Preventing Coherence State Side Channel Leaks Using TimeCache

Divya Ojha and Sandhya Dwarkadas, *Fellow, IEEE*

Abstract—Cache side channel attacks in the presence of shared memory have been used to extract cryptographic keys and enclave data, and are used by Spectre variants for leaking speculatively loaded data. Timing side channels exist in shared caches due to the difference in response latency of cached and uncached data. In prior work, we presented TimeCache, a cache design that prevents side channel exploits from reuse of shared memory. In this work, we extend TimeCache to also defend against attacks that exploit coherence states. TimeCache allows all running applications to use the entire cache, avoiding the need for partitioning in order to effect timing isolation. A per-process caching context prevents cache hits on data filled by another process. A novel bit-serial timestamp-parallel comparison logic allows low-overhead update of stale caching contexts. The defense is suited to all caches levels, and defends against an attacker running on any core. We evaluate TimeCache using the gem5 simulator to show that it is capable of preventing both reuse attacks and an attack based on coherence state leak. The average performance overhead for SPEC2006 is 1.13%, and for PARSEC and SPLASH is 0.46%.

Index Terms—Cache memories, support for security, security and privacy protection

1 Introduction

The modern processor cache hierarchy has been shown to be vulnerable to various side channel attacks. Side channel attacks exist due to shared hardware, and get stronger in the presence of shared libraries or software. These attacks can be used to extract cryptographic keys and data across even secure enclaves. Several attacks and defense techniques have been explored in the literature [19].

Several known attacks accomplish data leak by observing the caching behavior of shared memory [7], [9], [14], [18], [41], [44]. Shared memory or shared libraries help significantly reduce memory footprint and are employed to reduce overall system cost. Reuse attacks are a category of side channel attacks on shared memory where an attacker primes an expectation of a slow access by flushing the shared memory location from the cache prior to timing an access to shared memory, but experiences a fast memory operation due to a victim's access [41]. This form of attack is precise and hence a handy tool for crafting more sophisticated attacks on out-of-order processors [4], [14], [33]. Preventing reuse attacks can allow the system providers to deploy deduplication and boost performance by up to 40% [30]. Similarly, the coherence state of cached lines from shared memory regions can also leak information due to timing differences in accessing data in *Shared* or *Exclusive* state [39]. This work extends TimeCache, our prior defense against reuse attacks [20], to also prevent side channel attacks resulting from coherence state leak.

Other existing solutions to mitigate reuse attacks either resort to cache partitioning [5], [13], [22], [35], [36] or rely on a constant time implementation [16], [26], [27]. Cache partitioning effectively reduces cache capacity available to each process. Some of the cache partitioning defense techniques might also be suited only to the last level cache [17], [38], or might need predefined security domains [13]. The constant time implementations incur a high performance penalty, as every access is padded with additional accesses, increasing the run-time significantly.

TimeCache prevents reuse attacks by creating a "per-process view" of cache line arrival, while allowing entire cache utilization. It incurs a miss on a resident cache line for a *first access* by any process, which breaks the very construct of the attack. The attacker expects to learn other process's access pattern by incurring a hit due to the other process. The additional miss occurs only when the data is evicted and reloaded, and hence the performance of steady-state in-cache sharing is unaffected. TimeCache can also prevent coherence state leaks by enforcing memory access latency (or the highest-cost access latency) for the *first access*, rather than allowing a lower access latency for data sourced at (or brought in by) another core. As a consequence of this defense, system providers can continue to utilize memory deduplication techniques and reduce overall memory footprint [1], [12] without compromosing security.

The per-process caching context is stored as per-cache-line access bits. In order to maintain the caching context across context switches, the defense uses per-cache-line *last-filled* timestamps and a novel bit-serial timestamp comparator. The process's access bits are saved to memory at the

• *The authors are with the Department of Computer Science, University of Rochester, Rochester, NY 14627 USA. E-mail: {dojha, sandhya}@cs.rochester.edu.*

Manuscript received 15 February 2022; revised 6 July 2022; accepted 5 September 2022. Date of publication 29 September 2022; date of current version 13 January 2023.
This work was supported in part by NSF under Grants CNS-1618497 and CNS-1900803, and in part by the University of Rochester.
(Corresponding author: Divya Ojha.)
Recommended for acceptance by S. Sethumadhavan and S. Devadas Guest Editors.
Digital Object Identifier no. 10.1109/TC.2022.3209922

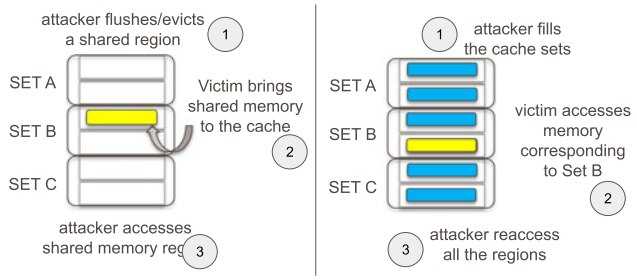


Fig. 1. Reuse (left) and contention (right) attacks.

time of a context switch along with the current time. The saved access bits are restored when the process resumes execution. The bit-serial timestamp comparator helps identify cache lines that were filled while the process was not executing, thereby resetting the access bits in the restored caching context.

We implement and evaluate TimeCache on gem5, and demonstrate its effectiveness in preventing reuse attacks using microbenchmarks and an RSA attack, as well as on attacks depending on coherence state leak. Performance evaluation shows that for SPEC2006 benchmarks, the average overhead is 1.13%, and for PARSEC and Splash2 benchmarks, the average overhead is 0.46%.

The following are the key contributions in this paper:

- Preventing side channel attack on shared software, resulting from reuse or coherence state leak, while allowing complete cache access to all the processes.
- Disallowing a cache hit on an existing cache line for the *first access* by a process.
- Creating a per-process view of cache line fill to identify *first access* and proposing a per-process caching context across context switches to prevent reuse-based attacks.
- Developing a novel bit-serial comparator for updating the per-cache-line caching context in constant time.
- Providing a simulation-based evaluation of the potential overheads associated with the defense and demonstrating that the defense is effective against attacks.

