CacheGuardian: A Timing Side-Channel Resilient LLC Design

Ziang Zhou[1,2], Qi Zhu[1,2], Hao Lan[1,2], Huifeng Zhu[3], Wei Yan[1,2,*], Chenglu Jin[4], Xuejun An[1,2], Xiaochun Ye[1,2]
[1] SKLP, Institute of Computing Technology, CAS, Beijing, China
[2] School of Computer Science and Technology, UCAS, Beijing, China
[3] Washington University in St. Louis [4] Centrum Wiskunde & Informatica, Amsterdam
{zhouziang23s, zhuqi23s, lanhao20s}@ict.ac.cn, zhuhuifeng@wustl.edu, yanwei@ict.ac.cn,
chenglu.jin@cwi.nl, {axj, yexiaochun}@ict.ac.cn

Abstract—**In cloud computing environments, the last-level cache (LLC) shared by multiple tenants is frequently exploited through timing side-channel attacks, enabling unauthorized data leakage. To address this issue, various defense mechanisms have been proposed. However, existing works exhibit deficiencies in terms of performance overhead, coverage of attacks, and detection accuracy. In response to these challenges, we propose CacheGuardian, a hardware-based LLC protection design which aims to provide stronger, broader, and more accurate protection against timing side-channel attacks with low performance overhead. It includes: (1) A *behavior-based, generic attack detector* capable of identifying multiple timing side-channel attacks in real time; (2) A *cache-set-level access control mechanism* that strictly restricts cache usage exclusively for the identified attackers instead of influencing all security domains.**

We implement our design in a gem5 simulator to evaluate both its security and performance. Our proof-of-concept attacks and SPEC 2017 benchmarks show that our design is effective against a wide range of timing side-channel attacks, reducing attack success rates by up to 256×, including camouflaged variants. Moreover, it improves the performance of benign workloads by an average of 2.26% with only 2.4% storage overhead.

I. Introduction

In cloud deployments, the shared last-level cache, designed to improve hardware utilization and reduce memory access latency, exposes a critical attack surface. Adversaries can exploit LLC sharing to mount timing side-channel attacks, inferring sensitive data (e.g., cryptographic keys or user information) by observing subtle variations in cache access patterns [1]–[9]. Despite extensive research on defenses against timing side-channel attacks, existing solutions face severe limitations in practice.

Limitation 1: Difficult Balance between Security and Performance: Numerous hardware designs have been proposed to protect the shared LLC against timing side-channel attacks. These countermeasures can be broadly categorized into two classes:

- **Physical resource isolation**: Techniques such as cache state flushing [10], cache partitioning [11]–[13], and other structural restrictions enforce strong isolation between security domains.
- **Obfuscation and noise injection**: These methods inject randomness or noise to obscure access patterns, including prefetcher-driven noise injection [14]–[16], randomized cache indexing [17]–[20], and the use of extra cache module [21], [22].

*Wei Yan is the corresponding author.

TABLE I
Coverage of Existing Hardware Defenses Against Last-Level Cache Timing Side-Channel Attacks

Existing Work	Reuse-based	Conflict-based	Coherence-based
HYBCACHE [13]	●	●	○
CEASER_S [19]	○	●	○
Randomization [20]	○	●	○
PREFENDER* [14]	◐	◐	○
SwiftDir [25]	○	○	●
TimeCache [26]	●	○	●
TreasureCache* [21]	●	◐	○
CacheGuardian	●	●	●

* PREFENDER [14] does not mitigate covert-channel attacks targeting individual cache lines. TreasureCache [21] is ineffective against conflict-based attacks on non-inclusive LLC [4].

However, both approaches have serious drawbacks: obfuscation-based defenses can only raise the bar for attackers but cannot fully eliminate leakage [23], [24], while physical resource isolation—despite its strong security guarantees—imposes significant performance penalties, with DAWG [12] demonstrating worst-case performance overheads reaching 12–15% and InvisiSpec [22] reaching approximately 21%

Limitation 2: Limited Coverage Across Attack Variants: Existing hardware defenses only address specific classes of LLC side-channel attacks. As shown in Table I, there is no broadly applicable protection strategy against all attacks.

Limitation 3: Inadequate Detection Depth of Camouflaged and Low-Frequency Attacks: Many existing side-channel detection schemes rely on CPU hardware performance counters (HPCs) as their primary indicator [27]. However, attacks that trigger infrequently [28] or are camouflaged within complex benign workloads [29] cause minimal perturbation to overall system performance. As a result, these attacks do not produce significant statistical deviations in HPC measurements, and thus evade detection by current HPC-based defenses.

To address these limitations, we propose **CacheGuardian**, a last-level cache protection design with two key innovations: (1) A *behavior-driven detection mechanism* capable of identifying diverse timing side-channel attacks (including camouflaged and low-frequency variants) in real time, and (2) A *cache-set-level access control* that enforces fine-grained restrictions *only* on malicious actors while preserving legitimate cache access patterns.

We summarize the key contributions of this work as follows:

- We introduce CacheGuardian, a fine-grained dynamic hardware protection mechanism for the shared LLC that integrates a behavior-based generic attack detector and selective cache-set-level access control to break the attack chain of timing side-channel exploits.
- We prove that CacheGuardian provides comprehensive protection against a wide range of LLC timing side-channel attacks, including reuse-based, conflict-based, coherence-state-based channels, and low-frequency camouflaged Spectre variants. It achieves up to **256×** reduction in attack success rates.
- Through gem5 and SPEC CPU2017 experiments, we show that CacheGuardian incurs negligible overhead, even improving benign workload performance by **2.26%** on average in multi-core attack coexistence scenarios, while requiring less than **2.4%** additional LLC SRAMs.

