TreasureCache: Hiding Cache Evictions Against Side-Channel Attacks

Mengming Li, Kai Bu, *Member, IEEE*, Chenlu Miao, and Kui Ren, *Fellow, IEEE*

***Abstract*—Cache side-channel attacks remain a stubborn source of cross-core secret leakage. Such attacks exploit the timing difference between cache hits and misses. Most defenses thus choose to prevent cache evictions. Given that two possible types of evictions—flush-based and conflict-based—use different architectural features, these defenses have to integrate hybrid defense strategies, incur OS modification, and sacrifice performance to completely throttle cache side-channel attacks. In this article, we present TreasureCache against cache side-channel attacks without modifying OS or sacrificing performance. Instead of preventing cache evictions with various costs, we advocate to allow cache evictions as is and hide exploitable evictions in our specialized small eviction-hidden buffer. The buffer guarantees a fast hit time comparative to LLC hits. This instantly closes the timing gap between accessing exploitable blocks when they are in and out of the LLC. Moreover, with the help of our buffer, we no longer have to disable flush instructions or shared memory. A lightweight constant-time flush instruction can help TreasureCache to prevent both flush-based and conflict-based side-channel attacks. We validate TreasureCache security and performance through extensive experiments. With a hardware overhead of less than 0.5%, TreasureCache reduces the secret-leakage resolution by about 1,000 times without introducing any performance slowdown.**

***Index Terms*—Cache side-channel attack, eviction-hidden buffer, secure replacement policy.**

I. Introduction

CACHE side-channel attacks that exploit timing channels caused by LLC evictions remain a stubborn source of cross-core information leakage [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. The timing difference arises from different access latencies of cache hits and cache misses. To infer the victim's access behavior from access latency measurements, the attacker needs to first evict victim data from the cache hierarchy. Whether the victim will reaccess the evicted data or not leads to differentiable access latency for the attacker's subsequent cache accesses. According to the eviction strategy, cache side-channel attacks can be classified into two types—flush-based and conflict-based [14], [15], [16], [17], [18], [19], [20], [21], [22]. Flush-based attacks enable the attacker to directly evict victim data using flush instruction (e.g., `clflush` in x86). Conflict-based attacks indirectly evict victim data using eviction sets. An eviction set consists of sufficient conflicting addresses that map to the same cache set with victim data [1], [23]. By accessing these conflicting addresses, the attacker can fully occupy the target cache set and evict victim data therein via cache replacement.

Table I classifies typical cache side-channel attacks. Next, we detail the essence of existing defenses mostly by preventing cache evictions.

A. Prevention of Cache Evictions

Since the root cause for side-channel attacks is evicting victim data, a driving strategy for defenses is to prevent cache evictions. The disparity in the exploited hardware logics renders most defenses hardly effective against both flush-based and conflict-based evictions. They usually need to unify hybrid defenses. Cache partitioning promises a stand-alone defense yet necessitates OS modification and sacrifices performance gains that the shared LLC offers [24], [25], [26], [27].

Prevention of Flush-Based Eviction: Given that flush-based evictions require both flush privilege and shared data (Section II-A), disabling either of these requirements can prevent the attacker from evicting victim shared data. For example, SHARP [15] prohibits flushing read-only or executable data in user mode. However, such defenses are found vulnerable to malicious privileged code (e.g., OS kernel and hypervisor) that exploits Flush+Reload [27]. The other line of flush countermeasures turn to throttle shared data between the attacker and victim across security domains. For example, The copy-on-access defense [18] and ScatterCache [19] require OS-assistance to identify security domains for processes and then duplicate shared data accessed by different security domains. Once the attacker and victim reside in different security domains, the attacker's

Manuscript received 12 February 2023; revised 19 October 2023; accepted 10 January 2024. Date of publication 16 January 2024; date of current version 4 September 2024. This work was supported in part by the National Key R&D Program of China under Grant 2020AAA0107705, in part by the National Natural Science Foundation of China under Grant 62032021, in part by Zhejiang Key R&D Plan under Grant 2019C03133, in part by Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang under Grant 2018R01005, and in part by the Alibaba-Zhejiang University Joint Institute of Frontier Technologies, and Research Institute of Cyberspace Governance in Zhejiang University. *(Corresponding author: Kai Bu.)*

Mengming Li is with the School of Software Technology, Zhejiang University, Hangzhou 310027, China, and also with ZJU-Hangzhou Global Scientific and Technological Innovation Center, Hangzhou 311215, China (e-mail: mmli@zju.edu.cn).

Kai Bu is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and also with ZJU-Hangzhou Global Scientific and Technological Innovation Center, Hangzhou 311215, China (e-mail: kaibu@zju.edu.cn).

Chenlu Miao is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: clmiao@zju.edu.cn).

Kui Ren is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and also with the Zhejiang Provincial Key Laboratory of Blockchain and Cyberspace Governance, Hangzhou 310027, China (e-mail: kuiren@zju.edu.cn).

Digital Object Identifier 10.1109/TDSC.2024.3354991

TABLE I
CLASSIFICATION OF CACHE SIDE-CHANNEL ATTACKS EXPLOITING LLC EVICTIONS [14], [15], [16], [17], [18], [19], [20]

Type	Attack	Eviction Scheme
Flush-based	Flush+Reload [3] Flush+Flush [2]	Flush Instruction
Conflict-based	Evict+Reload [4] Prime+Probe [1] Evict+Time [32]	Eviction Set

flush instructions and the victim's accesses hit different copies of the shared data. Therefore, the attacker can no longer infer the victim's access behavior.

Prevention of Conflict-Based Eviction: Defenses against conflict-based evictions aim to mitigate eviction set construction [14], [16], [17], [19], [28] or usage [15], [29], [30]. Given that eviction set construction exploits deterministic mapping from addresses to cache sets, randomized mapping inspires a series of defenses against eviction sets. Representative defenses such as CEASER [14], CEASER-S [28], ScatterCache [19], PhantomCache [16], and MIRAGE [17] randomize the mapping function of addresses and cache set indices. They render the time for constructing an eviction set as unbearably long as over hundreds of years. Randomized mapping, however, requires modification of the address mapping logic and leads to both logic and performance overhead [16].

Some other defenses do not directly throttle eviction set construction. Instead, they choose to mitigate the eviction effect by eviction sets. For example, SHARP [15] modifies the cache replacement policy such that the LLC data selected to evict do not reside in the victim's private caches. This way, the eviction effect leads to no cache miss for the victim and leaves no exploitable timing difference for the attacker. However, SHARP is found vulnerable to the Prime+Reprime+Probe attack [31]. Furthermore, RIC [29] and BITP [30] avoid such exploitable cache misses by mitigating the back-invalidation effect. RIC [29] relaxes the cache inclusion policy and requires that not all evicted LLC data be associated with their copies' invalidation in private caches. BITP [30] conforms to the inclusion policy yet prefetches back-invalidated data to negate the back-invalidation effect.

B. Lack of Efficient Unified Defense

As we investigated in Section I-A, it is highly desirable to explore an efficient unified defense against both flush-based [15], [18], [19] and conflict-based evictions [14], [15], [16], [17], [19], [28], [29], [30]. Table II compares properties of various defenses and demonstrates why existing defenses can barely offer an efficient unified solution. Most defenses target either flush-based or conflict-based evictions instead of both. Sporadic unified defenses such as SHARP [15] and ScatterCache [19] combine both types of defending techniques to jointly mitigate both evictions. They impose not only various hardware modifications (e.g., addressing mapping logic and cache replacement logic) but also software modification (e.g., OS assistance), or being found vulnerable to certain new attacks. Cache partitioning [24], [25], [26], [27] even further incurs strict isolation on cache organization and turns the shared LLC into a non-shared resource. This renders the existing unified defense associated with unbearable performance slowdown.

TABLE II
QUALITATIVE COMPARISON OF TREASURECACHE WITH EXISTING SIDE-CHANNEL–ATTACK DEFENSES THAT PREVENT CACHE EVICTIONS

Legend: FE: Flush Effective; CE: Conflict Effective; PSF: Performance-Slowdown Free; OAF: OS-Assistance Free

Solutions	FE	CE	PSF	OAF
Flush Restriction [15]	✓	✗	✗	✗
Duplication [18], [19]	✓	✗	✗	✗
Randomization [14], [16], [17], [19], [28]	✗	✓	✗	✓
RIC [29]	✗	✓	✓	✗
BITP [30]	✗	✓	✓	✓
SHARP# [15]	✓*	✓*	✗	✗
ScatterCache# [19]	✓	✓	✗	✗
Partition [24], [25], [26], [27]	✓	✓	✗	✗
TreasureCache	✓	✓	✓	✓

Notes: #SHARP [15], ScatterCache [19] adopt hybrid designs to mitigate flush-based evictions and conflictbased evictions. *SHARP's effectiveness towards flushbased evictions and conflict-based evictions are respectively impaired by privileged Flush+Reload attack [27] and Prime+Reprime+Probe attack [33].

C. Our Contribution: TreasureCache

In this paper, we present TreasureCache as the first step toward an efficient unified defense. In comparison with existing defenses (Table II), TreasureCache prevents both flush-based and conflict-based evictions yet without performance slowdown or OS assistance. It breaks this long-standing barrier from a new perspective. That is, we shun away from wrestling with eviction prevention. We turn to allow cache evictions as is but hide exploitable evictions in a specialized small eviction-hidden buffer. Buffered exploitable evictions no longer induce the attack vector as when they have to be reaccessed from memory. Reloading them from the buffer offers a comparative latency as LLC hits. This prevents them from leading to the broadly exploited timing channel over LLC hits and misses.