2 Background

2.1 Cache Side Channels

Memory access patterns revealed due to differences in cache access latency form *cache side channels*. Attacks exploiting cache side channels were exposed as early as in 1992 [10], and many more such attacks have evolved ever since.

There are two broad categories of cache side channel attacks, based on the presence or absence of shared software (whether instructions or data) between the attacker and the victim. Fig. 1 [20] depicts both kinds of attacks. *Reuse-based* attacks (depicted on the left) require the presence of shared memory, are more precise, and can infer a cache line accessed by the victim. The attacker expects a faster memory operation if the victim brings the shared line into the cache as a result of its memory operation. Some of the known reuse-based attacks are "flush+reload", "evict+reload" and "flush+flush" [7], [8], [41]. *Contention-based* attacks (depicted on the right) do not rely on shared software, and can learn a cache set accessed by the victim using a "Prime+Probe" [21] style of attack. The attacker brings data into all the cache sets or the cache sets of interest and reads the data again after a victim's execution. A slow access on any previously accessed data reveals that the same cache set was accessed by the victim during its execution phase.

2.2 Shared Software Attacks

Shared libraries constitute commonly used routines that can be mapped into the process space of different applications. They allow the same physical memory to be used by different processes, effectively reducing the overall system memory footprint. Shared memory improves the memory hierarchy efficiency but is vulnerable to leaking memory access patterns across processes [41]. The content of the shared library itself is not hidden from the attacker or the victim, but the access pattern to the shared library might reveal application specific data.

Side channels in shared caches become more precise in the presence of shared software. Until recently, these attacks were considered to be consequential only for cryptographic computation. However, there have been attacks inferring key strokes from another process [34] and leaking passwords across virtual machines on an Amazon EC2 cloud server [44]. More recent attacks on out-of-order processors demonstrate that these side channels can be useful gadgets for building sophisticated attacks like Spectre [14], [15], [29].

2.3 Coherence State Leak

Shared memory between distrusting entities running on separate cores can also be exploited in order to leak access information due to the ability to detect differences in the coherence states of the data. Coherence ensures that data cached in separate local caches is made consistent across the entire system: modifications made on one processor must be propagated to all copies of the data and writes to the same location by different processors must be serialized. The difference in access latency of some shared data in *exclusive*(E) versus *shared*(S) state in a remote cache allows the attacker to infer if a cache line is being accessed by the victim. The attacker runs on multiple separate cores, caching data on one core and timing accesses to the same shared data from another. The data is accessible with E latency if the victim running on a third core does not access the same data, and is accessible with S latency otherwise [39], revealing the victim's access pattern to the attacker.

2.4 Prior Solutions

Strict performance requirements from caching are important design considerations for the systems providing a defense against side channel attacks, as a slow defense might render the cache ineffective. The defense against reuse attacks either resort to cache partitioning [5], [13], [36] or obfuscate time using constant time algorithms [16], [26], [27], both of which add significant overheads. Partitioning hurts performance both due to replicating the shared data, and due to reduced effective cache available for execution. DAWG is a partitioned cache that also defends against coherence state

leaks [13]. Some defense techniques mitigate contention-based attacks (e.g., randomizing caches [23], [24], [37], SHARP [38], RPCache [36]) and are ineffective against reuse attacks. Some others might only defend the last level cache or might require prior knowledge of trust levels to group applciations into a few security domains [13], [36].

A hardware description language can be used to detect side channels in hardware through formal analysis [42]. It however requires identifying and associating different security labels with the hardware components at design time. Checkmate is another tool to detect the presence of side channels in a hardware design using relational model finding [32].

2.5 Threat Model

The threat model under consideration for this line of defense includes attacks that rely on the availability of shared memory between the attacker and the victim, i.e., both reuse-based attacks and attacks resulting from coherence state leak. We do not include contention-based attacks in the threat model.

The attacker and the victim in a reuse attack can run on the same or different cores. The attacks can be at any level of cache and the attacker can run simultaneously or interleaved with the victim. The system design does not rely on prior knowledge of separate security domains.

Access to shared memory leaks information when the accesses pattern is dependent on application-specific data. The attack is carried out in three steps; the attacker evicts the shared data either using a `clflush` instruction or by causing conflict misses; allows the victim to execute; times a subsequent operation (read or flush) on the evicted data. The latency of the memory operation reveals if the data exists in the cache hierarchy, potentially due to victim's accesses. The defense against `flush+flush` is discussed in our prior work [20]. In this paper, we focus on information leak due to reload of shared data in a shared cache or via a coherence hierarchy.

The attacks under consideration in this threat model have been shown capable of extracting RSA keys from the GnuPG shared library [41]. Recently they have also been used as building blocks for launching more sophisticated attacks like Spectre and its variants. Reuse attacks are low-noise, high bandwidth channels, and are therefore used for forming other attacks.

3 TimeCache

TimeCache prevents reuse-based attacks and attacks resulting from coherence state leak for shared memory by implementing per-process cache line visibility. Access to the same cache line results in a hit or a miss for different processes depending on their *caching context*, which allows us to retain the benefits of having a shared cache and yet prevents the attacks. The caching context of a process is the footprint of the accessed cache lines, accesses to which do not reveal any information to the attacker.

3.1 First Access

A *first access* by a process to a resident cache line is defined as the first instance of access since the cache line was brought in by some other process. There can be as many *first accesses* to a resident cache line as the number of processes accessing that cache line. The importance of *first accesses* lies in the formation of the attack. The attacker times its *first access* to data after evicting it from the cache, with a hit access latency implying that the data was accessed by the victim. TimeCache uses this key observation and enforces misses on *first access* to provide timing isolation. Future accesses by the same process will hit in the cache. Timing isolation is achieved without cache partitioning, retaining the ability to share the entire cache and avoiding the pitfalls of potentially multiple copies of the shared data in the partitioned approach. Once a cache line is evicted and brought back, all the processes accessing it will again experience *first access* misses.

