal eviction set. In comparison with global randomization across all the $c = 16{,}384$ sets, PhantomCache uses only a limited randomization space to achieve a strong security guarantee.

However, we have only proved the security of PhantomCache against existing minimization algorithms with a remove-and-test style [40]. It is possible that new attacks with less complexity than state-of-the-art $\mathcal{O}(|E|)$ or new minimization style other than remove-and-test will emerge. Such attacks may break the defense of PhantomCache as well as related schemes [34], [35], [45]. To the best of our knowledge, a general lower bound for remove-and-test algorithms or even all minimization algorithms still remains an open question. This is also why we cannot provide a further analysis or proof besides the current findings over the state-of-the-art $\mathcal{O}(|E|)$ algorithm.

E. Hardness of Salt Cracking

If the attacker usurps a privilege of directly using physical addresses to access memory, the secrecy of salts becomes critical. Otherwise, the attacker can bypass the hard minimization process. This is because that it can easily find a minimal eviction set by computing the candidate sets of each physical address. Therefore, salts should be robust against cracking. To further increase the attacker's leverage, we assume that the attacker can compute indices of candidate sets fairly fast and omit so caused time cost in the following analysis. Since a salt consists of two parts respectively for XORing with the tag field and index bits (Section IV-B), the length of a salt is equal to that of both fields. Given that typical modern processors use 64-bit physical addresses and 64-byte data blocks, the length of a salt is $64 - \log_2 64 = 58$ bits. Through a brute-force attack, the attacker needs to test $\frac{2^{57}}{r}$ salts on average to find a correct salt among r ones. A simple way to test a salt is using the salt to calculate a group of addresses with a common candidate set, and then accessing them to check whether the common candidate set can be primed. At least m such addresses are required to prime an m-way cache set. Therefore, testing a salt requires at least m memory accesses[5]. Because using different salts likely computes different group of addresses, most memory accesses during salt testing lead to main-memory accesses. Considering that the attacker can parallelize memory accesses in a multi-bank memory with b channels, the total number of memory accesses to crack one salt is $m \times \frac{2^{57}}{r} \times \frac{1}{b}$.

Consider a system with a 16-way LLC, a 4-channel memory, and a typical main memory access latency of 60 ns [21]. When we configure a highly secure $r = 8$ (Section V-D), the attacker need take more than 136 years to crack a salt. Given that the r salts are randomly initialized upon machine booting, it is impractical for the attacker to bypass the eviction set minimization process by cracking the salts.

We may also increase the salt cracking hardness by introducing the salt randomness upon the addresses. Currently, each salt is divided into two parts, $salt_{left}$ and $salt_{right}$ with the same size as that of tag bits and index bits, respectively. Given a specific salt, then it splits into the same two parts to XOR with different addresses. If we use longer salts, we can perform some address-specific computation over a long salt such that different addresses may generate different $salt_{left}$ and $salt_{right}$ from the same salt. This way, we can further randomize the inputs and therefore increase the randomness of address mapping. This surely will increase the hardness of salt cracking by learning the address mapping pattern.

[5]In fact, m memory accesses can only prime the cache set with a $\frac{1}{r^m}$ probability. A reliable test needs much more than m accesses.

F. Global Protection versus Selective Protection

Selective protection has been explored to find a trade-off between security and efficiency. In contrast with global protection that enforces protection on all cache accesses, selective protection enforces protection on only data of security interest and leaves other data simply following traditional cache accesses. For example, PLcache can assign cache partitions to data worthy of protection [42]. Corresponding processes access these data using proprietary locks granted to them. Selective protection is usually considered as a straightforward extension.

However, we find that selective protection is infeasible for PhantomCache due to security breach. Specifically, PhantomCache uses localized randomization that 1) deterministically chooses r candidate sets for an address to map and 2) randomly maps the address to one of the r sets. According to the security analysis in Section V-C, PhantomCache enforces an r^m blowup for the number of memory accesses to the attacker, where m represents set associativity. When selective protection is used, localized randomization applies to only data of security interest. Consider a victim address under protection for example. The attacker is not enforced with localized randomization and thus its accesses still follow deterministic mapping. If this is the case, the attacker can again use existing eviction set minimization algorithms to form an eviction set for a specific cache set fairly fast. Since the victim address has r candidate sets in PhantomCache, the attacker only needs to repeat the testing step in the algorithm for r times. That is, under selective protection, the attack overhead is subject to an r blowup, which is marginal in comparison with the r^m scale by global protection of PhantomCache and therefore much less secure. For example, given a 16-way 16 MB LLC with 16,384 sets and $r = 8$ candidate sets for PhantomCache address mapping (Table II), selective protection can drastically degenerate security in that a minimal eviction set found in at least 9,583,986 years under global protection can be found in only 0.024 seconds. Therefore, we do not suggest to use selective protection for PhantomCache; we consider only the global protection mode in what follows.