We address various challenges for such a small buffer to guarantee efficient protection. Two representative challenges bring forth the following questions.

- How to filter irrelevant non-exploitable evictions out of a large number of LLC evictions?
- How to prevent the attacker from maliciously evicting exploitable evictions out of the buffer?

To address the first challenge, we propose a secure placement policy to identify exploitable evicted blocks that associate with copies in private caches. This condition takes advantage of the fact that in attacks such as Flush+Reload, Evict+Reload, and Evict+Time, the LLC blocks to be evicted are usually cached in the victim's private caches. We also identify exploitable evicted blocks that are replaced by reloads of buffered blocks. This condition targets Prime+Probe and its variants such as Prime+Reprime+Prime [31], [33]. To address the second challenge, we propose a secure replacement policy to restrict each core from replacing the buffered blocks. Each buffered block is assumed to be owned by certain cores. Only buffered blocks with

no bound ownership can be replaced. Such a secure replacement policy prevents the attacker to revoke the victim core's ownership on buffered exploitable blocks. Finally, we incorporate a lightweight constant-time flush scheme against Flush+Flush, without having to disable flush instructions or shared memory.

In summary, we make the following major contributions to efficiently securing caches against both flush-based and conflict-based side-channel attacks.

- We advocate allowing cache evictions as is but hide them in a small specialized eviction-hidden buffer (Section IV). The buffer replaces traditional LLC misses due to exploitable evictions with buffer hits that are comparatively fast as LLC hits. It thus throttles the exploitable timing difference between LLC hits and misses as well as the so caused side-channel attacks. Such a defense no longer suffers from various software or hardware overhead as in existing eviction-prevention defenses.
- We present TreasureCache to leverage the eviction-hidden buffer toward a complete and efficient defense against both flush-based and conflict-based attacks (Section V). We explore a series of strategies to make our small buffer suffice for security and efficiency.
- We implement TreasureCache using the gem5 simulator (Section VI) and validate its security (Section VII) and performance (Section VIII-A) through extensive analytical and experiment results. With a hardware overhead of less than 0.5% (e.g., a 64 KB buffer), TreasureCache reduces the secret-leakage resolution by about 1,000 times without introducing any performance slowdown.

II. Attack

In this section, we review LLC side-channel attacks for ease of understanding TreasureCache effectiveness. They can be classified into flush-based attacks (i.e., Flush+Reload [3] and Flush+Flush [2]) and conflict-based attacks (i.e., Evict+Reload [4], Prime+Probe [1], and Evict+Time [32]).

A. Flush-Based Attacks

Both Flush+Reload [3] and Flush+Flush [2] attacks require flush privilege and shared data. Specifically, the attacker need be allowed to invoke flush instructions and to share data with the victim. This is why the attacker can directly use flush instructions to evict victim shared data with specific addresses. Flush instructions are readily supported on most Intel processors. Shared data can be obtained through shared library or memory deduplication [2], [3], [4], [18], [34].

Flush+Reload [3] iteratively evicts and reaccesses victim shared data. Each round of iteration consists of three steps.

- *Step 1. Flush:* The attacker uses flush instructions such as `clflush` to evict victim shared data (if cached) out of the cache hierarchy.
- *Step 2. Wait:* The attacker idles for a predefined timing interval, in which the victim may or may not access the victim shared data.
- *Step 3. Reload:* The attacker reloads the victim shared data and measures the access latency using timing instructions such as `rdtsc`. Then the attacker uses the measured latency to infer whether the victim has accessed the victim data during the waiting period. If the victim accessed the victim data therein, the attacker enjoys an LLC hit and thus fast access while reloading. Otherwise, since the attacker has already evicted the victim data in step 1, it encounters an LLC miss that associates with a relatively long access latency.

Flush+Flush [2] repeatedly issues flush instructions over victim shared data toward a faster attack speed than that of Flush+Reload. It removes the Reload step of Flush+Reload. The key motivation is that flushing cached data is noticeably slower than flushing uncached data. Therefore, by repeatedly flushing victim data and measuring flush latency, the attacker can use the flush speed to infer the victim's access behavior during the Wait step.

B. Conflict-Based Attacks

Conflict-based attacks apply to scenarios when shared data and flush privilege are not simultaneously satisfied. They need to construct eviction sets with sufficient conflicting addresses, the accesses of which evict victim data from caches.

Evict+Reload [4] replaces the Flush step of Flush+Reload with Evict due to lack of privilege for using flush instructions. Without flush instructions, the attacker employs an eviction set to evict victim shared data. In the Evict step, the attacker accesses the conflicting addresses in the eviction set to make sure that victim shared data are evicted out of the LLC. Leveraging cache inclusiveness, the LLC controller further issues a back-invalidation command to invalidate the copies of victim shared data in the victim's private caches.

Prime+Probe [1] no longer requires shared data between the attacker and victim. Prime+Probe iterates as follows.

- *Step 1. Prime:* As with the Evict step of Evict+Reload, the attacker accesses conflicting addresses in a pre-constructed eviction set. Since these conflicting addresses map to the same LLC set with that of victim data, the Prime step helps to fully occupy the set and evict victim data out of the LLC. Furthermore, back invalidation ensures the eviction of victim data from the victim's private caches.
- *Step 2. Wait:* The attacker behaves the same as in the Wait step of Flush+Reload.
- *Step 3. Probe:* The attacker accesses conflicting addresses in the eviction set again and measures the access latency. The measured latency indirectly indicates whether the victim has accessed victim data in the Wait step. If the victim has not accessed, all data loaded in the Prime step are still cached, yielding all cache hits and thus a fast probe speed. In contrast, if the victim has accessed the victim data in the Wait step, some loaded data in the Prime step should have been replaced and thus uncached. This leads to a cache miss and a slower probe speed than when the victim has conducted no access.

Evict+Time [32] measures the execution time of the victim rather than that of the attacker itself. The attacker periodically

evicts victim data (if cached) by accessing an eviction set of conflicting addresses that map to the same LLC set with that of victim data. In between two Evict steps is a Time step, which triggers victim execution and measures the execution time. If the victim executes slower than a pre-measured threshold with cache hits on the victim data, the attacker can infer that the victim has accessed the victim data during execution. This is because that the victim data have been evicted in the Evict step; accessing them in the Time step leads to a cache miss and thus slows down the victim execution. In the other case, if the victim takes a comparative execution time with the threshold, it is not affected by the eviction effect of victim data. The attacker can thus infer that the victim has not accessed the victim data in the Time step.

III. Threat Model

We adopt a threat model broadly used in cross-core side-channel attacks exploiting LLC evictions. The attacker and the victim run simultaneously on different physical cores sharing an inclusive LLC [1], [3], [14], [15], [17], [19], [29], [30]. The number of cores manipulated by the attacker can be also unlimited, as long as they differ from the victim core. The attacker can deploy any side-channel attacks reviewed in Section II to monitor the victim's memory access pattern and further infer the victim's secret through the pattern. For instance, the attacker can use flush instructions in user space to execute the routine of flush-based attacks such as Flush+Reload and Flush+Flush. It can also construct the eviction set to conduct conflict-based attacks such as Evict+Reload, Prime+Probe, and Evict+Time. Since the attacker and victim are resident in different cores, we regard each core as a security domain. TreasureCache aims to prevent the attacker in one security domain from exploiting cross-core LLC evictions to leak the victim's secret from another security domain.

We do not consider same-core attacks where the attacker shares the same core with the victim. For example, if the attacker running in one process intends to leak secrets from the victim belonging to another process but on the same core, it usually uses the L1 cache to construct the side channel [32], [35], [36], [37], [38]. Prevention of such same-core attacks are broadly investigated. For example, there are efficient schemes for protecting the L1 cache [39], [40], [41] but not suitable for applying to the LLC. Beyond existing schemes targeting the LLC-based cross-core attacks [14], [16], [17], [19], [28], [29], [30] as reviewed in Section I-B, TreasureCache aims to minimize impact on system performance. Furthermore, cache attacks that exploit the replacement policy [42], [43] and coherence protocol [34] are also out of our scope. They can be respectively throttled by unexploitable replacement policies [15], [44], [45] or safe coherence protocols [46], [47], [48].

IV. TreasureCache

In this section, we present TreasureCache as the first lightweight and unified architectural solution against eviction-based cache side-channel attacks. TreasureCache circumvents various complexities that traditional solutions wrestle with for avoiding LLC evictions. It allows LLC evictions to take place yet hides them in a small buffer near the LLC. The eviction-hidden buffer supports fast access and throttles the exploitable latency difference between LLC and memory accesses.

A. Motivation

We observe that complexity and overhead of existing defenses are mainly attributed to their inevitable hardware and software modifications for preventing cache evictions. To launch a successful side-channel attack, the attacker needs to first evict victim data out of the cache hierarchy. Then the attacker measures the access latency of its own (e.g., Prime+Probe) or that of the victim (e.g., Evict+Time) to infer whether the victim has loaded the evicted victim data back. No wonder, preventing victim data from being evicted can throttle side-channel attacks. However, as aforementioned in Section II, two sources of cache evictions are flush instructions and eviction sets that exploit `clflush`-alike instructions and deterministic address mapping, respectively. Both features are critical for coherence maintenance and performance speedup on modern processors. Simply disabling them prevents cache evictions at the expense of logic modification and performance slowdown.