II. Background

A. Timing Side-Channel Attacks

Timing side-channel attacks [23] targeting last-level caches in multi-core systems can be broadly categorized into two groups: *conventional cache attacks* and *transient execution attacks*. Conventional cache attacks include both *side-channel* and *covert-channel* attacks. In covert channels, two colluding processes or virtual machines exploit shared LLC resources to communicate secretly, bypassing system isolation mechanisms. Side-channel attacks can be further divided into three types:

- **Reuse-based attacks:** The attacker shares memory with the victim and flushes target cache blocks using `clflush` or eviction sets. After the victim potentially reloads the data, the attacker accesses it again. A low-latency access (cache hit) indicates that the victim accessed the data. Examples include *Flush+Reload*, *Evict+Reload*, and *Flush+Flush*.
- **Conflict-based attacks:** No shared memory is required. The attacker fills specific cache sets using an eviction set, waits for the victim's access to evict cache lines, and then measures the latency of re-accessing the data. A higher latency suggests eviction and allows the attacker to infer the victim's behavior. Examples include *Prime+Probe* and *Evict+Time*.
- **Coherence-state-based attacks:** These attacks exploit timing differences caused by cache coherence protocols such as MESI [30]. An attacker can infer or transmit information based on the latency differences when accessing data in different coherence states (e.g., Exclusive vs. Shared). This method is effective in multi-core environments and often used for covert communication.

In **transient execution attacks** [31], [32], adversaries exploit speculative execution in modern processors. Although transient instructions are eventually squashed, they may access sensitive data and leave microarchitectural traces. To extract this leaked data, the attacker must encode it through cache-based channels before the speculative window closes, using the aforementioned conventional techniques.

B. CPU Microarchitecture

Cache Hierarchy and Memory Operations: Modern processors employ multi-level cache hierarchies to reduce memory access latency. These cache levels typically consist of an L1 cache with low latency, an L2 cache with larger capacity, and a shared LLC. The cache system improves performance by taking advantage of spatial and temporal locality. Cache coherence across multiple cores is typically maintained by hardware protocols. However, in certain scenarios software may explicitly manage cache contents. The clflush instruction allows a program to evict a specific cache line, ensuring that subsequent accesses reflect the latest value in memory.

MSHRs: The cache controller maintains a set of Miss Status Handling Registers (MSHRs) to support non-blocking memory access. Each MSHR entry records the status of a pending cache miss. It stores information including the target memory address, the type of access request, and where the returned data should be placed. When multiple requests target the same cache line, the MSHR allows them to be merged. Different cache levels manage independent MSHRs, but cascading misses may indirectly compete for resources.

Control Register 3 and Context Switching: Modern operating systems frequently perform context switches between processes to enable multitasking. The Control Register 3 (CR3) holds the physical base address of the current process page table and defines its virtual address space. The operating system updates the value of CR3 during a context switch to activate the next process address space, effectively marking the transition between processes.

III. Threat Model

In this work, we assume the attacker and victim execute on different processor cores while sharing the last-level cache. The attacker may exploit reuse-based [1], conflict-based [2]–[4], and coherence-state-based [30] cache side-channel techniques, as well as transient execution attacks [32] that transmit information via the LLC, to steal the victim's private data.

We do not consider side channels built on other microarchitectural structures such as TLBs [33] or Line Fill Buffers [34].

IV. CacheGuardian

Existing defenses, as discussed in Section I, suffer from several key limitations, including the tradeoff between performance and security, limited coverage across attack types, and

TABLE II
GLOSSARY OF KEY TERMS AND PARAMETERS

Term / Symbol	Definition
CacheGuardian	Dynamic, fine-grained hardware protection framework
Guardian Detector	Behavior-based attack detector module
Guardian Buffer	Per-core buffer storing attack-phase metadata
`CPUID`	Core identifier metadata in each cache line
`ADDR_IDX`	Cache-set index extracted from physical address
`validPrepare`	Entry flag indicating completion of the Prepare phase
`validVictim`	Entry flag indicating completion of the VictimAccess phase
`accumulator`	Counter of completed attack rounds per entry
`attackType`	Metadata field recording the type of attack (e.g., INVALID or CONFLICT)
Blacklist	Core-ID list of detected attackers
Sensitive Flag	Bit-vector marking sensitive cache sets
`ATTACK_WINDOW`	Temporal detection window size (e.g., 1 ms)
N_{core}	Number of processor cores in the system
N_{cacheset}	Number of cache sets in the LLC

low detection accuracy. Consequently, our work aims to meet the following objectives:

- **Hardware-enforced physical resource isolation** rather than reliance on cache randomization or noise injection alone.
- **Minimized performance overhead** introduced by strict cache isolation.
- **Comprehensive protection** against a broad spectrum of LLC-based timing side-channel attacks.
- **Robust defense** against low-frequency or camouflaged attacks embedded within complex applications.

To meet these goals, we propose CacheGuardian, a dynamic, fine-grained hardware-based protection mechanism that consists of two key components:

1) **Guardian Detector**: a behavior-based and generic attack detection module. Unlike prior works that rely on hardware performance counters (HPCs) to identify potential threats, our detector is explicitly designed to capture behavioral signatures of various timing side-channel attacks. It can detect ongoing attacks in real time. It also accurately labels both the processor core initiating the attack and the targeted LLC cache set.
2) **Fine-Grained Cache Access Control**: The LLC controller leverages detection results from the *Guardian Detector* to enforce cache set-level access restrictions selectively. Access from identified attackers is constrained, while benign processes maintain unrestricted and efficient access to the cache.