VI. Implementation

We implement PhantomCache using ChampSim [1], a trace-based microarchitecture simulator. It models a full-fledged CPU of out-of-order cores with a 3-level on-chip cache hierarchy. This makes ChampSim well accepted in academia for evaluating cache performance. For example, it is the designated simulator for the Cache Replacement Championship at ISCA '17 and the Data Prefetching Championship at ISCA '19. PhantomCache implementation enforces our localized randomization technique over the conventional cache access management in ChampSim. As discussed in Sections III and IV, key components include

TABLE III. EXPERIMENT SETUP.

Module	Configuration
Processor	1∼8 cores, 3.2 GHz, out-of-order 256-entry ROB
Private L1 I/D cache	64-set, 8-way, 32 KB
Private L2 cache	512-set, 8-way, 256 KB
Shared LLC	2∼16 MB, 8-bank, 16-way, 20 cycles
Memory	800 MHz (DDR 1.6GHz), 1∼2 channels, 8-Banks each, 2 KB row buffers, tCAS-tRCD-tRP: 11-11-11

the set index calculation unit, the cache search unit, and the cache replacement unit. Since ChampSim features a non-inclusive cache by default, it should be first modified to support inclusion. To this end, we add a `back_invalidate` function and modify `handle_fill` and `handle_writeback` procedures in the cache module. When a data block is replaced, `back_invalidate` is invoked to evict the data block in higher-level caches as well if it exists therein.

As with hardware-only randomized mapping solutions (e.g., NewCache [25] and CEASER [34]), our modification over the inclusion-enabled ChampSim touches only the LLC module. It remains as a transparent layer between the L2 cache[6] and memory controller. To handle a memory access request, ChampSim searches the addressed data block through higher-level caches to lower-level caches until a cache hit returns. If all of the L1 cache, L2 cache, and LLC return cache misses, the memory access request is directed to memory. The addressed data block is then fetched from memory to each level of the cache hierarchy. To keep the interfaces for L2-LLC and LLC-memory communication intact, PhantomCache modifies the read and write procedures inside the LLC module. The modification for both procedures lies mainly in the handling of cache search and cache replacement. For the search process, we add the single–clock-cycle hash function (Section IV-C) and call it in the `get_set` function. This function returns the indices of a set of candidate sets rather than a single set index determined by the index bits of the requested address. To perform cache search and invalidation over all these candidate sets, we also modify the `check_hit` and `invalidate_entry` functions. A cache miss triggers the replacement process by randomly selecting one of the candidate sets. The data block fetched from memory is then placed into the selected cache following LRU.

VII. EVALUATION

Workloads. We evaluate PhantomCache performance by running workloads from the SPEC CPU 2017 benchmark package [3]. Specifically, we use all the 20 benchmarks from the SPECspeed 2017 Integer and SPECspeed 2017 Floating Point suites. For each benchmark, a representative slice is selected for fast simulation with performance estimates comparative to full-execution simulation [17]. Such a representative slice can be obtained using SimPoint [17] and Pin [27]. The workloads corresponding to the representative slices of all 20 adopted benchmarks are readily available in ChampSim. When running a workload only consisting of a single benchmark on a multi-core CPU, every core is running the same benchmark. We also evaluate the performance using mixed workloads that are generated by randomly selecting n out of the 20 benchmarks on an n-core CPU. Each selected benchmark is then pinned to a different core [34], [46]. We run at least 2 billion instructions per workload. The first 1 billion instructions are used for warming up the cache while the other 1 billion or more instructions are used for collecting performance statistics.

[6]This is because ChampSim features a 3-level cache. The L2 cache is exactly from which the LLC receives an access request.

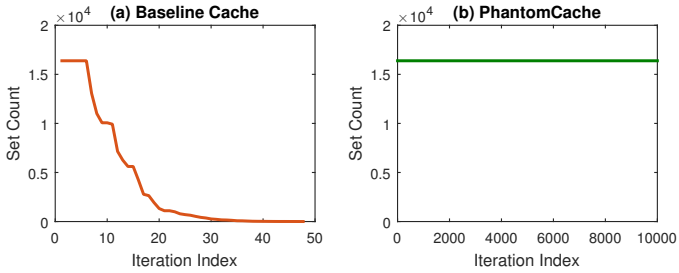

Fig. 9. Comparison of resistance to eviction set minimization.