In this paper, we are motivated to tackle eviction-based side-channel attacks from a new perspective—we still allow LLC evictions but hide potentially exploitable evictions in a newly introduced lightweight eviction-hidden buffer instead. Such an eviction hiding technique instantly promises both efficiency and security. First, by allowing victim-data evictions, we do not have to struggle with the unavoidable overhead by system modifications for preventing evictions. Managing a small eviction-hidden buffer is much more efficient than modifying OS as well as various cache layers. Second, when the attacker measures the access latency of victim data (e.g., Flush+Reload, Flush+Flush, Evict+Reload, Evict+Time) or data evicted by the victim data (e.g., Prime+Probe), the access request can be served directly from the fast eviction-hidden buffer instead of slow memory. This vanishes the traditionally exploitable timing difference and prevents side-channel attacks.

B. Challenges

The ultimate challenge for TreasureCache is how to use a small eviction-hidden buffer to hold a large number of potentially exploitable evictions. For simplicity, we may put all evicted LLC-blocks in the buffer, which expands to a large size for sure. However, the buffer has to be sufficiently small to guarantee a fast access speed. Otherwise, accesses over the buffer and the LLC may still induce a noticeable timing channel. Furthermore, given a limited buffer size, it is inevitable that some buffered blocks will be eventually evicted. Once this is the case, the hidden eviction becomes exposed and the side-channel attack revives.

C. Strategies

We address the preceding challenges by implementing a fully-associative eviction-hidden buffer with our specialized secure placement and replacement policies.

1) Indexed Fully-Associative Architecture: The first-to-go design strategy is definitely adopting a fully-associative architecture for the eviction-hidden buffer. In comparison with the other two architectural choices—direct-mapped and set-associative, fully-associative maximizes space utilization. It replaces a buffered block only when there is no empty slot left. As an optimization trick, we also devise an indexing scheme to map the physical memory address to the index of each buffered block and its states we maintain for enforcing secure replacement policies.

2) Secure Placement Policy: We solve the challenge of constrained buffer size by identifying exploitable evicted blocks out of many LLC evictions (Section V-C2). For example, given that side-channel attacks leverage cache inclusiveness [14], [15], [16], [17], [18], [19], [20], we consider an evicted block exploitable if it associates with copies in private caches of at least one core or is replaced by blocks loaded from the eviction-hidden buffer. Our specification of exploitable evicted blocks greatly helps to filter non-exploitable evictions from entering the buffer. We show its filtering effectiveness using SPEC CPU 2017 benchmarks in Section VII-C. Only about 0.09 exploitable evicted blocks owned by each core will be buffered in 10,000 cycles on average. This also addresses the concern about in-buffer eviction caused by normal programs.

However, as aforementioned in Section IV-B, the attacker may deliberately generate exploitable evicted blocks to flood the eviction-hidden buffer. A sophisticated attacker could even spice up this by generating evictions on multiple cores. We should prevent such exploitable evicted blocks from arbitrarily replacing buffered blocks. We present a specialized secure replacement policy to cope with this concern.

3) Secure Replacement Policy: The specialized secure replacement policy aims to prevent the attacker from maliciously evicting buffered exploitable blocks (Section IV-C3). Regarding each core as a security domain, we notice that a buffered block is owned by at least one security domain. The secure replacement should conform to the following restrictions.

- One security domain can only own a limited number of blocks in the buffer. The specific value is defined as the TreasureCache parameter.
- If buffering a new block block makes the number of blocks owned by one security domain exceed the predefined value, the security domain should release the ownership of an already buffered block.
- If the eviction-hidden buffer has no free space for buffering a new block, only buffered blocks owned by no security domain can be replaced.

This secure replacement is achieved by making the victim's security domain always own the exploitable blocks and preventing the attacker to unilaterally release the victim's ownership on them. To this end, TreasureCache assigns two types of ownership to the LLC evictions and sends them along with data blocks to the eviction-hidden buffer. The first is its evictor, representing the core that has evicted the block into the buffer. The other associates with its sharers, including any core whose private caches hold a copy of this block upon its LLC eviction. This way, either in the Prime+Probe (victim as evictor) or other attacks covered in Section II (victim as sharer), the victim's security domain can own the exploitable blocks. Such ownership regulation also prohibits the attacker from unilaterally crafting LLC evictions with the victim core as evictor or sharers to maliciously release the victim's ownership on buffered exploitable blocks.

V. Design

In this section, we detail the design of TreasureCache. It explores a series of efficient techniques to enforce our specialized placement and replacement policies over LLC evictions within the small eviction-hidden buffer. Exploitable LLC evictions are buffered and can be reloaded directly to the LLC upon an LLC miss that hits in the buffer. We make them hard to be maliciously evicted out of the buffer by the attacker. They thus become unexploitable because of no longer leading to distinguishable access latency as when they were reloaded from memory.

A. Architecture

The newly introduced eviction-hidden buffer locates between the LLC and memory. It is shared by all CPU cores and transparent to original memory access logics. The buffered data are potentially exploitable evicted blocks filtered from many more LLC evictions. They are neither strictly inclusive nor exclusive of data in the cache hierarchy. In other words, a block replaced from the eviction-hidden buffer does not necessarily generate back-invalidation to the LLC and higher-level caches. How the buffer interacts with the LLC and memory relies on four key buffer components.

- *Data Store* stores the data content for each buffered exploitable evicted block. Buffered blocks are organized in a fully associative way.
- *Secure Placement Policy* enforces conditions to identify exploitable evicted blocks out of LLC evictions. Only evicted blocks that satisfy the conditions therein need to be buffered.
- *Secure States* record the metadata for enforcing the secure replacement policy. For example, they track the block owners and the number of buffered blocks each core associates with. These states are updated upon a new block is added to the buffer or the secure replacement policy is enforced.
- *Secure Replacement Policy* leverages secure states to avoid buffered blocks with bound owner from being replaced. This prevents the attacker to deliberately evict exploitable blocks from the buffer.

B. Methodology

The eviction-hidden buffer interacts with the LLC upon an LLC eviction or an LLC miss.

Handling of LLC Evictions: This process aims to accurately identify exploitable evicted blocks along with their secure states.

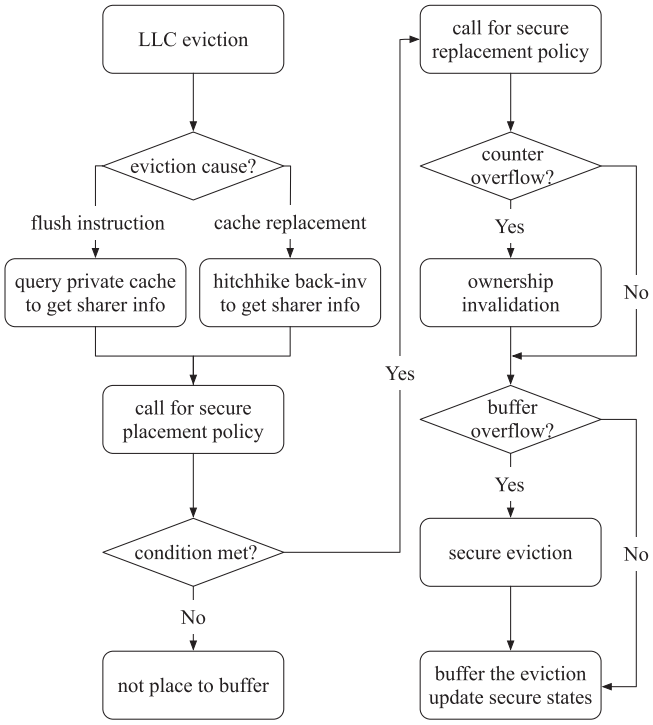

Fig. 1. TreasureCache policy for secure handling of LLC evictions.

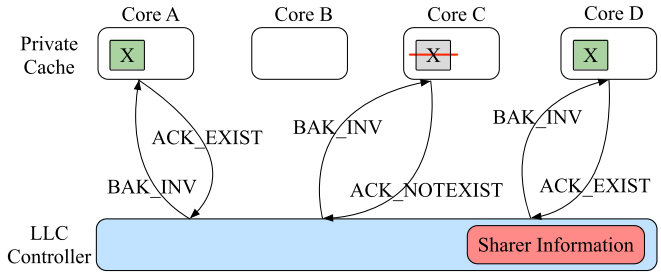

Fig. 2. Acquisition of sharer information for LLC evictions caused by cache replacement.

Specifically, useful states of an evicted block include its evictor (i.e., the CPU core that evicts this block) and sharers (i.e., the CPU cores whose private caches hold a copy of this block). We hitchhike the back-invalidation process upon LLC evictions to efficiently acquire the sharer information (Section V-C1). They enable the secure placement policy to decide whether this evicted block need be buffered. Once the decision favors buffering, the secure replacement policy may be further invoked if no available slot exists.

Handling of LLC Misses: We use buffered blocks to serve LLC misses. If the memory request that misses in the LLC can hit in the buffer, the buffer can send the requested data to the cache hierarchy and the requesting core from the buffer immediately. If the requested block still misses in the buffer, we forward the request to the lower level memory.

C. LLC Eviction Handling

LLC eviction handling involves with both our specialized secure placement and replacement policies. Fig. 1 sketches the workflow for TreasureCache to handle LLC evictions.

1) Sharer Information Acquisition: To process an LLC eviction, we need to determine the evicted block's sharers as critical arguments for buffer management policies. In directory-based coherence protocols [49], each cached block is associated with a bit vector with as many bits as cores. These bits are called core valid bits because they indicate on which cores' private caches the block may be cached [50], [51]. Specifically, the ith core valid bit of a block is set if the block has been requested by the ith core, which has not explicitly sent invalidation to the directory yet. It seems that we can directly use core valid bits to determine the current sharers of a block. However, these bits may be stale due to silent eviction of clean blocks in private caches [52]. Existing solutions have to query private caches of each core with the core valid bit set to obtain precise sharer information [15], [53]. If we simply apply this query to every LLC eviction as long as the evicted block has nonzero core valid bits, TreasureCache would suffer from non-negligible performance overhead.