An access by a process to a conventional cache may incur four kinds of misses: cold, capacity, conflict, or coherence, which may be due to its own prior accesses or due to accesses by other processes. Conversely, a process's cache accesses may experience a hit on data brought into the cache by itself or another process. This latter type of hit due to data brought into the cache by another process that forms a timing side channel, and is the target for elimination in TimeCache. TimeCache thus incurs a fifth kind of miss known as the *first access* miss.

Incurring a miss on *first access* requires identifying if a data request from a process is its *first access* to the existing cache line. To achieve this differentiation in the hardware, we add a per-execution-context security-bit (*s-bit*) to each cache line. The *s-bit* represents if the cache line has been accessed by the execution context. There are as many *s-bits* for a cache line as the number of execution contexts. The *s-bit* remains reset for a cache line unless the execution context has accessed a cache line and is reset when a line is evicted. This representation of accessed cache lines in the form of *s-bits* forms the caching context for a process.

A cache line fill is accompanied by setting the *s-bit* for only the execution context whose access results in the cache fill. The *s-bits* of all the other execution contexts remain reset until there is a request from those contexts to the cache line (see Fig. 3b).

3.2 TimeCache Accesses

Fig. 2 [20] shows the TimeCache hardware overview as modifications to a conventional cache. The example representation has a core with two hyperthreads and a cache with 8 cache lines. The additional hardware timestamp comparator under the design has the following components:

- An *s-bit* with each cache line, for every execution context. In the example here with 8 cache lines, the number of *s-bits* per cache line is 2, shown as 2 8-bit arrays *T1 s-bits* and *T2 s-bits*.
- A per cache line timestamp register *Tc*, to hold the time at which the cache line was filled.
- A transpose gate and bitline peripheral logic to perform bit-serial comparison.
- A shift register *Ts*, to hold the timestamp at which the context was preempted.

Fig. 3 [20] represents the actions at process creation, memory access, and preemption. The *Ts* and *s-bits* of a newly created process are initialized to 0. A conventional cache access

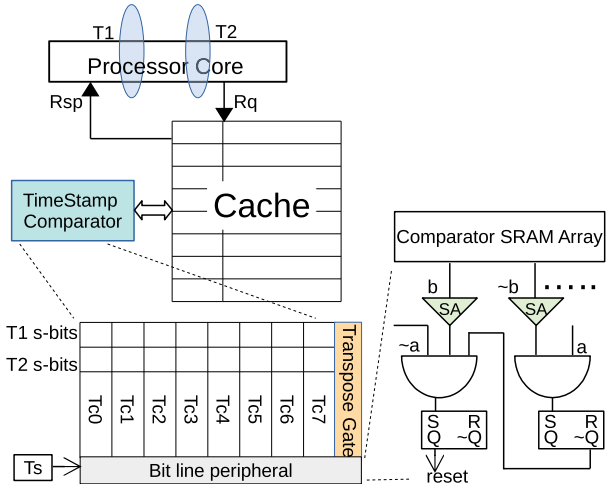


Fig. 2. TimeCache hardware, depicted for a cache with 8 lines accessed by a core with 2 hardware contexts.

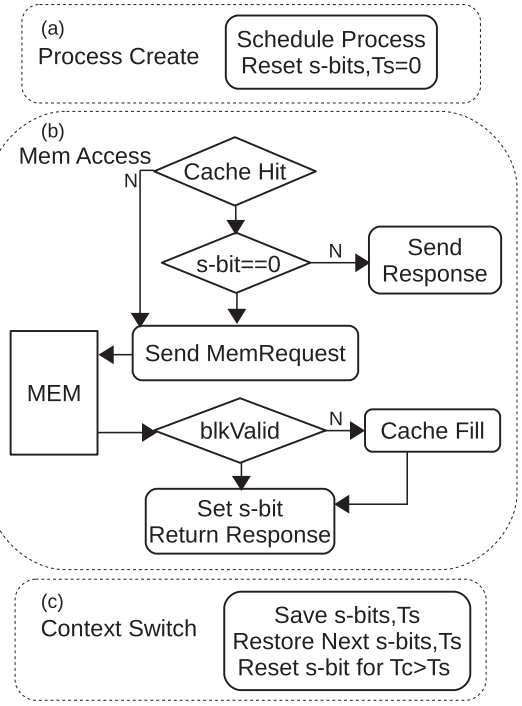


Fig. 3. Maintaining timing isolation: per-process flowchart.

proceeds by comparing the stored tags in a cache set with the tag-bits of the requested address. A resulting match on a valid cache line results in a cache-hit, otherwise a miss is incurred and the request is sent down to the next level in the memory hierarchy. TimeCache checks the *s-bit* of a cache line in addition to the above, and an access results in a cache-hit only when the *s-bit* is set, in which case the data is sent to the requesting processor.

A reset *s-bit* indicates that the cache line has not been accessed before, and that the access is a *first access*. A miss is incurred to delay the response to the processor, and the request is sent down the memory hierarchy. Upon receiving a response to this request, the received data is not filled if the data already present in the cache is still valid. The *s-bit* for this execution context is set at this instant to allow future access to go through as a cache hit. A cache access results in a cache hit only if the the corresponding *s-bit* is set. The *s-bits* are modified as follows:

- Restored from memory on a resuming context.
- Reset by the comparator at context switch if *Tc* is greater than *Ts*.
- Reset when the cache line is evicted.
- Set on a cache line fill, for the requesting hardware context.
- Set on a *first access* to an existing cache line.

It is possible that the *s-bit* for some requested data is reset in the higher-level cache (closer to the processor) and set in a lower-level cache. This can occur if a previously accessed data is replaced from a higher-level cache due to conflict misses, and is retained in the lower-level cache. In this case, the request is serviced with the lower-level cache access latency.