Metrics. We evaluate PhantomCache using three performance metrics—instructions per cycle (IPC), misses per 1,000 instructions (MPKI) of LLC, and miss rate of LLC. To evaluate how PhantomCache impacts cache performance, we normalize all these metrics using the ratio of PhantomCache's metrics to that of the baseline cache without modification. A higher normalized IPC indicates a better performance, exceeding 100% if PhantomCache outperforms baseline. Moreover, a lower normalized MPKI or normalized miss rate demonstrates a better performance. To measure aggregate performance, we further report the geometric mean of normalized IPC and the average of normalized MPKI and normalized miss rate.

Results. Based on the configuration in Table III, the results show that, to secure an 8-bank 16 MB 16-way LLC against the powerful $\mathcal{O}(|E|)$ attack, PhantomCache introduces a 1.20% slowdown on average among all 20 SPEC CPU 2017 benchmarks (Figure 10). When taking into account the same set of 17 benchmarks as the state-of-the-art ScatterCache [45], PhantomCache introduces an only 1.06% slowdown on average, being 2x more efficient than ScatterCache. In terms of mixed workloads, PhantomCache brings a much smaller slowdown of 0.50% on average.

A. Resistance to Eviction Set Minimization

We first evaluate the effectiveness of PhantomCache against eviction set minimization. Following recent related work [34], [35], [45], we run the sate-of-the-art $\mathcal{O}(|E|)$ eviction set minimization algorithm on a traditional cache and PhantomCache, respectively. The security metric we use is the number of cache sets accessed during each iteration of the algorithm. (The term "accessed" means that among all the memory accesses in this iteration, at least one address maps to that cache set.). If the eviction set minimization process succeeds, we expect to observe a noticeable decrease in the number of accessed cache sets as the iteration proceeds.

We configure a 16 MB 16-way LLC with 16,384 cache sets. The initial eviction set consists of $16{,}384 \times 16$ randomly picked addresses. The size of the initial set guarantees a high possibility that there are sufficient addresses to form an eviction set for the target cache line.

Fig. 10. PhantomCache performance with metrics normalized over baseline.

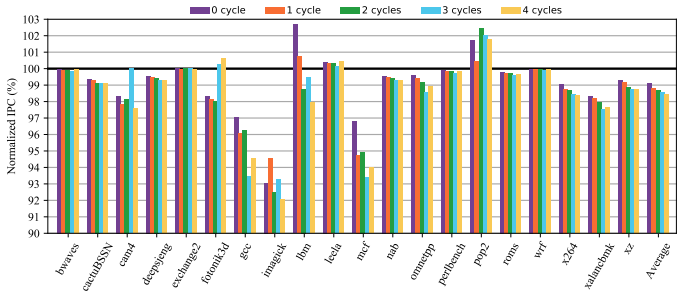

Fig. 11. Impact of calculation latency on PhantomCache performance.

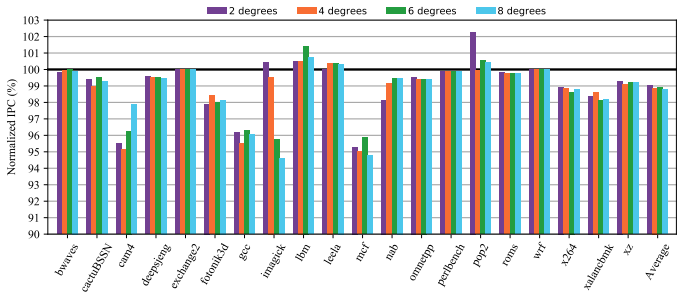

Fig. 12. Impact of randomization degree on PhantomCache performance.

As shown in Figure 9, the $\mathcal{O}(|E|)$ algorithm completes the eviction set minimization process in a traditional cache after only 48 iterations. However, in PhantomCache, the number of accessed cache sets of each iteration remains the same even after 10,000 rounds. The result shows that the most powerful eviction set minimization algorithm so far does not work in PhantomCache. An attacker gains no progress of minimization in PhantomCache as in a traditional cache.