In order to guarantee TreasureCache efficiency, we only query private caches for sharer information when necessary. In common cases, we hitchhike the back-invalidation process. They respectively apply to two sources of LLC evictions, flush instructions and cache replacement.

First, for LLC evictions caused by flush instructions, since the implementation specifics of `clflush` in x86 are proprietary, we conservatively query private caches to acquire the sharer information.

Second, for LLC evictions caused by cache replacement, we hitchhike the back-invalidation process for efficiently acquiring sharer information. Specifically, we introduce fine-grained acknowledgement responses (in reaction to invalidation) from private caches to the LLC controller. In the traditional acknowledgement design, once a block is about to be evicted, the LLC controller sends the back-invalidation command (`BAK_INV`) to private caches of any core with a set core-valid-bit for the block. Then all the commanded private caches reply the LLC controller with `ACK` regardless of whether they hold a copy of the requested block or not (due to silent eviction). We introduce a finer-grained `ACK` through two messages—`ACK_EXIST` and `ACK_NOTEXIST`—for private caches to respectively indicate existence and non-existence of the requested block. As shown in Fig. 2, only when the LLC receives `ACK_EXIST` can it record the core sending `ACK_EXIST` as a sharer. Such a finer-grained `ACK` brings no extra complexity to the critical path of cache replacement.

2) Secure Placement Policy: The first category of exploitable evicted blocks to buffer should associate with copies in one or more private caches. This condition takes advantage of the fact that in attacks such as Flush+Reload, Evict+Reload, and Evict+Time, LLC blocks to be evicted are almost always cached in the victim's private caches. For flush-based attacks, the condition holds because of the property that a flush request arrives at the LLC first before being routed to higher level caches [2]. Flush handling triggers a series of coherence requests from the LLC to private caches where the block-to-flush still exists. For

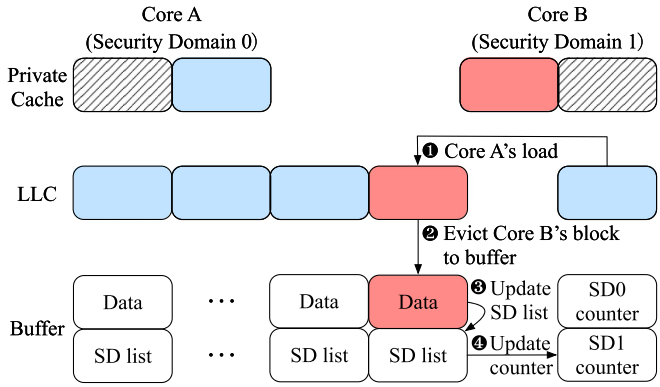

Fig. 3. Secure replacement process.

conflict-based attacks using eviction sets [15], [29], [30], back invalidation need be issued after an LLC eviction to eventually evict related copies from private caches.

Furthermore, we should also buffer another category of exploitable evicted blocks that are replaced by blocks loaded from the eviction-hidden buffer. Given the first condition, the evicted blocks accessed by the victim can always be buffered. However, in the Prime+Probe attack, the attacker probes its own eviction set to indirectly infer the behavior of exploitable blocks, rather than load the victim's block. A sophisticated attacker may employ the Prime+Reprime+Probe attack [31], [33] to bypass the first condition. In particular, the attacker manages to make the exploitable blocks only reside in the LLC and then waits for the victim's memory access. Once the victim reloads a buffered block to LLC, some of the attacker's in-LLC–only block will be replaced. We buffer these blocks as well to avoid LLC hits and misses they induce for the attacker to exploit otherwise.

3) Secure Replacement Policy: As discussed in Section IV-C3, our secure replacement policy specialized inside the buffer aims to prevent the attacker from evicting buffered exploitable blocks. Along with sharers and evictors, we introduce three secure states—`security_domain_list` per buffered block, `counter` per core, and one `global_threshold`—for policy enforcement. Fig. 3 sketches the buffer organization augmented with the secure states. Together they can bind each block to one or more cores. No core can unilaterally replace a buffered block that is still bound to some other cores. A buffered block can be replaced only when all its bound cores have imposed replacement on it. This way, the attacker can hardly revive a side-channel attack through first evicting victim blocks from the LLC and then evicting them from the buffer.

We first define the introduced secure states as follows.

- `security_domain_list`*:* Regarding each core as a security domain, we use `security_domain_list` per block to track the block's evictor and sharers. It is a bit vector with a length equal to the core count. We say that the ith core or security domain owns a block if the ith bit in the block's `security_domain_list` is set. When a new LLC eviction is buffered, we update its evictor and sharers information in the `security_domain_list`.
- `counter`*:* We use a `counter` per core to track the number of buffered blocks owned by each security domain. A core's `counter` increments or decrements upon it has a new owned block or invalidates ownership from one of its owned blocks.
- `global_threshold`*:* Due to limited buffer capacity, we predefine a `global_threshold` to enforce a maximum counter for each security domain.

The secure replacement policy takes effect using the above states in two scenarios. The first is when buffering a new LLC eviction makes one or more security domain's `counter` exceed `global_threshold`. The other is when no empty slot is left for buffering a new LLC eviction. Together, they ensure that any evicted block crafted by the attacker cannot force a buffered block owned by the victim to be replaced.

Replacement Upon Counter Overflow: If a security domain's `counter` is about to exceed `global_threshold`, TreasureCache enforces `ownership_invalidation` that randomly selects one of the security domain's owned blocks and then clears its corresponding bit in the selected block's `security_domain_list`. The security domain then further owns the newly buffered block, which initially causes `counter` overflow.

Replacement Upon Buffer Overflow: When the buffer has no empty slot to accommodate a new evicted block, TreasureCache enforces `secure_eviction` to randomly replace a buffered block with no declared ownership. That is, the selected block to replace should have all bits in its `security_domain_list` set as 0. We can guarantee the existence of such blocks by configuring buffer size greater than or equal to the product of `global_threshold` and core count (i.e., the number of security domains). We now reason about why this condition holds. When $\texttt{buffer\_size} > \texttt{global\_threshold} \times \texttt{core\_count}$, there must exist a block that is not claimed by any core. This is because a core can only own `global_threshold` different blocks in the eviction-hidden buffer. At least a number $\texttt{buffer\_size} - \texttt{global\_threshold} \times \texttt{core\_count}$ of buffered blocks are unclaimed. Once we set $\texttt{buffer\_size} = \texttt{global\_threshold} \times \texttt{core\_count}$, an extreme case occurs when each core owns exactly `global_threshold` different blocks in the buffer and no two cores share the same ownership for any buffered block. The number of owned blocks in the eviction-hidden buffer then reaches the maximal value. If a core expects to own a new block in the buffer, it must first trigger the ownership invalidation process to release its ownership on a previously buffered block and then allocate this empty slot to the new block. The `ownership_invalidation` process hereby ensures the existence of at least one unclaimed block.

D. LLC Miss Handling

LLC miss handling includes two strategies for fast accessing the eviction-hidden buffer.

First, we directly use the LLC controller to manage the eviction-hidden buffer for reducing redundant network traffic.

The data read from the eviction-hidden buffer can be directly sent to the LLC through the LLC controller.

Second, we organize the eviction-hidden buffer as an indexed fully-associative architecture to accelerate data searching (Section IV-C1). We store the data store as well as the secure states using SRAM arrays. For a buffered block, we can use the same index to access its data and secure states. We achieve this by using a lookup table to store the mappings from the physical memory address to the index of buffered data as well as associative secure states. Specifically, we can implement the lookup table with content-addressable memory (CAM) [54] and connect its output port to the data store. When an access request arrives at the eviction-hidden buffer, we use the lookup table to quickly decide whether the requested block is buffered and if yes, obtain the index of it. We then use the index to quickly retrieve the data and secure states of the requested block. Accessing data in the CAM-based eviction-hidden buffer can be performed within two cycles (one cycle for searching the lookup table and another cycle for obtaining the data [41], [54], [55]). This helps us gain a comparative searching speed as in a direct-mapped architecture while enjoying the maximized space utilization with the fully-associative buffer. We suggest that chip-area overhead by the eviction-hidden buffer depends on concrete implementation. However, we can use its requirement of storage capacity to estimate its relative area overhead in comparison with that of cache hierarchies. Section VIII-D shows that the eviction-hidden buffer enforces an affordable storage overhead about only 0.5% of the LLC capacity. Furthermore, since the eviction-hidden buffer serves only LLC misses, which already account for a negligible proportion of overall memory accesses by design. This helps to indicate limited power consumption by the eviction-hidden buffer.

E. Dirty Block Overwriting

It is vital to guarantee that the eviction-hidden buffer dose not affect the program behavior. TreasureCache strictly keeps the data contents in the eviction-hidden buffer consistent with DRAM and I/O devices. Two types of operations about dirty blocks are implemented to maintain such attributes.

First, TreasureCache ensures consistency between the buffer and DRAM by not buffering any dirty block. Dirty LLC evictions are simultaneously written back to the eviction-hidden buffer and memory. Given that the eviction-hidden buffer is neither strictly inclusive nor exclusive of the cache hierarchy, it is possible that the LLC eviction selected by the secure placement policy has already been buffered. In this case, clean evicted blocks need not be written to the buffer again. Dirty evicted blocks, however, need to overwrite the buffered copies to avoid data inconsistency. Such data inconsistency may occur to an evicted dirty block that is previously reloaded from the buffer to caches and then modified at least once before being evicted from the LLC. If we do not overwrite the buffered copy with the newly evicted dirty block, the latest value gets lost.