A. Overall Structure

To support the implementation of *CacheGuardian*, we extend the baseline LLC module with additional hardware logic and interface modifications. Figure 1 illustrates the overall architecture of CacheGuardian.

We first optimize the LLC controller to enable extended interactions with processor cores. Specifically, we modify the CPU-side interface to collect additional private core information, allow condition-specific signaling to the *Guardian Detector*, and adapt the cache access handling logic accordingly. In addition, we introduce several new hardware structures:

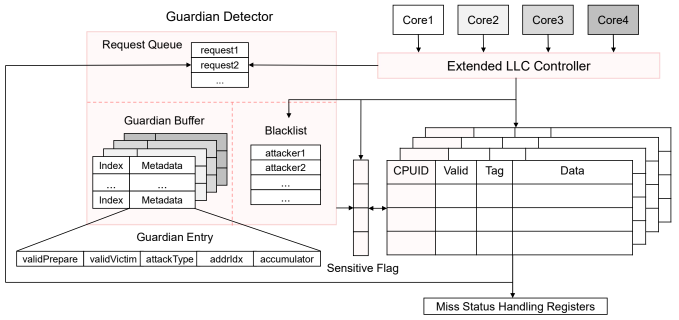


Fig. 1. Overall structure of CacheGuardian.

- **CPUID Metadata**: Each cache set in the LLC is augmented with a metadata field, `CPUID`, which records the last accessing core of the corresponding cache block.
- **Guardian Detector**: The core component responsible for identifying attack behaviors. It includes a dedicated storage structure, the *Guardian Buffer*, which is partitioned into N_{core} associative ways—one per CPU core. Each way contains multiple fully-associative *Guardian Entries*, indexed by cache set addresses and managed using a random replacement policy.
- **Guardian Entry**: Each entry corresponds to a specific cache set and maintains the following metadata: `validPrepare`, `validVictim`, `attackType`, `addrIdx`, and an `accumulator`, that tracks the frequency of suspicious access patterns.
- **Sensitive Flags and Blacklist**: A bit vector of size N_{cacheset} marks cache sets flagged as sensitive by the *Guardian Detector*. Additionally, a *Blacklist* is used to track the CPUIDs of processor cores identified as attackers.
- **Guardian Detector Request Queue**: A buffering queue between the *Guardian Detector* and LLC controller that temporarily holds unprocessed requests.

B. Behavior-Driven Generic Attacks Detector

In Section II, we analyze the execution patterns of various timing-based cache side-channel and covert-channel attacks, including reuse-based, conflict-based, and coherence-state-based channels. Despite differences in their implementation, we observe that these attacks follow a common three-phase execution pattern, which we abstract into the following 3 stages:

- **AttackerPrepare**: The attacker prepares the cache by evicting the target cache set using precomputed eviction sets (as in conflict-based attacks), or by issuing flush-like instructions (as in reuse/coherence-based attacks), which invalidate specific cache lines and trigger writebacks to memory. In both cases, the LLC receives eviction-related

signals originating from the attacker's core (denoted as `core_A`).
- **VictimAccess**: The victim (running on a different core, `core_V`) accesses the same cache set, leading to a cache miss. This access generates a new entry in the LLC's Miss Status Handling Registers.
- **AttackerObserve**: The attacker re-accesses the evicted addresses or re-flushes the targeted address, measuring access latency to infer the victim's access behavior. This step again involves memory accesses to the previously targeted cache set.

A successful attack attempt is completed when all three phases occur in sequence for the same cache set. Based on this insight, we extend the LLC controller logic and introduce the *Guardian Detector*, a dedicated hardware module that monitors and records the sequence of these phases. The detector maintains a per-core attack history and tracks the number of complete attack attempts (attack rounds) observed for each cache set. Once the attack count exceeds a configurable threshold, the *Guardian Detector* flags the attacker's core in a *Blacklist* and marks the corresponding cache set as sensitive in a *SensitiveFlag* bitmap. The LLC controller then uses these flags to enforce fine-grained access control to mitigate the attack.

To effectively track and mitigate malicious behavior in timing-based side-channel attacks, we augment the LLC Controller with enhanced control logic and metadata while designing a set of communication primitives to enable request transmission to the *Guardian Detector*. Specifically, each cache line in the LLC is extended with a `CPUID` field, which records the core ID of the most recent accessor. When a memory access request from a processor core is processed, the LLC stores the requester's core ID in the corresponding cache line's `CPUID`. In parallel, the LLC controller issues an `ATTACKER_OBSERVE` request to the Guardian Detector Request Queue (GDRQ), containing the source `CPUID` and the `ADDR_IDX`, which identifies the index of the target cache set.

During the tag-matching process across the cache set, if all cache blocks within a set are found to originate from the same core (i.e., their `CPUID` fields match), the LLC controller infers that a single core has fully manipulated the cache set. In this case, it emits an `ATTACKER_PREPARE` request to the GDRQ, including the associated `CPUID`, `ADDR_IDX`, and the `ATTACK_TYPE`, which is set to `CONFLICT`. Additionally, when the LLC controller receives an 'invalidate' request, triggered by flush-like instructions or back-invalidation mechanism, it interprets this as another form of attacker preparation. An `ATTACKER_PREPARE` request is again sent to the GDRQ, but the attack type is marked as `INVALID`.

Finally, when an access from a different core results in an LLC miss, the Miss Status Handling Register (MSHR) processes the request and signals a `VICTIM_ACCESS` event to the GDRQ, containing the `CPUID` and `ADDR_IDX`. This event marks the second phase of a potential attack attempt.