B. Processor Capacity

To evaluate how PhantomCache affects cache performance, we start with experiments under various processor capacity settings. Processor capacity differs mainly in the number of cores, the size of LLC, and the number of channels connecting to memory. We consider both single-core and multi-core processors. Each core is usually assigned with a 2 MB cache [18]. For randomization degree, we use $r = 8$ by default as it guarantees a strong security level. Figure 10 reports the normalized performance metrics on three different processors. PhantomCache imposes only an average normalized-IPC degradation of 0.05%, 1.02%, 1.20% on the 1-core, 4-core, and 8-core processors, respectively. The corresponding increase of normalized MPKI is 0.08%, 0.14%, and 0.41%. Normalized miss rate shows a much smaller growth up to 1.40%. The performance of PhantomCache, however, does not necessarily lead to performance degradation for all workloads. For example, the `lbm` workload and the `leela` workload in Figure 10(c) improves the performance by 0.73% and 0.34%, respectively. To evaluate how salt choice affects performance, we run multiple trials of small workloads (each with 0.1-billion instructions) with different salts per run. The results show little effect on performance by different salts. This mainly attributes to strong randomness of the mapping function (Section IV-C). Moreover, a single trial of a workload with 1-billion or more instructions may suffice to quantify the average over multi-trial of the same workload [34], [46].

We understand the slight impact of PhantomCache on processor performance as follows. First, it takes extra access latency because of newly introduced components such as the hash function. Since all the introduced operations complete in only one clock cycle (Section IV-D), the extra latency cannot fluctuate overall performance in comparison with a 20 clock-cycle LLC hit and a 100+ clock-cycle memory access. Second, since PhantomCache changes the mapping pattern of LLC, conflict misses are affected. Addresses that map to the same set in the baseline cache can map to different sets in PhantomCache. While lessening cache conflicts in this set, the remapped addresses may increase cache conflicts in other sets. This feature has an unpredictable influence on MPKI and miss rate. Some workloads may happen to enjoy fewer cache misses while others may suffer more. As shown in Figure 10, the overall impact on performance is as minimal as 1.20%. Note that the state-of-the-art ScatterCache does not test the three benchmarks of `gcc`, `wrf`, and `cam4` due to compilation failure [45]. They happen to be the major contribution to performance degradation in our test as shown especially in Figure 10(b) and Figure 10(c). Toward an objective comparison, we recalculate the overall performance degradation of PhantomCache without considering `gcc`, `wrf`, and `cam4`. The result is decreased to 1.06%, which is 2x more efficient than ScatterCache with 2% performance degradation [45].

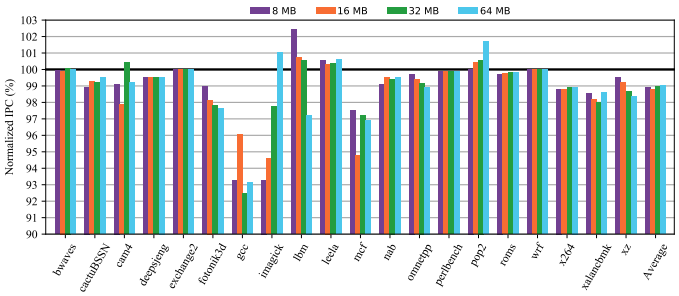

Fig. 13. Impact of LLC capacity on PhantomCache performance.

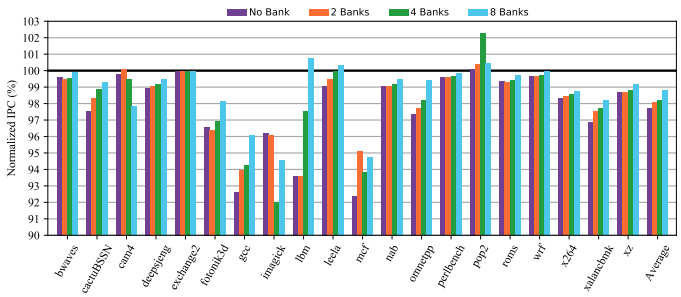

Fig. 14. Impact of bank numer on PhantomCache performance.

C. Calculation Latency

PhantomCache simply uses XOR and a hash function to efficiently calculate the candidate sets of a requested data block. Since the calculation process is necessary for each LLC access, its latency is performance critical. We evaluate the impact of calculation latency using an 8-core CPU, a 16 MB LLC, and a 2-channel DRAM. Figure 11 shows the performance of PhantomCache with the calculation latency ranging from 0 cycles to 4 cycles. A 4-cycle latency decreases normalized IPC by 1.53%. As expected, a lower latency yields a better performance. Based on our current design, PhantomCache promises only one clock cycle latency. This decreases normalized IPC by only 1.20%.