Second, while I/O devices initiate DRAM writing requests through direct memory access (DMA), TreasureCache synchronizes the written content to the corresponding block (if any) in the buffer. For instance, we assume that the to-be-written data block is stored in the buffer. At first, the CPU receives the writing request and begins to invalidate the corresponding block in caches. The data block in the buffer is not affected. Then, the system starts to write the memory. The data content in the buffer is updated along with the DRAM.

F. Constant-Time Flush Instruction

Finally, we integrate constant-time flush instructions into TreasureCache against Flush+Flush. It would be a luxury to simply enforce constant-time flush against flush-based attacks if our eviction-hidden buffer were not adopted. Existing countermeasures have to disable either flush instructions or shared memory (Section I-A), which are supposed to bring respective benefits to performance. TreasureCache can use the eviction-hidden buffer to prevent the Flush+Reload attack. The other type of the flush-based attack—Flush+Flush [2]—exploits the timing difference between flushing a cached block and flushing an uncached block (Section II-A). Flushing a cached block takes a longer time. It is effective to enforce a comparatively long time for flushing an uncached block against Flush+Flush [30].

A feasible constant-time flush design is not to abort flush instructions early in case of LLC misses [2]. Instead, we still forward the invalidation request to private caches after TreasureCache queries them for sharer information. Even if the queried cache does not hold the requested block to invalidate, it idles for a predefined time to imitate an invalidation operation. This prevents the attacker from inferring whether a flush request hits or misses in the LLC.

VI. Implementation

We implement TreasureCache using gem5 [56]. As TreasureCache introduces an eviction-hidden buffer to deal with LLC evictions from one or more cores, we expect that the implementation emulates accurate cache access behaviors. We thus adopt the gem5 Ruby build. Its built-in Ruby offers a detailed model of cache and memory hierarchies to emulate practical handling of memory accesses [57]. We implement the eviction-hidden buffer as an LLC extension. We modify the LLC controller to manage both the LLC and the eviction-hidden buffer.

Settings: We run TreasureCache on a 2.5 GHz out-of-order CPU. We concentrate more on multi-core settings because the side-channel attacks under concern aim to steal cross-core secrets. The CPU follows typical configuration settings in the literature [15], [16], [17], [19], [30], [55]. As shown in Table III, it adopts a three-layer cache hierarchy and MESI [52], [58] as the cache coherence protocol. We slightly modify the back-invalidation process to accommodate the implementation of sharer information acquisition (Section V-C1). In the baseline system, the private cache first searches the tag array upon receiving the back-invalidation command. The search result determines whether to evict the block indicated by that command. After the invalidation process completes, the private cache is supposed to respond to the LLC with `ACK` signals. In TreasureCache, we add a selective circuit triggered by different search results to respond LLC with either `ACK_EXIST` or

TABLE III
EXPERIMENT SETUP

Module	Configuration
Processor	1∼8 cores, 2.5 GHz out-of-order
Private L1 I/D cache	32 KB each, 4-way Latency: 1 cycle
Private L2 cache	256 KB, 8-way Latency: 5 cycles after L1
Shared L3 cache	2 MB per core, 16-way Latency: 10 cycles after L2
Eviction-hidden buffer	4∼32 KB per core Latency: 2 cycles `global_threshold`: 64∼512
Line/Block size	64 Bytes
Coherence protocol	MESI
Memory	Latency: 50 ns after eviction-hidden buffer

`ACK_NOTEXIST`. The selective circuit can be built with simple combinational logic, being easily incorporated into the original time slot and introducing no impact on the overall pipelines. Therefore, we expect that TreasureCache hitchhikes the back-invalidation process to acquire the sharer information without any additional delay. We will validate security and efficiency of TreasureCache with varying buffer settings in Sections VII and VIII-A, respectively.

Workloads: We practice TreasureCache with the latest SPEC CPU 2017 benchmarks [59]. For each benchmark, we execute the first billion of instructions for warming up the system. Then we use the second billion of instructions for collecting performance statistics. Besides measuring the performance of each individual benchmark, we also generalize workloads using a mixed set of benchmarks. Specifically, we randomly select n benchmarks as a mixed workload for an n-core CPU. We fully warm up the system until every benchmark in the mixed workload has executed at least one billion of instructions. Then we collect performance statistics until the slowest benchmark completes the second billion of instructions.

VII. SECURITY

In this section, we validate the security of TreasureCache. Our goal for cache protection is guaranteeing a low resolution of secret recovery that cannot lead to practical side-channel attacks. This requires TreasureCache to maintain any exploitable eviction in the eviction-hidden buffer for a sufficiently long time. The buffer structure should also deliver a comparative hit latency with that of the LLC. We sketch the timeline of buffered exploitable evictions against various exploitations as follows.

- *t0∼t1:* This is the time period where TreasureCache should accurately identify exploitable LLC evictions and place them in the buffer (Section VII-A).
- *t1∼t2:* TreasureCache should guarantee an indistinguishable latency for LLC hits and buffer hits (Section VII-B). This ultimately hides the access latency gap that otherwise exist between LLC accesses and memory accesses. After exploitable evicted blocks are buffered, TreasureCache should also prevent the attacker from maliciously replacing them out of the buffer (Section VII-C).
- *t2 onwards:* After a buffered exploitable block is eventually evicted, TreasureCache should guarantee that the so caused buffer misses bring forward no additional security issues than traditional LLC misses (Section VII-C).

A. Secure Placement of Exploitable Evictions

Our secure placement policy cannot be evaded by the attacker. As presented in Section V-C2, there are two categories of exploitable LLC evictions for TreasureCache to buffer. Each category calls for a respective placement policy.

The first policy requires that LLC evictions with associative copies in private caches should be buffered. To bypass this policy, the attacker needs to evict exploitable blocks from the victim's private caches before evicting them from the LLC. This contradicts with the settings of cross-core side-channel attacks of our interest, where back invalidation triggered by LLC evictions is a main source for the attacker to indirectly evict data from the victim's private caches. In Flush+Reload, Evict+Reload, and the Evict+Time attack, the attacker thus has no means to evict data from the victim's private caches first, and thus cannot evade the first placement policy.

The second policy requires that LLC evictions induced by reloaded blocks from the eviction-hidden buffer should also be buffered. It complements TreasureCache with the capability to buffer exploitable evictions that target the attacker's own data. Such exploitable data need be evicted by victim data. The attacker can deliberately make them locate in only the LLC rather than its own private caches (e.g., Prime+Reprime+Probe [31], [33]). This helps the attacker's data to evade the first placement policy. To further evade the second policy, the attacker needs to make sure that TreasureCache does not buffer the victim data it evicts using an eviction set during the prime or reprime phase. However, the attacker cannot manipulate the victim's data to this end because of the first placement policy.

B. Indistinguishable Buffer-Hit and LLC-Hit

We use measurements to demonstrate that loading blocks from the eviction-hidden buffer shows an indistinguishable latency as an LLC hit. Specifically, we build a multi-core micro benchmark that randomly issues load and store operations. We then run it on TreasureCache and sample 1,000 access latencies for either of LLC hits and buffer hits in comparison with that of buffer misses. As shown in Fig. 4, access lantencies of both LLC hits and buffer hits are centralized around 24 cycles. This closes the door for a timing difference that the attacker craves for when reloading exploitable evicted blocks from the buffer.

C. Unexploitable Secret-Recovery Resolution

Finally, we show that TreasureCache can buffer exploitable evicted blocks for a sufficiently long time and leave the attacker with an unexploitable secret-recovery resolution. In TreasureCache, the attacker might observe the victim's memory access if buffered exploitable blocks are evicted out of the eviction-hidden buffer within one attacking epoch. The corresponding probability mainly depends on the rate of owning new blocks

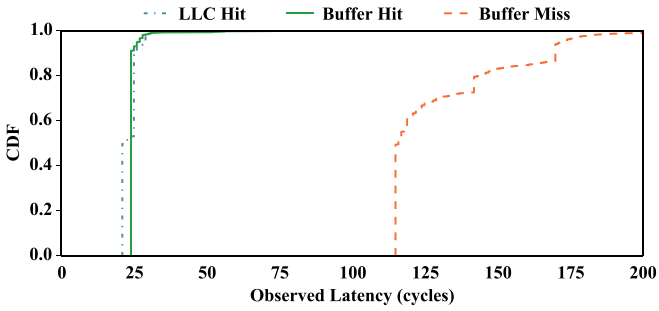

Fig. 4. Comparison of access latency over LLC and eviction-hidden buffer.

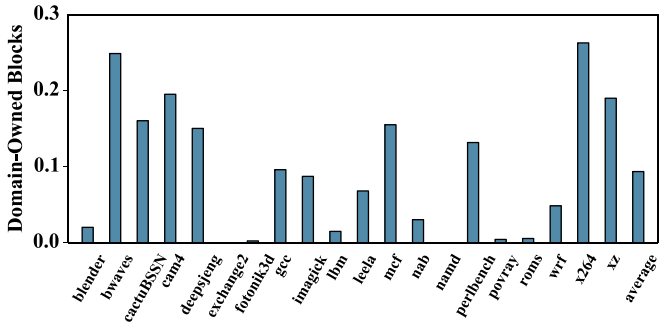

Fig. 5. Number of additional buffered LLC evictions owned by each security domain per 10,000 cycles. The results are derived by running SPEC 2017 benchmarks on a two-core CPU.

for the victim's security domain. This is because only the victim per se can revoke its ownership of buffered exploitable blocks. To make this probability above 0.5 (greater than probability of random guessing), the attacker is required to prolong the attacking epoch. However, this reduces the attack resolution and the attacker fails to recover secrets.