3.3 Handling Context Switches

The caching context represented by the set of *s-bits* is specific to the process running on the hardware execution context, and must be managed carefully when a different process gets the context. The *s-bits* of one process should not affect the accesses of another process, and the potentially stale caching context of a process should be updated when restored.

TimeCache uses a combination of hardware and the software to save, restore, and update the caching context at the time of a context switch. Software saves the entire caching footprint of the process being preempted, along with the time of the context switch (*Ts*). It then transfers the similarly stored caching context of the process resuming execution into the corresponding *s-bits* for that execution context.

TimeCache hardware ensures that the potentially stale restored caching context is updated to reflect the current state of the cache. It is possible that some of the cache lines that were accessed when a process was last run are either evicted or reloaded when the process was scheduled out. Thus, the *s-bits* for those lines are not representative of the caching context anymore, and must be reset for security. Such cache lines are identified when the process resumes execution based on a comparison between the time at which the cache lines are filled (*Tc*) and the time when the process was scheduled out (*Ts*). The comparison is triggered only at a context switch, and after restoring the process specific *s-bits* and *Ts*. *Ts* is the instant before which the *s-bits* were up-to date. The *Tc* of cache lines reloaded since the process was scheduled out will be greater than the restored *Ts* of the process.

Comparing the *Tc* of individual cache lines with *Ts*, and performing this comparison for all the cache lines at context switch can add significantly large overhead for bigger (e.g., last-level) caches. To solve this problem, the comparisons are performed in parallel with a bit-serial comparator, as explained in the following subsection.

Saving and restoring the caching footprint allows TimeCache to enjoy fast data access across context switches as long as the data is not evicted from the cache. Thus, our design leverages access locality across context switches while at the same time provides timing isolation. We use the operating system to do the bookkeeping and provide isolation across process boundaries. However, any trusted computing base

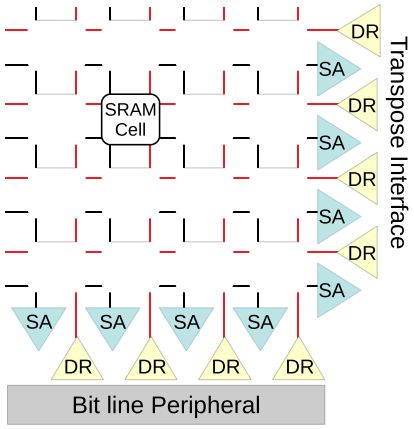


Fig. 4. Transpose SRAM array for timestamps.

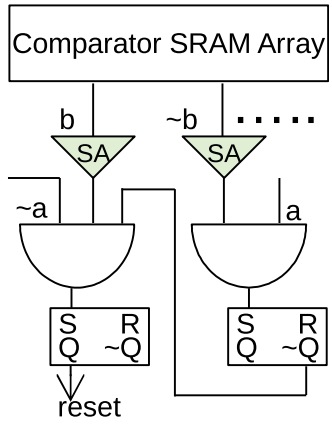


Fig. 5. Bit-line peripheral for comparison.

can be used to perform the bookkeeping, to provide side channel isolation across execution boundaries. For instance, when executing in enclaves with an untrusted operating system, the caching context save and restore can be performed at enclave entry and exit using the enclave libraries.

3.4 Timestamp Comparator: Bit-Serial, Timestamp-Parallel Comparison of Timestamps

A regular access to the timestamp comparator for checking an *s-bit* or for updating *Tc*, is bit-parallel, i.e all the bits related to a cache line may be accessed at a time. However, accessing the comparator in this manner for *s-bit* update at the time of a context switch will require time proportional to the number of cache lines. This overhead can be reduced by accessing *(Tc)* and the *s-bits* in a transposed manner to be able to perform bit-serial timestamp-parallel comparisons, similar to neural cache [6]. Thus, the comparison takes time linear in the number of *s-bits* and timestamp bits per cache line rather than linear in the number of cache lines.

3.4.1 Transpose Interface

The SRAM array is constructed using 8-T bit cells [6] and with two sets of sense amps and drivers to allow transposed access to the stored data. Fig. 4 [20] shows the comparator array with additional sense amps and drivers at the 'transpose interface'. The transpose gate is used for regular cache operations like updating a cache line timestamp on a cache line fill, resetting the *s-bits* on eviction, setting an *s-bit* on *first access*, or for reading an *s-bit* at the time of cache access. The bit-line peripheral interface on the other hand is used for *s-bit* saves and restores, and for performing parallel comparisons of all the timestamps simultaneously at a context switch. Timestamp rollover is discussed in Section 5.3.

3.4.2 Bit-Serial Comparison Logic

The greater of two unsigned integers can be determined by comparing their bits starting from the MSB (most significant bit): the comparison continues until non-identical bits are found at a bit position: the greater of the two numbers has the first non-identical bit position set. The logic for the comparison at each bit position starting from MSB can be summarized as follows:

- if both the numbers have the same bit value for a bit position, increment the bit position and continue the comparison.
- if the bits at the bit position are different, the number with a set bit is determined to be the greater of the two numbers, and the comparison can be terminated.

For instance, the greater of '1100' and '0101' is '1100' because the non-identical bit position is closest to the MSB.

The parallel comparison of the timestamps begins by loading the *Ts* into the shift register. The iterative comparison of bits is carried out by reading *Tc* from the SRAM array using the bit-line peripheral interface, and shifting in the bits from *Ts* into the bit-line peripheral logic. Starting from the MSB,

- If *Tc[i]* = 0, Ts[i] = 1; comparison stops, *s-bit* does not reset.
- If *Tc[i]* = 1, Ts[i] = 0; the bit-line peripheral latches '1', this output is used to reset the *s-bit*.
- If *Tc[i]* = *Ts[i]*; continue comparison.