D. Randomization Degree

So far, we report all performance statistics using a number of $r = 8$ candidate sets for an address. We choose it because of its strong security guarantee. As shown in Table II, 8-degree PhantomCache costs an attacker more than 500 years to find a minimal eviction set. A smaller $r = 6$, however, can also impede the eviction set minimization process for as long as 7.8 years. This should be sufficient for securing most computers. One may thus become curious about how PhantcomCache performs using fewer candidate sets. We evaluate the impact of randomization degree using an 8-core CPU, a 16 MB LLC, and a 2-channel DRAM. Figure 12 shows the normalized performance of PhantomCache with a randomization degree of 2, 4, 6, and 8. We observe that the performance is relatively insensitive to randomization degree. This encourages adopters of PhantomCache to strive for an even stronger security than 8 candidate sets provide, as long as they consider the corresponding hardware cost affordable.

E. LLC Capacity

As the original goal of PhantomCache is LLC friendliness, we further evaluate the scalability of PhantomCache as LLC capacity increases. We evaluate the impact of LLC capacity using an 8-core CPU, a 2-channel DRAM, and an LLC with varying capacity. Figure 13 shows the normalized IPC of PhantomCache in an 8 MB, 16 MB, 32 MB, and 64 MB LLC. As expected, a workload performs better upon a larger LLC. This is because of the intrinsic cache property that a larger cache guarantees fewer memory accesses, which is much slower than cache accesses. Furthermore, in comparison with the baseline cache, the average performance degradation by PhantomCache using an 8 MB, 16 MB, 32 MB, and 64 MB LLC is at most 1.20%. This demonstrates that PhantomCache can efficiently secure large LLCs against conflict-based cache timing attacks.

F. Bank Number

We evaluate how PhantomCache affects cache performance when we realize it using multi-banked LLCs with different number of banks. Specifically, we use a 16 MB 16-way LLC and divide it to different numbers of banks. "No Bank" corresponds to when the cache is not divided. In this case, PhantomCache has no parallelism to enjoy and has to sequentialize all cache accesses. Figure 14 reports the performance results. Generally, the performance degradation decreases as the number of banks increases. For the no-bank PhantomCache, the average performance degradation is within 2.27%. For the 8-bank PhantomCache, the average performance degradation is only 1.20%.

G. Sliced LLC

Beyond the multi-banked structure, we further adapt PhantomCache to recent sliced LLCs and evaluate its performance. Modern Intel CPUs use sliced LLCs where the LLC is split into different slices. A physical address maps to a slice through some special hash function [29]. The hash function is designed to guarantee that consecutive memory lines can be distributed across different slices and thus can be read by different cores in parallel to improve efficiency. However, the slice mapping function of existing sliced LLCs is deterministic [29]. It does not apply to PhantomCache that requires randomized mapping. For adapting PhantomCache to an s-slice LLC, we choose to use the $\log s$ most significant bits of the computed set index by PhantomCache as the slice ID. Since the set index is a hash result of the physical address and a random salt, it has intrinsic randomness. Taking its s most significant bits can also guarantee randomness. This further ensures that consecutive addresses be distributed across different slices.

We compare the performance of the preceding adapted PhantomCache with that of the baseline sliced LLC that uses the reverse-engineered hash function [29] of an Intel sliced LLC. We run both approaches on an 8-core CPU with 16 MB 8-slice LLC with 8 candidate sets for PhantomCache and report the results in Figure 15. In terms of normalized IPC, PhantomCache introduces a performance degradation of 1.52% (and 1.38% without including `gcc`, `wrf`, and `cam4`) to the baseline sliced LLC. The performance degradation is slightly larger than previous 1.20% on the multi-banked LLC. It implies that the slice mapping function of PhantomCache does impact on the effect of memory access parallelism across different slices. Since currently we simply use the 3 most

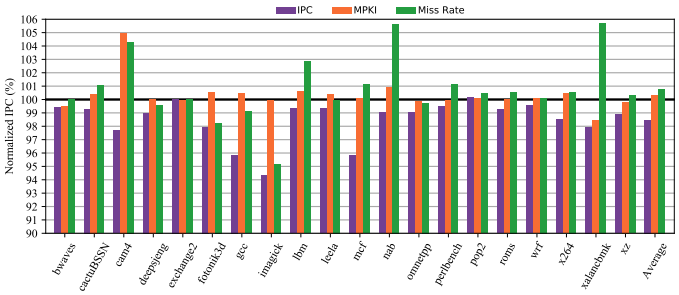

Fig. 15. PhantomCache performance with metrics normalized over the baseline sliced LLC.