We investigate the number of newly owned blocks for evicting a buffered exploitable block. This highly depends on the value g of `global_threshold` that upper bounds the number of owned buffered blocks per security domain. Upon owning a new block, the victim's security domain is required to already own a number `global_threshold` of buffered blocks to invalidate its ownership of one of its owned blocks by `ownership_invalidation` (Section IV-C3). The best case for the attacker is that buffering this new block can also make the de-owned buffered block be evicted out of the buffer through `secure_eviction`. In practice, a de-owned buffered block can survive at least for a certain amount of time. We conservatively use the attacker's most favorable case to lower bound its attack cost. Given random invalidation, the buffered exploitable block can be de-owned with probability of $\frac{1}{g}$. Every run of `ownership_invalidation` triggered by a newly buffered block is independent. Let N denote the number of newly owned blocks for eventually replacing the buffered exploitable block. It follows the geometric distribution with the success probability for each trial as $\frac{1}{g}$:

$$Pr(N = k) = \frac{1}{g} \times \left(1 - \frac{1}{g}\right)^{k-1}. \quad (1)$$

The cumulative distribution function of N is as follows.

$$F_N(k) = 1 - \left(1 - \frac{1}{g}\right)^{k}. \quad (2)$$

Consider `global_threshold` is set as 128 for example, where 128 is a well accepted security parameter for limiting space per security domain using cache partitioning [55]. In TreasureCache, the true positive rate (TPR) for the attacker can be formulated by the cumulative distribution function of N. To make it above 0.5, k should be greater than 89. Fig. 5 reports the rate of owning new blocks for each core in SPEC CPU 2017 benchmarks. The results show that each core owns only about 0.09 block on average to the buffer every 10,000 cycles. Taking both of the preceding factors into account, we can estimate that the attacker needs an attacking epoch more than $\frac{89}{0.09} \times 10{,}000 = 9{,}888{,}888 = 9.89 \times 10^6$ cycles. According to SHARP [15], in a successful cache side-channel attack the attacking epoch should be set within 2,500-10,000 cycles. A larger attacking epoch would make the secret-recovery resolution unexploitable [3]. Given that TreasureCache has reduced the attack resolution by 1,000 times, the attacker can hardly leak any useful information.

VIII. Performance

Metrics: In this section, we evaluate TreasureCache performance using two typical metrics—instructions per cycle (IPC) and misses per 1,000 instructions (MPKI). The baseline cache measures MPKI using LLC misses while TreasureCache uses buffer misses instead because the small eviction-hidden buffer in TreasureCache delivers a comparative hit latency as LLC hits (Fig. 4). For performance comparison, both metrics are normalized over that of the baseline cache.

Results: It has been a luxury to outperform the baseline cache hierarchy while securing it against both flush-based and conflict-based side-channel attacks. To the best of our knowledge, TreasureCache is the first to break this barrier. TreasureCache can improve the IPC of baseline by 0.03%∼0.28% with 1∼8 cores on a 2.5 GHz processor. Such a performance gain requires only a small buffer of 8∼64 KB, bringing an affordable storage overhead less than 0.5% of the LLC capacity.

A. Overall Performance

We start with evaluating overall performance of TreasureCache on single-core and multi-core systems. Without loss of generality, we instantiate a multi-core system with a two-core processor. Performance scalability with more cores will be evaluated in Section VIII-C. As aforementioned in Section VI, we run mixed benchmarks for multi-core evaluation. A set of mixed benchmarks for a two-core processor consists of two randomly chosen SPEC benchmarks. Each is pinned to one of the two cores for execution and statistics collection. Given `global_threshold` set as 128 and block size set as 64 bytes, we configure the eviction-hidden buffer with size of $8 \times$ `core-count` KB. We calculate the normalized metrics for each core and average them over the core count.

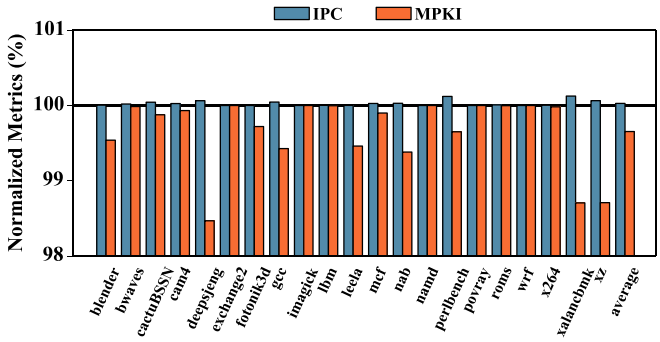

Fig. 6. TreasureCache performance on a single-core system.

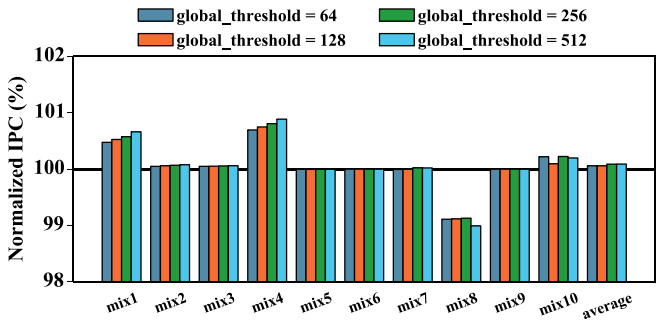

Fig. 7. Normalized IPC of TreasureCache with varying global_threshold.

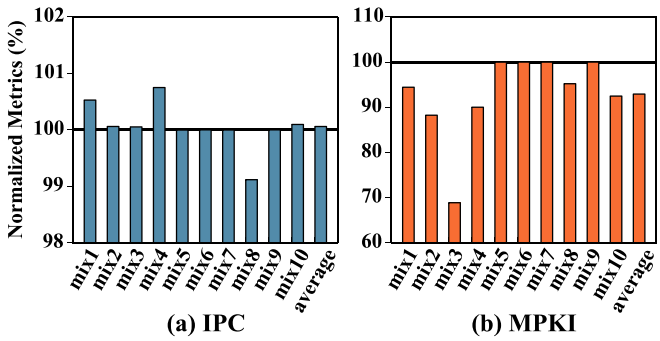

Fig. 8. TreasureCache performance on a two-core system.

Figs. 6 and 8 report the normalized performance metrics of TreasureCache over that of baseline for single-core and two-core evaluation, respectively. On the single-core system, TreasureCache outperforms the baseline in terms of a 0.03% higher IPC and a 0.35% lower MPKI on average. On the two-core system, TreasureCache shows a much higher speedup. It outperforms the baseline in terms of a 0.06% higher IPC and a 7.08% lower MPKI on average. TreasureCache offers such performance gains because it turns many memory accesses due to LLC misses into buffer hits, which are fast as LLC hits.

B. Buffer Capacity

We continue with evaluating how buffer capacity impacts TreasureCache performance. More specifically, we tune buffer capacity with different `global_threshold` settings. As

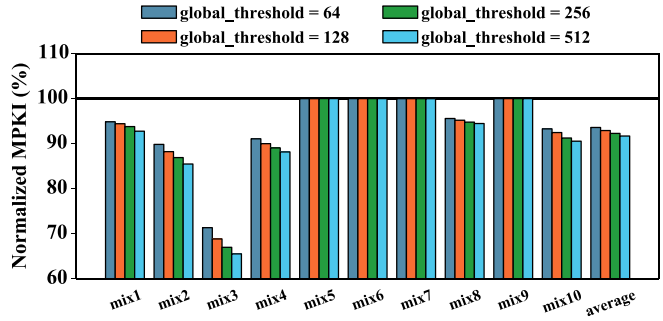

Fig. 9. Normalized MPKI of TreasureCache with varying global_threshold.

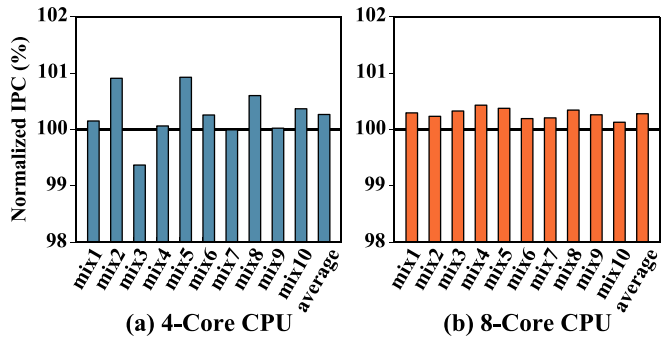

Fig. 10. Normalized IPC of TreasureCache with increasing core count.

described in the Section IV-C3, the eviction-hidden buffer size is greater than or equal to `global_threshold` × `core-count`. Given a fixed core count, the buffer size increases with the value of `global_threshold`. On the two-core system with 64-byte blocks, we evaluate TreasureCache with `global_threshold` of 64, 128, 256 and 512 blocks and buffer size of 8 KB, 16 KB, 32 KB, and 64 KB, respectively.

Figs. 7 and 9 compare performance of TreasureCache with that of baseline as `global_threshold` varies. We expect a higher IPC and a lower MPKI upon a larger `global_threshold`. Larger `global_threshold`s bring a larger eviction-hidden buffer, which can buffer more blocks and prevent their subsequent reuse from much slower memory accesses. In addition, our indexed fully-associative architecture speeds up the buffer access. This helps to reduce the impact of buffer size on lookup latency. Figs. 7 and 9 conform to the expected trend. TreasureCache increases the IPC of baseline by 0.06%∼0.09% and reduces the MPKI of baseline by 6.40%∼8.29% with `global_threshold` ranging from 64 to 512. This encourages adopters of TreasureCache to strive for more performance gains using a large buffer, as long as they consider the corresponding storage cost affordable.