The *Guardian Buffer* is the primary storage component of the *Guardian Detector*, responsible for tracking behaviors indicative of cache-based attacks. The complete buffer is partitioned into N_{core} isolated segments, each dedicated to storing information for a specific processor core, and each segment operates independently from the others. Every *Guardian Entry* within the buffer maintains two metadata fields, `validPrepare` and `validVictim`, which together represent the current state of the entry (`IDLE`, `PREPARED`, `ACCESSED`). These states indicate the phase of a potential attack targeting the corresponding cache set. Figure 2 illustrates the finite-state machine (FSM) that governs the transitions between these states. Upon receiving requests from the Guardian Detector Request Queue, the detector executes the corresponding actions:

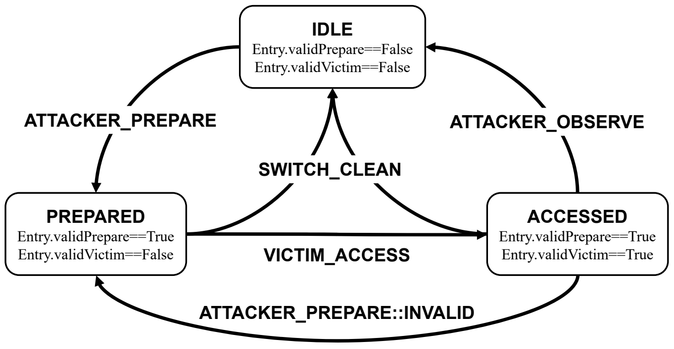


Fig. 2. Finite-state machine (FSM) for Guardian Entry state transitions.

- **ATTACKER_PREPARE**: The detector selects the buffer segment corresponding to the `CPUID` and indexes into the appropriate entry using `ADDR_IDX`. If the entry is missing, it is allocated using a random replacement policy. The entry's `addrIdx` and `attackType` fields are updated based on the request, and the state is set to `PREPARED`. Additionally, if the entry is already in the `ACCESSED` state and the new request's type is `INVALID`, the `accumulator` counter is incremented by one.
- **VICTIM_ACCESS**: The detector scans all buffer segments *except* the one indexed by the `CPUID` of the request. For any entry currently in the `PREPARED` state with a matching index, its state is transitioned to `ACCESSED`. If no match is found, the request is discarded.
- **ATTACKER_OBSERVE**: The detector searches the segment corresponding to the `CPUID` for an entry in the `ACCESSED` state with the given `ADDR_IDX`. Upon a successful match, the entry's `accumulator` is incremented and the state is reset to `IDLE`. Otherwise, the request is ignored.

The *Guardian Detector* monitors the `accumulator` field in each entry and compares it against a predefined threshold. Once this threshold is reached, an attack is considered to be detected. However, choosing an appropriate threshold is non-trivial: a low threshold may mistakenly classify benign behaviors as attacks, increasing the false positive rate; whereas a high threshold may miss camouflaged, low-frequency attacks, resulting in a high false negative rate.

Prior studies on timing side-channel attacks [1]–[3] have shown that attackers typically probe the cache every 2500–5000 clock cycles or every 60–180 μs to maximize accuracy, often repeating the sampling procedure for 15,600 to 500,000 times within a short execution interval. Even in more camouflaged and low-frequency triggered scenario [28], where the attacker attempts to extract only one bit per iteration, aligning the sampled traces still requires frequent sampling within a window. Empirical results suggest that at least 40-50 consecutive observations within a sampling window are needed to achieve a success rate of 90% or higher.

To exploit this observation and enhance detection accuracy, we introduce an `ATTACK_WINDOW`, a temporal detection window of 1 ms (equivalent to 2 million cycles on a 2 GHz processor). If the `accumulator` reaches the threshold within this window, the corresponding attack is confirmed. In that case, the Detector updates the *Blacklist* with the attacker's `CPUID` and sets the associated cache set index in the *SensitiveFlag* vector. Otherwise, the `accumulator` is reset to zero. We defer detailed analysis of threshold selection to Section V-B1.

In modern operating systems, malicious processes are typically assigned fixed-length time slices ranging from 1 to 10 milliseconds. Once a malicious process exhausts its allotted time slice, the OS triggers a preemptive context switch, replacing the current process with a new one. This switch leads to the "pollution" of the LLC cache by the new process, disrupting the attacker's ability to exploit residual cache states. To leverage this behavior, we monitor the processor's *CR3* register, which holds the base address of the active page table. A change in *CR3* indicates a context switch on the corresponding core. To capture this event, we extend the CPU-side interface of the LLC controller: upon detecting a *CR3* switch on a core, the LLC controller issues a `SWITCH_CLEAN` signal to the Guardian Detector Request Queue, including the corresponding `CPUID`. Upon receiving this signal, the *Guardian Detector* invalidates all entries in the buffer way associated with the specified `CPUID`. If the core has previously been blacklisted (i.e., listed in the *Blacklist*), it is removed. Furthermore, if the *Blacklist* becomes empty after this operation, all bits in the *SensitiveFlag* vector are cleared, indicating a reset of previously identified sensitive cache sets.

It is important to note that the LLC controller sends requests to the *Guardian Detector* in parallel with its normal cache access path. The detection logic is off the critical path of LLC request processing and does not introduce latency overhead to normal memory operations.