Fig. 5 [20] shows the logic at the bit-line peripheral to perform the comparison and to reset the *s-bit*. The comparison can be done using 2 S-R latches, a 3-input and gate and a 2-input and gate. The *Ts* bits are 'a' and *Tc* bits are represented as 'b' in the figure.

As multiple timestamps get compared, the comparison of two timestamps can stop once a *Tc* is determined to be smaller than *Ts*. The output of the right and is set when *Tc* is less than *Ts*, and latched in the right S-R latch. Further comparisons are stopped by sending $\bar{Q}$ from the right S-R latch to the left and gate.

The left S-R latch registers the output of the left and gate if $Tc > Ts$. At the end of all the comparisons, the latched signal form the left S-R latch is used to reset the *s-bit* for the corresponding line. This is done by activating the wordline for the *s-bits*, and the bit-line drivers for which the S-R latch is set.

4 TimeCache: Mitigating Coherence Based Attacks

TimeCache can be used to mitigate attacks based on coherence state leaks, as these attacks are very similar to reuse-based attacks by construction. The attack requires the presence of shared memory between the attacker and the victim, like reuse-based attacks. The attacker and the victim,

TABLE 1
Evaluation Setup

Real Processor	
Core	i7-7700, 3304.125
L1D, L1I, L2, LLC cache	32K, 32K, 256K, 8192K
gem5 Simulator	
Core	TimingSimpleCPU, 2GHz
L1D, L1I, LLC cache	32K, 32K, 2048K

however, run on separate cores, and the coherence protocol maintains data consistency for shared data across cores.

The attack is based on the difference in cache access latency for data available in *shared* versus *exclusive* state. The difference in *E* and *S* access latency may exist because the coherence directory may hold a copy of the shared data, and service requests for the data in *S* state. A request to *exclusive* data must, however, be serviced by the core holding the data since it may have been modified.

In a two-pronged attack, an attacker may continuously evict and time its accesses from some core C1, while putting the data in *exclusive* state by accessing it from another core C2 [39]. Since the access that it times after eviction is the *first access* from that execution context, the attack depends on timing a *first access*. The attacker expects to incur an *E* access latency if the victim running on yet another core C3 does not access the same data, and *S* access latency otherwise. This is similar to the attacker in a reuse-based attack expecting a cache access latency for operation on data accessed by the victim.

In TimeCache, the *first access* by any core to a cache line results in a miss, thereby eliminating an *S* access latency. The attacker thread timing accesses on core C1 does not experience LLC access latency, as the *s-bit* for the *S* cache line in LLC is not set for C1, even if it is set for C2 and/or C3. The request from C1 is sent down the memory hierarchy instead, and the response is sent to the processor after incurring the latency associated with a miss. Removing the *S* access latency is sufficient to prevent the specific attack where the attacker expects *S* access latency due to victim's access. We further extend TimeCache to prevent *E* access latency, as a fast access due to another core can still result in an attack. We disallow the cache holding *exclusive* data from servicing data requests, unless the available data is dirty. The design allows data movement for modified shared data from one cache to another while eliminating the timing side channel.

5 Performance Evaluation

We implemented TimeCache in the gem5 cycle-accurate simulator [3] using L1I and L1D caches of 32KB each and an L2 (LLC) cache of 2MB. We added a timestamp and a per-hardware-context *s-bit* to each cache line, which are manipulated as described in Section 3. The process context for a request packet in the cache is determined by the CR3 register within the simulator. Changes in the CR3 register are used to trigger the timestamp comparisons and the *s-bit* saves and restores.

Table 1 [20] specifies the real and simulation system parameters used for the evaluation.

The following subsections present an analysis and evaluation of the security and the performance overheads of our timestamp-based defense on the gem5 simulator.

5.1 First Access and Exclusive Response Delays

We evaluate the performance overhead of our first-access delay mechanism due to context switches on a single core by simulating benchmarks from SPEC2006 for 1 billion instructions in gem5 using full system simulation mode. We run two instances of each SPEC2006 benchmark on a single core with and without TimeCache. Fig. 6 [20] presents the normalized execution time (execution time using TimeCache/execution time without TimeCache) of each benchmark. When running two instances of the same benchmark, the number of first accesses is impacted by sharing benchmark-specific code and shared libraries in the shared caches while context switching across these processes. For instance, while running two instances of h264, the memory shared between the processes includes benchmark-specific code and the libc routines for file operations like fopen, lseek, memset, and free. In addition to the above, kernel-space