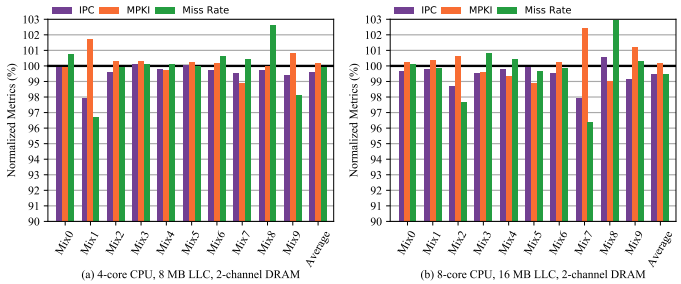

Fig. 16. PhantomCache performance with metrics normalized over baseline using mixed workloads.

significant bits of the computed set index for slice mapping, it is challenging to reserve the property of evenly mapping consecutive addresses to different slices as the baseline sliced LLC. Improving the slice mapping of PhantomCache toward such a property certainly improves PhantomCache performance. However, such an improvement may or may not bring much intricacy to hardware and is left for future work.

H. Mixed Benchmark

As with some related work [34], [35], [46], we also evaluate PhantomCache performance using mixed workloads. On an n-core CPU, a mixed workload is generated by randomly selecting n out of the 20 benchmarks. During experiment execution, each of the n selected benchmarks is pinned to a different core. This resembles the environment of a real system where different applications run on different cores in parallel. We run 10 mixed workloads for both 4-core CPU and 8-core CPU. PhantomCache is configured as default with $r = 8$ and mapping latency of 1 clock cycle on an 8-bank LLC. Figure 16 reports the performance comparison with the baseline 8-bank LLC. We observe that PhantomCache performs better on mixed workloads than on individual benchmarks. For example, it introduces an only 0.50% average performance degradation to an 8-core CPU with an 8-bank 16 MB 16-way LLC (Figure 16(b)).

We analyze the slight performance overhead of PhantomCache as follows.

Limited miss rates on L1 and L2 caches. By design, the higher-level L1 and L2 caches, can satisfy most of the processor's memory accesses. In contrast, the LLC has much less visibility to the memory activity [26]. Therefore, slower LLC handling may not contribute too much to overall slowdown.

Pipelined cache optimization. Modern caches are optimized with pipelined cache-requests handling. That is, n consecutive requests do not necessarily cause n times latency as that of a single request.

Parallelism over multiple banks. Multiple banks mitigate the contention caused by the 8 inflated requests for every original requests. The more the banks, the more requests can be processed in parallel.

Bank assignment unevenness. Most workloads may not fully utilize the multi-banked LLC. We observe that it is uncommon that multiple cores access the LLC at the same time. Some banks that stay idle in the baseline cache will be made full use of in PhantomCache. This further reduces contention and thus increase PhantomCache performance.

I. Hardware Overhead

The hardware complexity of PhantomCache design depends on the system configuration. Consider a system using r-degree PhantomCache, an m-way set associative cache, and physical addresses with t-bit tags and i-bit index bits.

Logic overhead. The logic overhead in LLC mostly stems from the mapping logic. The mapping function contains a hash with $2 \times t \times i$ 2-input AND gates and $(t + 2 \times i)$ 2-input XOR gates (Figure 5). Thus, all r mapping units contain $r \times (t+2 \times t \times i+2 \times i)$ 2-input gates, consisting of $r \times t \times i$ AND gates and $r \times (t \times i+t+2i)$ XOR gates.

Under our default LLC with $t = 44$, $i = 14$, $m = 16$, and $r = 8$, the total logic overhead is 10,432 2-input gates. Specifically, it includes 4,928 AND gates and 5,504 XOR gates. In terms of gate equivalent (GE), an AND gate typically has the same size as 1.5 NAND gates and an XOR gate typically has the same size as 2 NAND gates. The logic overhead of the mapping function is around 18,400 GEs.

The logic overhead of PhantomCache is less than two times of the logic overhead of 128-bit AES encryption, which is around 10,000 GEs [4]. This is considered affordable in modern CPUs.

Storage overhead. PhantomCache needs extra storage space for salts, cache set indices, and LFSR state values. The r salts take up $r \times (t+i)$ bits, the r cache set indices take up $r \times i$ bits, and the t state values take up $t \times i$ bits. Using our default LLC with $t = 44$, $i = 14$, and $r = 8$, the extra storage overhead is only 1,192 bits (149 bytes). Besides, each cache line stores an extra $\lceil \log_2 r \rceil$-bit random number for indexing salts. When $r = 8$, it takes up 3 bits and introduces around 0.50% storage overhead per cache line.