C. Fine-Grained Control for Malicious Requests Only

In the design of *Guardian Detector*, we introduce two storage structures—*Blacklist* and *SensitiveFlag*. The *Blacklist* is used to track the processor cores currently executing detected malicious processes, with one entry per core (up to N_{core} entries). The *SensitiveFlag* is a bit array of size N_{cacheset}, indicating potentially targeted cache sets. Notably, we maintain only a single instance of the *SensitiveFlag* shared across all entries in the *Blacklist*. This design ensures consistent protection not only for single-threaded attacks but also when a single attacker launches malicious processes across multiple cores by leveraging VM-level identifiers such as VMID.

TABLE III
GEM5 CONFIGURATION

Parameter	**Configuarations**
Processor	DerivO3CPU, 4 cores
Frequency	2GHz
L1 I/DCache	32KB,4-way 64B cache line
LLC extended with CacheGuardian	2MB,16-way 64B cache line 20 MSHRs, 12/entry

TABLE IV
SPEC2017 MIXED WORKLOADS

Workload ID	**Benchmark Pair**
W1	`mcf_s` + `gcc_s`
W2	`imagick_s` + `xalancbmk_s`
W3	`fotonik3d_s` + `nab_s`
W4	`roms_s` + `omnetpp_s`
W5	`gcc_s` + `cactuBSSN_s`
W6	`mcf_s` + `imagick_s`
W7	`xalancbmk_s` + `nab_s`
W8	`fotonik3d_s` + `roms_s`
W9	`omnetpp_s` + `cactuBSSN_s`
W10	`gcc_s` + `xalancbmk_s`
W11	`mcf_s` + `roms_s`
W12	`imagick_s` + `nab_s`
W13	`gcc_s` + `fotonik3d_s`
W14	`cactuBSSN_s` + `imagick_s`

When processing memory requests from the processor cores, the LLC controller typically parses the physical address to extract the cache set index (`ADDR_IDX`) and tag, and compares the tag against all cache lines in the corresponding set. To enforce access control, the controller also consults the *Blacklist* and *SensitiveFlag* structures. If the requesting core's `CPUID` is listed in the *Blacklist* and `SensitiveFlag[ADDR_IDX]` is set, the controller identifies this request as a potential attack and redirects it to the lower memory hierarchy without allocating or updating the corresponding cache set.

This mechanism preserves the functional semantics of cache maintenance instructions (e.g., `clflush`), while preventing the attacker from evicting or probing the targeted cache lines. As a result, any timing observations made during the `AttackerObserve` phase will constantly receive memory-level latency, effectively breaking the timing side channel.

Overall, this fine-grained access restriction ensures strict cache isolation for untrusted processes without interfering with benign applications, thereby eliminating cache-level sharing that side-channel attacks rely on.

V. Evaluation

A. Methodology

We implement CacheGuardian by modifying the last-level cache structure and its controller logic in a cycle-accurate simulator gem5 [35]. Our system configuration is listed in Table III, which consists of four out-of-order x86 processor cores, each operating at 2 GHz. Every core is equipped with a private 32 KB L1 instruction cache and a 32 KB L1 data cache. The cores share a unified 2 MB L2 cache extended with CacheGuardian, which serves as the LLC. The LLC includes 20 MSHRs, and each MSHR can accommodate up to 12 outstanding requests.

Security Analysis. To evaluate the effectiveness of *CacheGuardian*, we developed several proof-of-concept (PoC) timing side-channel attacks targeting the LLC. These include reuse-based side channels such as the classic *Flush+Reload* attack, and conflict-based covert channels exemplified by the *Prime+Probe* technique. In addition, we reproduced a camouflaged Spectre attack based on `gcc_s`, following the methodology presented in [29].

Performance Evaluation. For performance assessment, we use the SPEC CPU2017 benchmark suite [36] with the reference input set. Since our defense mechanism primarily targets shared LLCs in multicore systems, we follow the classification methodology in [37] and select nine memory-intensive benchmarks. These benchmarks are organized into 14 pairwise combinations to construct mixed workloads (Table IV), each running concurrently on separate cores. Each benchmark first executes one billion instructions to warmup the LLC, followed by another one billion instructions for the performance measurement and analysis.

B. Security Analysis

As described in Section IV, timing side-channel attacks typically consist of multiple stages to form a complete *attack chain*, where each stage is a necessary step for the success of the attack. By identifying the common prerequisites across different attacks and effectively disrupting them, we can break the entire attack chain and mitigate a broad class of timing side-channel attacks.

We observe that, in the third stage of most such attacks, the adversary reaccesses a target memory address and infers sensitive information by measuring the access latency. In CacheGuardian, the extended LLC controller restricts the attacker's access to the specific cache set identified as sensitive. As a result, the attacker can only observe a fixed memory access latency, regardless of the cache state. This effectively removes the timing variation that the attacker relies on to infer secrets.

In this section, we analyze the effectiveness of CacheGuardian in defending against various Timing side-channel attacks, including reuse-based, conflict-based, and transient execution attacks, as well as attacks exploiting Coherence Protocol state. Our analysis demonstrates that CacheGuardian effectively disrupts the attack chains by limiting attackers' access to critical cache resources and blocking covert data transmission through side channels, without introducing new security vulnerabilities.

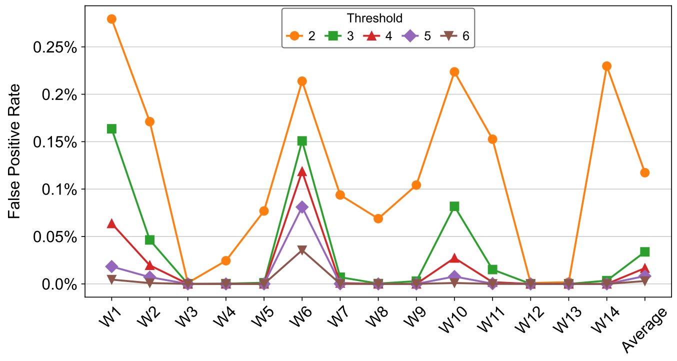


Fig. 3. Normalized False positive rate under different thresholds.