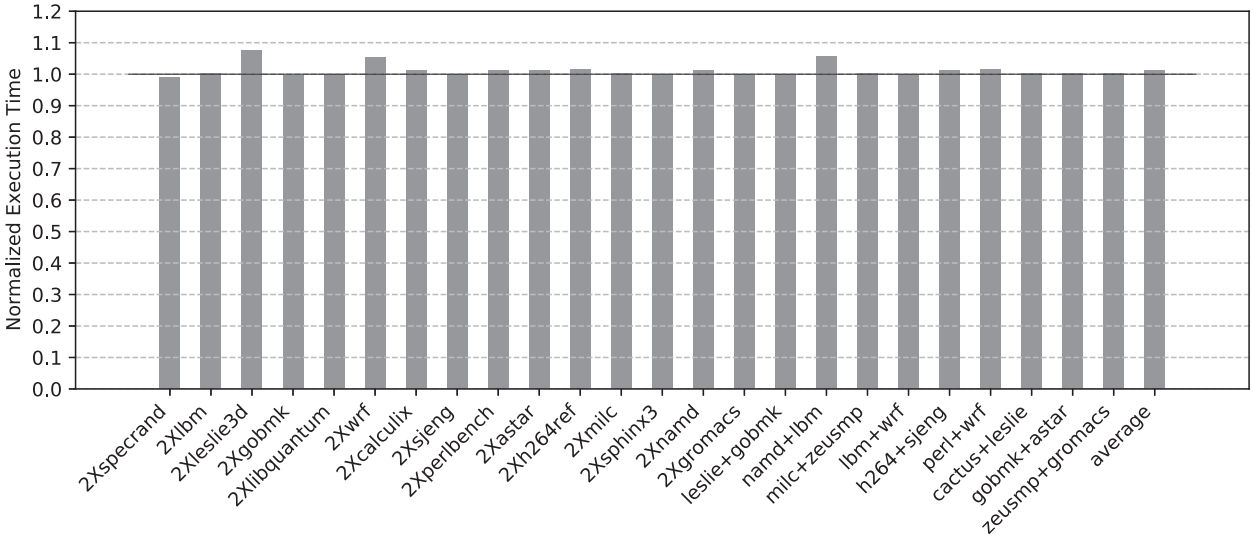


Fig. 6. Single-core SPEC2006 performance normalized execution time due to TimeCache's delayed first accesses (execution time with TimeCache/execution time without); The average overhead is 1.13% for two instance of same or different benchmarks on a single core.

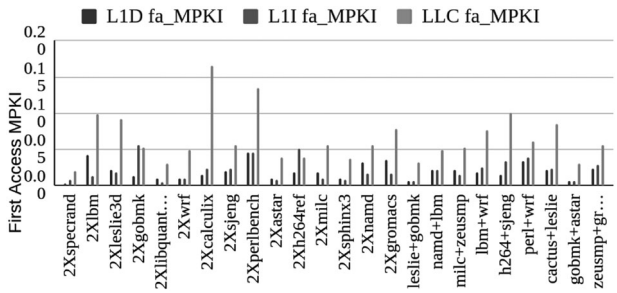


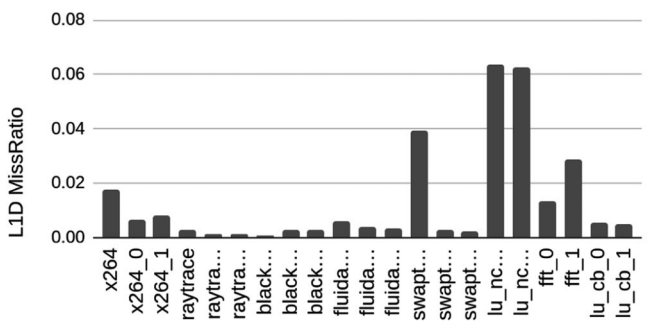
Fig. 7. Delayed access MPKI at each cache level for single-core experiments.

memory is shared across processes and accesses to kernel subroutines, system calls, and kernel data structures may incur *first access misses* when executing in privileged mode within a process context. We also run a combination of different benchmarks on a single core, where the shared access is limited to shared libraries and kernel space memory. The geometric mean of overheads across all workloads is 1.13%.

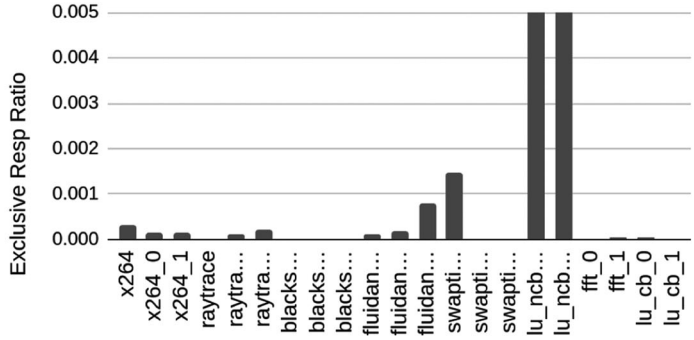
Fig. 7 [20] shows *first access* misses per thousand instructions. The last-level cache is expected to have a greater number of *first access misses* compared to the L1 cache, as it is larger and retains more shared content. The larger first access MPKI for wrf and perlbench is due to their larger shared instruction memory footprint: they have a higher first access MPKI in the last-level cache when running two instances of the same benchmark (i.e., wrf with wrf and perlbench with perlbench). However, when wrf and perlbench are run together, their effective *first access misses* are lower because of cache contention. Similarly, lbm and leslie3d also have lower effective *first access misses* due to capacity evictions when sharing the cache with namd and gobmk.

We also evaluate the overhead due to the defense against coherence state attacks when running pthread-based PARSEC and SPLASH benchmarks run on multiple cores. We use the simulator's system emulation mode in these evaluations, where the clone syscall is emulated to instantiate a new thread on another core. We use 2 threads/cores in our evaluation. Fig. 8a shows the L1D miss ratio in the baseline cache. Fig. 8b shows the corresponding ratio of responses received from a remote cache with cache line in *E* state. These responses are prevented in the defense and add an average overhead of 0.23%. The ratio of delayed responses for most benchmarks is low, with the exception being lu_ncb with a ratio of 2.2%. Fig. 9a shows the *first access* misses per thousand instructions, which are a result of cache lines in *S* state: the first access MPKI is a maximum of 0.75. The cumulative slowdown from delaying the *first accesses* and the *exclusive* state responses results in an overall slowdown of 0.46%, as shown in Fig. 9b.

The exact overheads and the change in the number of misses per thousand instructions (MPKI) for the last-level cache is presented in Table 2. The increase in execution time is proportional to the increase in MPKI, which changes both due to additional *first accesses* and due to the change in caching behavior from incurring *first access misses*. The increase in MPKI is small, which explains the low overhead.