J. Energy Overhead

Energy overhead is a major concern of PhantomCache because it multiplies the number of requests in every access. The energy consumption of a cache is mainly composed of two parts: static power and dynamic power. Static power is generally referred as leakage power of the cache and dynamic power is consumed when the cache is accessed [7]. PhantomCache only affects the dynamic power of LLC because it only increases the cache activity during cache access. Regarding dynamic power, PhantomCache is similar to a normal set-associative cache with an r times higher associativity. We evaluate the case of 16MB, 8-bank, 16-way LLC with 8 candidate sets and compare it with a baseline LLC with no candidate sets.

TABLE IV. POWER CONSUMPTION OF PHANTOMCACHE IN COMPARISON WITH BASELINE USING A 16 MB 8-BANK, AND 16-WAY LLC. PHANTOMCACHE USES 8 CANDIDATE SETS.

LLC	Power Consumption (watt)		
	static power	dynamic power	overall power
Baseline	7.69	1.21	8.89
PhantomCache	7.91	6.97	14.87

We use Xilinx Vivado 2018.2 [14] to estimate static energy overhead of PhantomCache. Without loss of generality, we choose the device xc7k325tffg676 of the Xilinx Kintex-7 family as an instance. The static energy overhead arises from the mapping circuit PhantomCache introduces for mapping an address to r candidate sets. The static power consumption of the introduced mapping circuit is only 0.22 W ($r = 8$), which is marginal in comparison with 7.69 W consumed by the entire baseline 8-bank LLC. Note that the analysis using Xilinx Vivado emulates an FPGA while PhantomCache is to be implemented on an ASIC. This means that the estimation result is an overestimation.

We then use CACTI 6.0 [31] to estimate dynamic energy overhead of PhantomCache. To estimate the dynamic power consumption of PhantomCache, we use CACTI to evaluate the energy consumption of read and write operations in a 128-way set-associative LLC. It depends on two factors. One is the respective power consumption of a single read or write access. The other is the respective count of read or write accesses generated by a benchmark. In general, more accesses consumer more power for any LLC including both the baseline and PhantomCache. Since PhantomCache performs each read access by searching all r candidate sets, the more read accesses a benchmark contains, the more power PhantomCache consumes than the baseline does. Based on the performance measurement in Section VII-G, we use ten mixed workloads with each includes 1-billion instructions from 8 randomly selected benchmarks. They are sufficiently generic in terms of both diversity and scale. In total, these workloads generate 51,686,150 read accesses and 34,842,011 write accesses over the baseline LLC. For PhantomCache, the number of read accesses is 51,844,724 and the number of write accesses is 34,701,023. It introduces 0.31% more read accesses and 0.40% less write accesses. On average, a mixed workload consumes 1.21 W on the baseline LLC and 6.97 W on PhantomCache.

Table IV summarizes the power consumption of PhantomCache in comparison with the baseline. PhantomCache consumes 67.27% more power than the baseline LLC.

This result is counterintuitive because it seems that PhantomCache should have consumed multiple times of energy. Two reasons may account for the result. First, the extra requests produced by PhantomCache only access the tag array in the cache to search for the target block while data array is not accessed. Second, the static power of PhantomCache remains nearly the same as the traditional cache, which mitigates the impact of increased dynamic power.

VIII. CONCLUSION

We have studied the idea of exploiting localized randomization against conflict-based cache timing attacks. It proves to have the same strong defense effect as global randomization countermeasures and avoids the inefficient mechanisms in preceding global randomization designs such as random replacement and dynamic remapping. We implement localized randomization through PhantomCache. The analysis of its security shows that the attacker cannot successfully launch a conflict-based cache timing attack within 100 years when the degree of PhantomCache is set to 8. Finally, we implement PhantomCache using ChampSim and the evaluation shows that PhantomCache only brings a 0.50% performance degradation and affordable hardware overhead.

ACKNOWLEDGMENT

The work is supported in part by The Natural Science Foundation of Zhejiang Province under Grant No. LY19F020050, National Natural Science Foundation of under Grant No. 61772236, Zhejiang Key R&D Plan under Grant No. 2019C03133, Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Research Institute of Cyberspace Governance in Zhejiang University, and Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang. We would like to sincerely thank NDSS 2020 Chairs and Reviewers for their review efforts and helpful feedback. All the thoughtful and constructive comments have guided us toward a much higher paper quality. We would also like to extend our gratitude to Bowen Huang and Seetal Potluri for insightful discussions on PhantomCache performance. Finally and foremost, Kai Bu wholeheartedly appreciates all the students from the Computer Architecture classes for their support and encouragement.