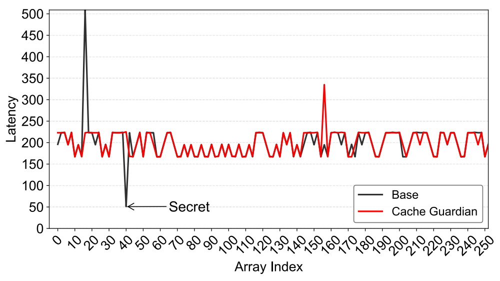
1) Threshold Selection: As introduced in Section IV, CacheGuardian adopts a threshold-based detection scheme: when the accumulator in a Guardian Entry reaches a predefined threshold, an attack is considered to be detected. To empirically determine a suitable threshold, we fix the OS time slice length to 10 ms and run the mixed workloads listed in Table IV under two conditions: (1) with the *Guardian Detector* enabled but without applying the `ATTACK_WINDOW` restriction, and (2) without activating the fine-grained access limitation mechanism of CacheGuardian. We evaluate the false positive rate (FPR) of benign workloads under different thresholds. The normalized results are shown in Figure 3.

Since reuse-based attacks and Coherence state attacks (e.g., via `clflush`) are generally more precise and aggressive than conflict-based attacks, we conservatively set the threshold for `CONFLICT` attacks to 4, and that for `INVALID` attacks to 2. These values are significantly lower than the frequency of attack attempts observed in prior work [1], [2], [28], which typically conduct dozens of accesses per attack window. Thus, our chosen thresholds are sufficient to detect attacks effectively and quickly.

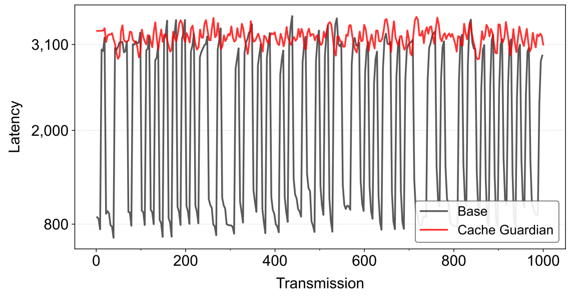
It is worth noting that once the hardware-enforced `ATTACK_WINDOW` is activated, it becomes more difficult for benign programs to accidentally exceed the detection threshold. In rare corner cases, if multiple memory-intensive benign processes generate a high volume of cache activity and exceed the threshold, the associated `CPUID` entries will be automatically removed from the *Blacklist* once their time slice expires. Therefore, such false positives have a limited impact on overall system performance (Section V-C1). We use the thresholds above in all subsequent experiments and analyses.

2) Evaluation on Reuse-Based and Conflict-Based Attacks: Due to limitations in experimental platforms and the availability of open-source implementations of existing attacks, we developed our own PoC code targeting LLC-based timing side-channel attacks to evaluate the effectiveness of CacheGuardian.

For reuse-based attacks, we deploy the attacker and victim on separate cores. The victim continuously accesses a 256-byte memory region, while the attacker applies a Flush+Reload technique to infer which specific byte has been accessed,



(a) Reuse-based attack latency



(b) Conflict-based attack latency

Fig. 4. Attacker-measured access latency in reuse-based and conflict-based attacks

attacking one byte at a time. Figure 4a shows the latency distribution collected by the attacker for a single byte under two settings: with and without CacheGuardian enabled. The results indicate that, with CacheGuardian, the attacker can no longer distinguish the victim's access patterns based on latency differences. In our test environment, the attack success rate without any protection approaches 100%, while it drops to 0.391% with CacheGuardian enabled—an approximate 256× reduction.

To evaluate the defense against conflict-based covert-channel attacks, we implement an LLC-based covert communication channel using Prime+Probe. The Trojan and Spy processes are placed on different processor cores and communicate through selected cache sets in the LLC, transmitting a total of 1024 bits. Figure 4b shows the latency observed by the Spy during the probe phase under both protected and unprotected configurations. With CacheGuardian enabled, the latency values for transmitting both bit 0 and bit 1 converge to the same range (3000–3400 cycles), preventing the Spy from inferring the transmitted bit value. In our experiments, using a Gaussian Mixture Model (GMM) classifier, the attack achieves a success rate of 99.41% without protection, which drops to 50.83% under protection—close to random guessing.

3) Detection of Camouflaged Transient Execution Attacks: In this experiment, we evaluated the effectiveness of CacheGuardian to detect low-frequency camouflage attacks. We reproduce the Spectre speculative execution attack described in [29], where the payload is concealed using the standard library function `gcc_s` and the leaked data is exfiltrated via the LLC using a reuse-based side channel. The Spectre trigger point is injected into the `bitmap_find_bit()` function and is activated with a probability of 1 in 10,000.