C. Processor Capacity

We now evaluate the scalability of TreasureCache given more cores. Specifically, we run mixed benchmarks on four-core and eight-core systems. Fig. 10 reports the corresponding IPC of TreasureCache normalized over that of baseline. TreasureCache averagely increases the IPC of baseline by 0.27% and 0.28%

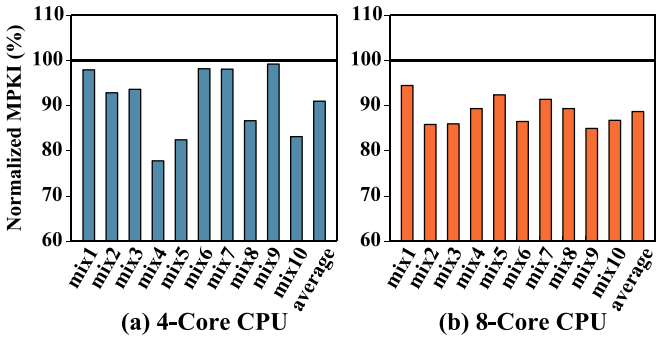

Fig. 11. Normalized MPKI of TreasureCache with increasing core count.

TABLE IV
STORAGE OVERHEAD

Component	Storage Overhead (bit)
Data Store	$c \times g \times 64 \times 8$
Mappings	$c \times g \times (\log_2 g + m)$
`security_domain_list`	$c \times g \times \log_2 c$
`counter`	$c \times \log_2 g$
`global_threshold`	$\log_2 g$
Overall: $c \times g \times (\log_2 (c \times g) + m + 512) + (c + 1) \times \log_2 g$	

on the four-core system and the eight-core system, respectively. Furthermore, as shown in Fig. 11, TreasureCache reduces the MPKI of baseline by 8.98% and 11.27% given four cores and eight cores, respectively.

In summary, TreasureCache offers more performance gains as the system scales. We thus expect TreasureCache to be well adaptive to large-scale systems.

D. Storage Overhead

Finally, we investigate the storage overhead of TreasureCache. It introduces an extra eviction-hidden buffer beyond the cache hierarchy. The buffer stores exploitable evicted data, mappings of their physical addresses to buffer indices (Section V-D), and secure states such as per–buffered-block `security_domain_list`, per-core `counter`, and a single `global_threshold`. Given a system with a number c of CPU cores, m-bit physical addresses, 64-byte blocks, and `global_threshold` set as g, we can approximate the storage overhead of different types of information in the buffer as in Table IV. The overall storage cost `buffer` is as follows:

$$c \times g \times (\log_2 (c \times g) + m + 512) + (c + 1) \times \log_2 g. \quad (3)$$

Following a conventional LLC configuration with 2 MB per core (Table III) [15], [16], [17], [30], [55], we can normalize the buffer's storage overhead over LLC storage space.

$$\frac{\texttt{buffer}}{c \times 2 \times 10^6 \times 8} \approx \frac{g \times (\log_2 (c \times g) + m + 512)}{1.6 \times 10^7}. \quad (4)$$

It turns out to be relatively insensitive to core count. This further validates scalability of TreasureCache.

To put the storage overhead in a real context, we consider modern processors with 64-bit physical addresses (i.e., $m = 64$). We consider a sufficiently large `global_threshold` of $g = 128$ for guaranteeing security (Section VII-C). Under such practical settings, TreasureCache requires only a $c \times (9 + \frac{\log_2 c}{64})$ KB buffer space by (3) and introduces an only 0.45% storage overhead by (4).

IX. RELATED WORK

In this section, we review related work that applies a buffer inside the cache hierarchy and is thus considered relevant to TreasureCache. Such a structural design strategy has been exploited to improve cache performance [60], secure cache directory [61], and defend against transient execution attacks [62], [63], [64]. Next, we will show how our TreasureCache differs from these solutions.

Jouppi [60] has proposed to improve direct-mapped caches with a small fully-associative victim cache. This cache temporally stores evictions from the direct-mapped cache. When the CPU reaccesses these evicted blocks, they are loaded from the victim cache to the direct-mapped cache again. The victim cache in [60] and the eviction-hidden buffer in our TreasureCahce can both preserve the blocks evicted from the higher level cache. However, the traditional victim cache is not designed for security. It thus cannot effectively hide the exploitable blocks and an intelligent attacker can easily circumvent its protection. For a shared victim cache, the attacker can craft any number of conflict blocks at any time into it to evict the exploitable blocks, as long as the attacker can manipulate enough CPU cores. This makes the traditional victim cache expose to side-channel attacks. As a main contribution of our paper, TreasureCache develops placement and replacement policies to filter exploitable blocks and prevent the attacker to interfere with the exploitable blocks.

SecDir [61] initiates protection of non-inclusive caches against side-channel attacks [65]. Such side-channel attacks targeting non-inclusive caches exploit the inclusive directory structure to leak secret [65]. For a non-inclusive LLC, there exists a structure called Extended Directory (ED) around the LLC to maintain the coherence states of cache lines in L2 cache but not in the LLC. The attacker can maliciously induce conflicts in ED to transfer exploitable blocks from the L2 cache to the LLC. The timing difference between the L2 cache and LLC/DRAM is used to infere the secret [65]. SecDir [61] complements ED with Victim Directory (VD) to defeat directory side-channel attacks. To prevent the attacker from creating inclusion victim through the eviction of the directory entries, VD is used to buffer the conflicting TD entry. However, the sophisticated attacker may also evict the buffered TD entry. SecDir further assigns each core a VD and enforces hard isolation between different cores. This prohibits the malicious from evicting the directory entry of the victim core. Unlike SecDir, TreasureCache does not enforce hard isolation for the eviction-hidden buffer. The exploitable blocks evicted from the LLC can still be shared by multiple cores. Not enforcing hard isolation serves for both efficiency and security. If we enforce hard isolation to the eviction-hidden buffer and make the victim core own the exploitable block, the timing difference may be still detectable during attacker's reload operation. On the contrary, if the attacker core owns the exploitable block, it

can easily evict the exploitable block from the eviction-hidden buffer. As a result, TreasureCache uses the secure replacement policy to dynamically partition the buffer. TreasureCache allows each core having access to the blocks which are resident in the eviction-hidden buffer while preventing the malicious core from unilaterally evicting the buffered exploitable blocks.

TreasureCache can be slightly adapted to protect non-inclusive caches against side-channel attacks. Similar to the case for protecting inclusive caches, each L2 cache can associate with an eviction-hidden buffer. This buffer is used to preserve L2 evictions due to ED conflicts. Once an ED conflict occurs, information of the sharer and the evictor can be acquired from the ED and transited along with the back-invalidation command (i.e., invalidate the conflicted exploitable block in the L2 cache). TreasureCache can uses the received information to decide whether to store the exploitable block in the eviction-hidden buffer and tag it with the victim's ownership. Upon being reloaded, the data block will be served from the eviction-hidden buffer. This prevents potential timing channels. Moreover, the suggested adaptation is different from SecDir because TreasureCache dose not alter the directory structure and still allows transferring exploitable blocks. It simply accelerates the access of exploitable blocks to confuse the attacker.

A series of defenses against cache-based transient execution attacks [62], [63], [64] also use a buffer to avoid transient instructions exploiting cache states. For example, InvisiSpec [62] and MuonTrap [63] use a Speculative Buffer (SB) or an L0 cache to hide data blocks loaded by unsafe instructions. Only when they are deemed as correct path instructions can the associated data blocks be placed into the original cache hierarchy. Different from the eviction-hidden buffer, SB and L0 cache are not used to store the cache evictions. They thus are orthogonal with the eviction-hidden buffer. Apart from that, Revice [64] employs a victim cache to preserve data blocks replaced by the transient cache lines. When restoring the cache states changed by the transient instructions, Revice can quickly retrieve data blocks in the victim cache. Although the additional buffer in [62], [63], [64] can be used to hide exploitable blocks, it still follows the aforementioned intrinsic limitation of the traditional victim cache. That is, they are not effective for defeating eviction-based cache side-channel attacks and thus cannot supersede the eviction-hidden buffer.

X. Conclusion

We have studied the idea of buffering exploitable LLC evictions to prevent cache side-channel attacks without sacrificing performance. It has been a luxury to outperform unprotected caches while securing them against both flush-based and conflict-based attacks. The root cause lies in their inevitable hardware and software modifications for preventing cache evictions. Using our specialized small eviction-hidden buffer, we can afford to allow cache evictions yet hide exploitable evictions in the buffer. Buffer hits are comparatively fast as LLC hits. This closes the timing channel that otherwise exists between LLC hits and misses. We implement the eviction-buffering idea through TreasureCache. Extensive analytical and experiment results show that TreasureCache not only guarantees security but also improves performance.

Acknowledgment

We would like to sincerely thank the Editors and Reviewers of IEEE Transactions on Dependable and Secure Computing for your review efforts and helpful feedback. We also wish you health and safety during the pandemic.