REFERENCES

[1] "The champsim simulator. https://github.com/champsim/champsim."

[2] "Intel digital random number generator (drng) software implementation guide. https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide."

[3] "Spec cpu2017 home page: www.spec.org/cpu2017."

[4] S. Banik, A. Bogdanov, and F. Regazzoni, "Exploring energy efficiency of lightweight block ciphers," in *International Conference on Selected Areas in Cryptography*. Springer, 2015, pp. 178–194.

[5] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[6] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *Latincrypt*, 2012, pp. 159–176.

[7] S. Chakraborty, D. Deb, D. Buragohain, and H. K. Kapoor, "Cache capacity and its effects on power consumption for tiled chip multiprocessors," in *2014 International Conference on Electronics and Communication Systems (ICECS)*. IEEE, 2014, pp. 1–6.

[8] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou, "Detecting covert timing channels with time-deterministic replay." in *OSDI*, 2014, pp. 541–554.

[9] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *MICRO*, 2014, pp. 216–228.

[10] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.

[11] P. Damaschke, "Threshold group testing," in *General theory of information transfer and combinatorics*, 2006, pp. 707–718.

[12] P. Deepthi and P. Sathidevi, "Design, implementation and analysis of hardware efficient stream ciphers using lfsr based hash functions," *Computers & Security*, vol. 28, no. 3-4, pp. 229–241, 2009.

[13] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani, "Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels," in *HOST*, 2018, pp. 187–190.

[14] T. Feist, "Vivado design suite," *White Paper*, vol. 5, p. 30, 2012.

[15] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *USENIX Security Symposium*, 2017, pp. 217–233.

[16] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches." in *USENIX Security Symposium*, 2015, pp. 897–912.

[17] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

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

[19] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of the 11th ACM on Asia conference on computer and communications security*. ACM, 2016, pp. 353–364.

[20] H. Krawczyk, "Lfsr-based hashing and authentication," in *CRYPTO*, 1994, pp. 129–139.

[21] D. Levinthal, "Performance analysis guide for intelr core (tm) i7 processor and intel xeon (tm) 5500 processors," *Intel Performance Analysis Guide*, 2009.

[22] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices." in *USENIX Security Symposium*, 2016, pp. 549–564.

[23] 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 *HPCA*, 2016, pp. 406–418.

[24] F. Liu and R. B. Lee, "Random fill cache architecture," in *MICRO*, 2014, pp. 203–215.

[25] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[26] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *S&P*, 2015, pp. 605–622.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6, 2005, pp. 190–200.

[28] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 118–129, 2012.

[29] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 48–65.

[30] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer, "Hello from the other side: Ssh over robust cache covert channels in the cloud," *NDSS*, 2017.

[31] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[32] K. Olukotun, T. Mudge, and R. Brown, "Performance optimization of pipelined primary cache," in *ISCA*, 1992, pp. 181–190.

[33] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *CCS*, 2015, pp. 1406–1418.

[34] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *MICRO*, 2018.

[35] ——, "New attacks and defense for encrypted-address cache," in *ISCA*, 2019, pp. 360–371.

[36] A. Seznec, "A case for two-way skewed-associative caches," 1993, pp. 169–178.

[37] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE MICRO*, vol. 36, no. 2, pp. 34–46, 2016.

[38] V. Varadarajan, T. Ristenpart, and M. M. Swift, "Scheduler-based defenses against cross-vm side-channels." in *USENIX Security Symposium*, 2014, pp. 687–702.

[39] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in xen," in *CCSW*, 2011, pp. 41–46.

[40] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *S&P*, 2019.

[41] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: secure dynamic cache partitioning for efficient timing channel protection," in *DAC*, 2016.

[42] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, 2007, pp. 494–505.

[43] ——, "A novel cache architecture with enhanced performance and security," in *MICRO*, 2008, pp. 83–93.

[44] M. Weiß, B. Heinz, and F. Stumpf, "A cache timing attack on aes in virtualization environments," in *FC*, 2012, pp. 314–328.

[45] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *USENIX Security*, 2019.

[46] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks," in *ISCA*, 2017, pp. 347–360.

[47] M. Yan, Y. Shalabi, and J. Torrellas, "Replayconfusion: detecting cache-based covert channel attacks using record and replay," in *MICRO*, 2016.

[48] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack." in *USENIX Security*, vol. 1, 2014, pp. 22–25.