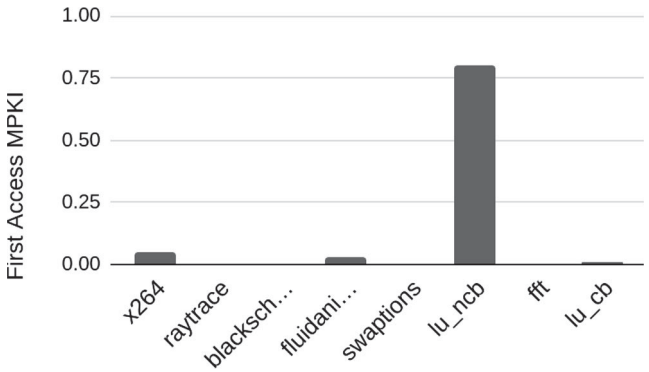


(a) The ratio of misses to total number of accesses at L1D

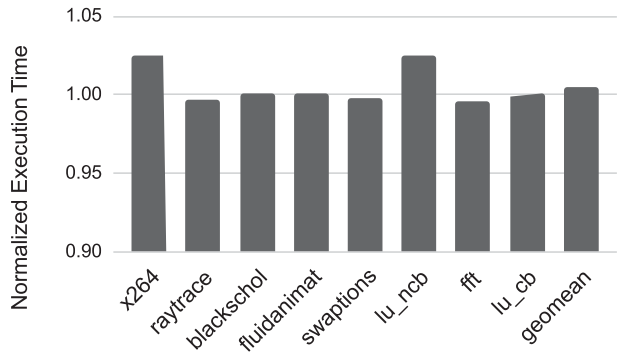


(b) Ratio of accesses receiving response from *E* state

Fig. 8. 2-core, 2-thread PARSEC and SPLASH benchmark.



(a) Number of *first access* misses per thousand instructions



(b) Normalized execution time of the defense with respect to the baseline.

Fig. 9. First access miss and overall slowdown; PARSEC and SPLASH benchmark.

TABLE 2
SPEC2006, PARSEC, and SPLASH Execution Time Overhead, 2MB LLC MPKI

Workload	Overhead	MPKI LLC Baseline	MPKI LLC TimeCache
2Xspecrand	0.9908	0.0035	0.0238
2Xlbm	1.0039	14.0349	14.138
2Xleslie3d	1.0751	20.6163	24.3556
2Xgobmk	0.9961	3.2832	3.3361
2Xlibquantum	1.0001	5.8532	5.8831
2Xwrf	1.0135	4.7286	4.8964
2Xcalculix	1.0548	0.2099	0.2672
2Xsjeng	0.999	16.7773	16.8382
2Xperlbench	1.0134	1.021	1.1582
2Xastar	1.0107	0.5654	0.6144
2Xh264ref	1.014	0.555	0.5953
2Xmilc	1.0026	16.4722	16.5295
2Xsphinx3	0.9982	0.2648	0.3118
2Xnamd	1.0108	0.1623	0.2181
2Xgromacs	0.9992	0.292	0.3703
leslie+gobmk	0.9996	22.3133	22.3669
namd+lbm	1.0579	6.3764	7.1136
milc+zeusmp	1.0024	12.5757	12.6121
lbm+wrf	1.0007	9.7181	9.7898
h264+sjeng	1.0108	9.0769	9.1915
perl+wrf	1.0143	1.3984	1.4626
cactus+leslie	1.0034	21.2749	21.3736
gobmk+astar	0.9994	1.1053	1.1469
zeusmp +gromacs	1.0035	5.6352	5.5924
average	1.0113	7.2630	7.5077
x264	1.0255	0.7461	0.8656
raytrace	0.9974	0.0559	0.0565
blackscholes	1.0013	0.0461	0.0506
fluidanimate	1.0009	0.1312	0.1587
swaptions	0.9979	0.005	0.0053
lu_ncb	1.0246	2.2274	2.7026
fft	0.9957	3.1995	3.1992
lu_cb	1.0006	0.9077	0.9169
average	1.0046	0.9149	0.9944

5.2 LLC Size Sensitivity Analysis

To analyze the sensitivity of our design to cache size, we evaluate the performance overhead with different LLC sizes for the single benchmark/single core tests (Fig. 10 [20]). Since the bigger caches are expected to have lower eviction rates for the same workload, there are effectively fewer first accesses, resulting in a smaller additional delay. Hence, we see the performance overhead in bigger caches to be smaller. Our analysis with 2MB, 4MB, and 8MB LLC sizes shows an average performance overhead of 1.13%, 0.4%, and 0.1%, respectively. With the increasing size of the last-level cache, the baseline MPKI reduces as the cache can retain a larger fraction of the working set memory [11], resulting in fewer *first access misses* after a context switch. These numbers indicate that the defense scales well with larger caches.

5.3 Space Overhead, Timestamp Rollover, and Scaling

The increase in area due to the additional hardware is primarily due to the separate SRAM array of timestamps and *s-bits*, and the comparison logic. This separate SRAM array uses 8-T rather

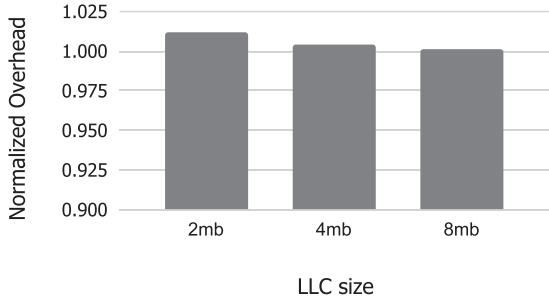


Fig. 10. Sensitivity analysis of the overhead relative to increasing last-level cache size.

then 6-T cells and also includes an additional set of sense-amps and bit-line drivers. The other components required are the timestamp comparison logic at each bit-line peripheral, consisting of 2 latches, 2 and gates, and a shift register to hold Ts.

In our evaluation, we use 32-bit *Tc* timestamps to keep the area overhead low. The number of bits used for the timestamp counter has an impact on the frequency of timestamp rollover and is a parameter that can be tuned by the chip maker. A timestamp rollover can result in an additional miss after 2^{32} cycles depending on the *Ts* of the process. We illustrate the correctness of operation using 2 decimal digits of precision for *Tc*, resulting in a rollover every 100 cycles for the purpose of illustration.