References

[1] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 605–622.
[2] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush flush: A fast and stealthy cache attack," in *Proc. Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, Springer, 2016, pp. 279–299.
[3] Y. Yarom and K. Falkner, "FLUSH RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Conf. Secur. Symp.*, 2014, pp. 719–732.
[4] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. USENIX Secur. Symp.*, 2015, pp. 897–912.
[5] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 987–1002.
[6] S. B. Dutta, H. Naghibijouybari, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Leaky buddies: Cross-component covert channels on integrated CPU-GPU systems," in *Proc. 48th Annu. Int. Symp. Comput. Architecture*, 2021, pp. 972–984.
[7] A. Agarwal et al., "Spook.js: Attacking chrome strict site isolation via speculative execution," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 699–715.
[8] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "SpecHammer: Combining spectre and rowhammer for new speculative attacks," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 681–698.
[9] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2019, pp. 142–157.
[10] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *Proc. 12th USENIX Conf. Offensive Technol.*, 2018, Art. no. 3.
[11] M. Lipp et al., "Meltdown: Reading kernel memory from user space," in *Proc. USENIX Secur. Symp.*, 2018, pp. 973–990.
[12] J. Cook, J. Drean, J. Behrens, and M. Yan, "There's always a bigger fish: A clarifying analysis of a machine-learning-assisted side-channel attack," in *Proc. 49th Annu. Int. Symp. Comput. Architecture*, 2022, pp. 204–217.
[13] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: Attacking ARM pointer authentication with speculative execution," in *Proc. 49th Annu. Int. Symp. Comput. Architecture*, 2022, pp. 685–698.
[14] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proc. IEEE/ACM 51st Annu. Int. Symp. Microarchitecture*, 2018, pp. 775–787.
[15] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks," in *Proc. 44th Annu. Int. Symp. Comput. Architecture*, 2017, pp. 347–360.
[16] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "PhantomCache: Obfuscating cache conflicts with localized randomization," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–17.
[17] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *Proc. USENIX Secur. Symp.*, 2021, pp. 1379–1396.
[18] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 871–882.
[19] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "SCATTERCACHE: Thwarting cache attacks via cache set randomization," in *Proc. USENIX Secur. Symp.*, 2019, pp. 675–692.
[20] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! But we can fix it," in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 955–969.

[21] D. Ojha and S. Dwarkadas, "TimeCache: Using time to eliminate cache side channels when sharing software," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Architecture*, 2021, pp. 375–387.
[22] J. Van Bulck et al., "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *Proc. USENIX Secur. Symp.*, 2018, pp. 991–1008.
[23] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 39–54.
[24] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proc. IEEE/ACM 51st Annu. Int. Symp. Microarchitecture*, 2018, pp. 974–987.
[25] F. Liu et al., "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2016, pp. 406–418.
[26] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: Secure dynamic cache partitioning for efficient timing channel protection," in *Proc. ACM/EDAC/IEEE 53rd Des. Autom. Conf.*, 2016, pp. 1–6.
[27] X. Dong, Z. Shen, J. Criswell, A. L. Cox, and S. Dwarkadas, "Shielding software from privileged side-channel attacks," in *Proc. USENIX Secur. Symp.*, 2018, pp. 1441–1458.
[28] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proc. ACM/IEEE 46th Annu. Int. Symp. Comput. Architecture*, 2019, pp. 360–371.
[29] M. Kayaalp et al., "RIC: Relaxed inclusion caches for mitigating LLC side-channel attacks," in *Proc. ACM/EDAC/IEEE 54th Des. Autom. Conf.*, 2017, pp. 1–6.
[30] B. Panda, "Fooling the sense of cross-core last-level cache eviction based attacker by prefetching common sense," in *Proc. 28th Int. Conf. Parallel Architectures Compilation Techn.*, 2019, pp. 138–150.
[31] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$ A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 591–604.
[32] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Proc. Cryptographers' Track RSA Conf.*, 2006, pp. 1–20.
[33] D. Kumar, C. S. Yashavant, B. Panda, and V. Gupta, "How sharp is SHARP?," in *Proc. 13th USENIX Conf. Offensive Technol.*, 2019, Art. no. 4.
[34] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2018, pp. 168–179.
[35] O. Aciiçmez, "Yet another microarchitectural attack: Exploiting i-cache," in *Proc. ACM Workshop Comput. Secur. Architecture*, 2007, pp. 11–18.
[36] O. Acıiçmez and W. Schindler, "A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL," in *Proc. Cryptographers' Track RSA Conf.*, 2008, pp. 256–273.
[37] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *Proc. 15th Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2009, pp. 667–684.
[38] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on AES to practice," in *Proc. IEEE Symp. Secur. Privacy*, 2011, pp. 490–505.
[39] 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, Sep./Oct. 2016.
[40] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *Proc. IEEE/ACM 41st Int. Symp. Microarchitecture*, 2008, pp. 83–93.
[41] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proc. 34th Annu. Int. Symp. Comput. Architecture*, 2007, pp. 494–505.
[42] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD REFRESH: Abusing cache replacement policies to perform stealthy cache attacks," in *Proc. USENIX Secur. Symp.*, 2020, Art. no. 111.
[43] W. Xiong and J. Szefer, "Leaking information through cache LRU states," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2020, pp. 139–152.
[44] W. Xiong, S. Katzenbeisser, and J. Szefer, "Leaking information through cache LRU states in commercial processors and secure caches," *IEEE Trans. Comput.*, vol. 70, no. 4, pp. 511–523, Apr. 2021.
[45] Y. Cui, C. Yang, and X. Cheng, "Abusing cache line dirty states to leak information in commercial processors," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2022, pp. 82–97.
[46] F. Yao, M. Doroslovački, and G. Venkataramani, "Covert timing channels exploiting cache coherence hardware: Characterization and defense," *Int. J. Parallel Program.*, vol. 47, no. 4, pp. 595–620, 2019.
[47] C. Miao, K. Bu, M. Li, S. Mao, and J. Jia, "SwiftDir: Secure cache coherence without overprotection," in *Proc. IEEE/ACM 55th Int. Symp. Microarchitecture*, 2022, pp. 662–677.
[48] K. Loughlin, S. Saroiu, A. Wolman, Y. A. Manerkar, and B. Kasikci, "MOESI-prime: Preventing coherence-induced hammering in commodity workloads," in *Proc. 49th Annu. Int. Symp. Comput. Architecture*, 2022, pp. 670–684.
[49] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," *ACM SIGARCH Comput. Archit. News*, vol. 18, pp. 148–159, 1990.
[50] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *Proc. 18th Int. Conf. Parallel Architectures Compilation Techn.*, 2009, pp. 261–270.
[51] R. Singhal, "Inside intel core microarchitecture (nehalem)," in *Proc. IEEE Hot Chips Symp.*, 2008, pp. 1–25.
[52] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synth. Lectures Comput. Archit.*, vol. 15, no. 1, pp. 1–294, 2020.
[53] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr, and J. Emer, "Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies," in *Proc. IEEE/ACM 43rd Annu. Int. Symp. Microarchitecture*, 2010, pp. 151–162.
[54] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE J. Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006.
[55] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid side-channel-resilient caches for trusted execution environments," in *Proc. USENIX Secur. Symp.*, 2020, pp. 451–468.
[56] N. Binkert et al., "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
[57] Ruby home page, 2024. [Online]. Available: https://www.gem5.org/documentation/general_docs/ruby
[58] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proc. 11th Annu. Int. Symp. Comput. Architecture*, 1984, pp. 348–354.
[59] J. Bucek, K.-D. Lange, and J. V. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 41–42.
[60] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. 17th Annu. Int. Symp. Comput. Architecture*, 1990, pp. 364–373.
[61] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, "SecDir: A secure directory to defeat directory side-channel attacks," in *Proc. ACM/IEEE 46th Annu. Int. Symp. Comput. Architecture*, 2019, pp. 332–345.
[62] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *Proc. IEEE/ACM 51st Annu. Int. Symp. Microarchitecture*, 2018, pp. 428–441.
[63] S. Ainsworth and T. M. Jones, "MuonTrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *Proc. 47th Annu. Int. Symp. Comput. Architecture*, 2020, pp. 132–144.
[64] S. Kim et al., "ReViCe: Reusing victim cache to prevent speculative cache leakage," in *Proc. IEEE Secure Develop.*, 2020, pp. 96–107.
[65] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 888–904.

Mengming Li received the master's degree from the School of Software Technology, Zhejiang University, Hangzhou, China. His research interest includes computer architecture.

Kai Bu (Member, IEEE) received the BSc and MSc degrees in computer science from the Nanjing University of Posts and Telecommunications, Nanjing, China, in 2006 and 2009, respectively, and the PhD degree in computer science from The Hong Kong Polytechnic University, Hong Kong, in 2013. He is currently an associate professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. His research interests include network security and computer architecture. He is a member of the ACM, and the CCF. He is a recipient of the Best Paper Award of IEEE/IFIP EUC 2011 and the Best Paper Nominee of IEEE ICDCS 2016.

Chenlu Miao is currently working toward the master's degree with the School of Software Technology, Zhejiang University, Hangzhou, China. Her research interest includes computer architecture.

Kui Ren (Fellow, IEEE) is professor and associate dean of the College of Computer Science and Technology, Zhejiang University, where he also directs the Institute of Cyber Science and Technology. Before that, he was SUNY Empire Innovation professor with the State University of New York at Buffalo. His research interests include data security, IoT security, AI security, and privacy. He received many recognitions including Guohua Distinguished Scholar Award of ZJU, IEEE CISTC Technical Recognition Award, SUNY Chancellor's Research Excellence Award, Sigma Xi Research Excellence Award, NSF CAREER Award, etc. He has published extensively in peer-reviewed journals and conferences and received the Test-of-time Paper Award from IEEE INFOCOM and many Best Paper Awards from IEEE and ACM, including ACM MobiSys, IEEE ICDCS, IEEE ICNP, IEEE Globecom, ACM/IEEE IWQoS, etc. He is a fellow of ACM. And he serves on the editorial boards of many IEEE and ACM journals. He also serves as chair of SIGSAC of ACM China Council, a member of ACM ASIACCS steering committee, and a member of S&T Committee of Ministry of Education of China.