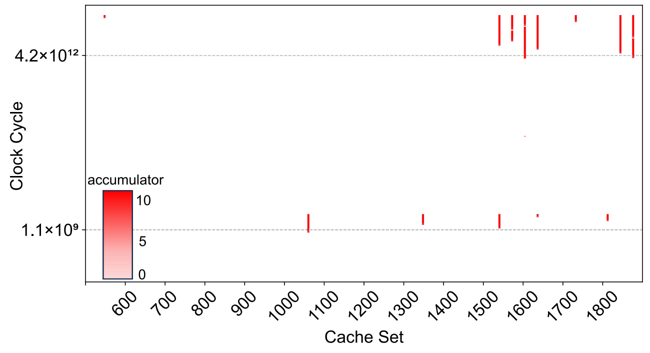


Fig. 5. Detection trace of camouflaged attacks

Figure 5 presents the detection results recorded by the *Guardian Detector* during the execution of the attack. Around the 1.1×10^9 and 4.2×10^{12} clock cycles, the detector successfully captures the attacker's covert communication behaviors targeting specific cache sets. These events are reflected in the accumulators within the corresponding Guardian Entries.

The results demonstrate that even when the attack frequency is extremely low—insufficient to cause any observable system-wide performance anomaly—the attacker must still undergo the three phases described in Section IV, enabling CacheGuardian to detect and restrict the attack activities effectively.

4) Security Analysis of Coherence Protocol State Attacks: Yao et al. [30] proposed a covert timing channel attack that exploits the coherence protocol states defined by the MESI protocol. In this attack, the Trojan process spawns two threads to collaboratively manipulate the coherence state of a shared cache line to either Exclusive (E) or Shared (S). The Spy process accesses the same cache line and infers the transmitted bit ('1' or '0') based on the access latency, which differs depending on the coherence state of lines.

Because coherence protocols ensure consistency of the same memory address across multiple cores, the Trojan and Spy processes must share the same data, making this attack similar in structure to reuse-based channels. The Spy first flushes the target cache line using instructions like `clflush`, causing it to be evicted from all cache levels, including the LLC and private caches. This invalidation is observed by the LLC controller, which issues a maintenance request and triggers the *Guardian Detector* to allocate an entry. Next, the Trojan loads the target data into the cache, establishing either the E or S state. The associated memory request is matched by the *Guardian Detector* with the existing entry. Finally, when the Spy reloads the data to observe the latency, the Detector records this access and increments the `accumulator`. Once the number of detected rounds exceeds the threshold, the Detector enforces memory-level access timing, preventing the Spy from distinguishing timing differences and thereby disrupting the covert channel.

5) Analysis of Additional Side-Channel: CacheGuardian introduces three storage structures: the *Guardian Buffer*, *Black-*

TABLE V
NORMALIZED IPC OF SPEC2017 AND NORMALIZED MSHRS MISSES

Workload	Normalized IPC	Normalized MSHRs Misses
perlbench_s	6.92%	39.77%
gcc_s	0.96%	42.63%
bwaves_s	-1.85%	70.29%
mcf_s	8.39%	69.85%
cactuBSSN_s	8.09%	49.69%
lbm_s	3.44%	50.32%
omnetpp_s	0.29%	31.97%
wrf_s	1.41%	53.45%
xalancbmk_s	0.35%	37.32%
x264_s	-1.87%	47.47%
cam4_s	21.73%	48.69%
pop2_s	-0.41%	29.65%
deepsjeng_s	-0.02%	47.34%
imagick_s	0.02%	27.12%
leela_s	0.84%	12.81%
nab_s	0.12%	27.52%
exchange2_s	-2.60%	-10.37%
fotonik3d_s	-0.10%	0.94%
roms_s	-0.10%	46.15%
xz_s	-0.48%	75.70%
Average	2.26%	39.92%

list, and *SensitiveFlag*. Both the *Guardian Buffer* and the *Blacklist* are partitioned into *N_core* isolated segments, with each core owning a separate and non-overlapping portion. This design prevents any inter-core interference or information leakage across cores. The *SensitiveFlag* is used solely by the LLC controller to enforce memory access restrictions on processes identified as potential attackers. It does not affect normal processes. Therefore, an attacker can at most infer the memory access behavior of other attackers, potentially introducing a low-bandwidth covert channel, but cannot observe or infer any memory activity of benign applications.

C. Performance Evaluation

1) Performance Evaluation with SPEC CPU2017 Benchmarks: We first evaluate the system performance under full CacheGuardian protection using the mixed workloads from Table IV. The experimental results showed that in 14 mixed workload test sets, only the 'fotonik3d + roms' combinations exhibited performance changes. Specifically, 'fotonik3d' saw a 0.19% decrease in performance, while 'roms' experienced a 0.4% performance increase. The remaining 13 workload combinations showed no performance variation. This shows that in attacker-free multi-core systems with memory-intensive workloads, CacheGuardian imposes minimal performance overhead, as it seldom activates protection—even if it occasionally misclassifies and restricts benign programs' access to the LLC.

Additionally, we conduct another set of experiments in which reuse-based attack programs were deployed alongside SPEC CPU2017 benchmarks on different cores, assessing CacheGuardian's performance impact on the benchmarks. Table V shows the performance changes of each benchmark after enabling CacheGuardian protection. We observed that some benchmarks experienced over a 5% performance improvement (such as `perlbench`, `mcf`, `cactuBSSN`, and `cam4`). This is due to CacheGuardian restricting any malicious process access to specific cache sets, which allowed the benchmarks to utilize more LLC cache resources, improving memory access efficiency.

However, performance losses are observed in some benchmarks. We attribute this to the fact that enabling protection led to malicious processes causing more LLC misses and generating additional miss requests to the MSHRs, competing with normal benchmark processes for MSHRs resources. Table V presents the increase in LLC MSHRs miss counts caused by this resource competition before and after CacheGuardian protection. On average, LLC MSHRs misses increased by 39.92% after enabling CacheGuardian, reducing the MSHRs available to the benchmarks and impacting performance.

In summary, CacheGuardian results in an overall performance improvement of 2.26% in multi-core scenarios where attack programs and SPEC CPU2017 benchmarks coexist.

2) Storage Overhead Analysis: Finally, we investigate the storage overhead of CacheGuardian using the configuration from the Evaluation section. The primary storage units and their corresponding overheads are as follows:

- The *Guardian Buffer* is divided into four sections, each containing 2048 entries. The metadata of each entry includes 1 bit for `validPrepare`, 1 bit for `validVictim`, 1 bit for `attackType`, 3 bits for `accumulator`, and 11 bits for `addrIdx` (since `addrIdx` is indexed over 2048 entries). The total storage for the *Guardian Buffer* is $4 \times 2048 \times (3 + 3 + 11) = 139,264$ bits.
- The *Blacklist* stores 1 `CPUID` per entry, using 8 bits per entry. With 4 entries, the total overhead for the *Blacklist* is 32 bits. The Blacklist can be implemented using registers.
- The *SensitiveFlag* occupies 2048 bits.
- Each cache line in the LLC includes an additional `CPUID` metadata. With 2MB of LLC, there are 2^{15} cache lines, each 64 bytes in size. The storage overhead for the `CPUID` metadata in the LLC is $8 \times 2^{15} = 262,144$ bits.

The total additional SRAM usage for CacheGuardian is $139,264 + 2,048 + 262,144 = 403,456$ bits, or 49.25 KB, which accounts for **2.4%** of the 2MB LLC Data Block SRAM. When considering the SRAM used by the LLC's own metadata and other management structures, the overhead introduced by CacheGuardian is even smaller. Overall, CacheGuardian is reasonable when implemented in a modern multi-core computing system.

VI. CONCLUSION

In this work, we propose a security-oriented design named CacheGuardian. Focusing on the LLC timing side-channel vulnerabilities, we present a real-time detection mechanism against multiple attacks, including the low-frequency triggered camouflage attack. CacheGuardian also achieves a trade-off between performance and security. In the future, the rise of artificial intelligence will introduce a wider range of multi-tenant scenarios in the cloud, leading to various side-channel attacks. CacheGuardian can also be applied to protect these applications.

References

[1] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.

[2] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[3] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 591–604.

[4] 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 *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 888–904.

[5] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Last-level cache side-channel attacks are feasible in the modern public cloud," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 582–600.

[6] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: a fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer, 2016, pp. 279–299.

[7] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive {Last-Level} caches," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 897–912.

[8] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 360–371.

[9] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+ Abort: A Timer-Free High-Precision l3 cache attack using intel TSX," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 51–67.

[10] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 395–410.

[11] 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 *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2016, pp. 406–418.

[12] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 974–987.

[13] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid Side-Channel-Resilient caches for trusted execution environments," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 451–468.

[14] L. Li, J. Huang, L. Feng, and Z. Wang, "P refender: A prefetching defender against cache side channel attacks as a pretender," *IEEE Transactions on Computers*, 2024.

[15] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani, "Defeating cache timing channels with hardware prefetchers," *IEEE Design & Test*, vol. 38, no. 3, pp. 7–14, 2021.

[16] H. Fang, M. Doroslovački, and G. Venkataramani, "Reuse-trap: Re-purposing cache reuse distance to defend against side channel leakage," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[17] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 775–787.

[18] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 675–692.

[19] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 360–371.

[20] W. Song, Z. Xue, J. Han, Z. Li, and P. Liu, "Randomizing set-associative caches against conflict-based cache side-channel attacks," *IEEE Transactions on Computers*, vol. 73, no. 4, pp. 1019–1033, 2024.

[21] M. Li, K. Bu, C. Miao, and K. Ren, "Treasurecache: Hiding cache evictions against side-channel attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 5, pp. 4574–4588, 2024.

[22] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.

[23] J. Zhang, C. Chen, J. Cui, and K. Li, "Timing side-channel attacks and countermeasures in cpu microarchitectures," *ACM Computing Surveys*, vol. 56, no. 7, pp. 1–40, 2024.

[24] F. Jiang, F. Tong, H. Wang, X. Cheng, Z. Zhou, M. Ling, and Y. Mao, "Pcg: Mitigating conflict-based cache side-channel attacks with prefetching," *arXiv preprint arXiv:2405.03217*, 2024.

[25] C. Miao, K. Bu, M. Li, S. Mao, and J. Jia, "Swiftdir: Secure cache coherence without overprotection," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 662–677.

[26] D. Ojha and S. Dwarkadas, "Timecache: Using time to eliminate cache side channels when sharing software," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 375–387.

[27] S. M. Ajorpaz, D. Moghimi, J. N. Collins, G. Pokam, N. Abu-Ghazaleh, and D. Tullsen, "Evax: Towards a practical, pro-active & adaptive architecture for high performance & security," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1218–1236.

[28] J. Jiang, C. Soriente, and G. Karame, "On the challenges of detecting side-channel attacks in sgx," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022, pp. 86–98.

[29] W. Kosasih, Y. Feng, C. Chuengsatiansup, Y. Yarom, and Z. Zhu, "Sok: Can we really detect cache side-channel attacks by monitoring performance counters?" in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 172–185.

[30] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 168–179.

[31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom *et al.*, "Meltdown: Reading kernel memory from user space," vol. 63, no. 6. ACM New York, NY, USA, 2020, pp. 46–56.

[32] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[33] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "Tlb poisoning attacks on amd secure encrypted virtualization," in *Proceedings of the 37th Annual Computer Security Applications Conference*, 2021, pp. 609–619.

[34] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[35] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[36] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[37] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a decade: Did spec cpu 2017 broaden the performance horizon?" in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 271–282.