1) *Processes that preempt before and resume after rollover:* The rollover is detected by comparing *Ts* and time at resumption (e.g., 98 and 105). Since there can be newer unaccessed cache lines with rolled over (smaller) *Tc* (e.g., 1(03)), we reset all *s-bits* when rollover is detected after a process resumes.
2) *Processes that are running during a rollover:* No action is needed while the process is running as the s-bits are up-to-date.
3) *Assuming no rollover between* Ts*and time at resumption (e.g., 102 and 105):* When the process resumes, since there can be older cache lines with bigger *Tc* (e.g., 78), unnecessary resets may occur, but correctness of operation is maintained.

Thus, cache line timestamp rollover can result in additional misses but retains correctness of the defense.

An *s-bit* is required per hardware context that shares the cache for each cache line. The total number of *s-bits* can be significant for the LLC in server-class processors. Coherence directories have a similar scalability concern with a large number of cores, as they store availability information for each core. In order to keep the number of *s-bits* low, design principles used for coherence directories could be applied, for example, limited pointers [2] or a level of indirection as in SPACE [45]. For example, the limited pointer [2] directory design work demonstrated empirically that applications typically share data across a few processors. Since pointers require $\log(n)$ bits (for n hardware contexts), keeping track of a limited number of sharers would reduce area overhead to $O(\log(n))$ as opposed to n bits per cache line.

5.4 S-Bits Save and Restore Overhead

When a process is resumed, the *s-bits* and the *Ts* that were saved for the process at the time of preemption

must be restored. The overhead due to copying the *s-bits* is low for small cache sizes. The entire *s-bit* array for an L1 cache of size 64KB can be copied in 2 64-byte cache-line-size memory accesses. The overhead scales with the size of the cache. The copy can take 256 cache-line-size transfers for a last-level cache of size 8MB. The *s-bits* can be read and written in parallel via the 'regular' bit-line interface when a save or restore is required at a context switch. The save and restore is done to and from a kernel memory region reserved for the *s-bits*, to which the process context points.

On an Intel i7-7700 processor operating at 3.6Ghz, the time to copy *s-bits* for an 8MB size cache *without* caching is 2.4μ s. This is of comparable magnitude to a null context switch or system call. A typical process time slice varies from 1 ms to several ms, so the 2.4 μs overhead is at most 0.24% of the process run-time. An extra layer of buffering in hardware could allow the copy to be performed in parallel with the execution of the next process.

The save and restore of s-bits can also be done using a DMA transfer. We calculate the latency of transferring a buffer size equivalent to the one required for our simulation system. The time taken to save and restore a caching context on a Xeon processor using a single DMA channel is 1.08 μs. We add this delay to each context switch in our simulation system to account for the overhead due to the *s-bit* book keeping.

6 Security Analysis

We evaluate TimeCache against reuse-based attacks and against the attacks depending on coherence state of the shared memory.

6.1 Microbenchmark Evaluation

A reuse attack in TimeCache is prevented by delaying the *first access*. The attack sequence when carried out on TimeCache is as follows:

- The attacker evicts a shared location.
- It waits for the victim's execution, potentially caching the data evicted by the attacker.
- The attacker accessed the priorly evicted data, expecting to experience a hit in the cache, but since it is the attacker's *first access*, the *s-bit* is not set, the access results in a miss even when the data resides the cache.

To ensure the correctness of operation, we create microbenchmark attacks with parent and child processes as the attacker and the victim. The processes access shared memory mapped region with 256 cache lines. The attacker or the parent process continuously times its accesses to the shared cache lines and flushes them. The child process performs continuous read operation on the shared lines. The attack is run on the gem5 simulator, and the access times are recorded in the user space application using `rdtsc` across the access. A hit ($\sim$14 cycles) in the parent process due to the child's accesses indicates a successful attack. The attack is prevented by our defense simulation, i.e., the parent process accesses do not result in cache hit due to the victim.

```
if parent
  read shrd_mem; // cache hit
  flush shrd_mem;
  sched_yield();
else
  read shrd_mem;
```

6.2 Attacking RSA

We implement the `flush+reload` reuse attack on GnuPG shared library for RSA encyption, as discussed in the original paper [41]. The attacker and victim are independent programs running on the same machine, hence share the caches. The attack is tested both on real machine and the simulator.

We install a non-stripped GnuPG library on a real machine and locate the offsets for Square, Multiply, and Reduce functions used by the encryption. The shared library implements encryption using the chain of functions as Square-Reduce-Multiply-Reduce for processing '1' and Square-Reduce for processing a '0' bit. This is a typical example where the access to shared library is dependent on process specific secret data, i.e., the encryption routines are accessed based on process specific key bits.

The original attack proceeds by determining the timing threshold for a hit on the system. The attacker process flushes the cache and then accesses an address from the Square, Multiply, and Reduce functions in a loop. An independent victim process performs encryption simultaneously. Depending on whether the attacker experiences a hit or a miss on accesses to the encryption subroutines, it learns whether a '1' or a '0' is being proccesssed for the victim's encryption. This essentially allows the attacker to learn the key bits over some repetitions. We simplify the attack simulation and assume a hit in the attacker process due o victim's encryption to be a successful attack.

We calculate the cache access latency on both the real machine ($\sim$40 cycles for LLC) and the simulator ($\sim$14 cycles for shared L1), and use their respective calculated access latencies as the threshold for a cache hit. On the real machine, the attacker and victim run on separate cores and share the LLC, whereas The attack on the simulator is run on a single core, i.e the attacker and the victim share the L1 cache. The attacker and the victim are independent user space processes, with access to a shared memory region. The attacker process repeatedly times its access to a flushed memory region, and is able to incur hits due to the accesses from a simultaneously running victim. The attack goes through both on the real machine and on the gem5 simulator in full-system simulation mode.

Our defense simulation prevents accesses with cache hit latency ($\sim$14 cycles) in the attacker process, by identifying its timed accessed as its *first access*. It allows a cache access latency only on data that has been accessed earlier with a memory access latency. The attacker perceives a miss ($\sim$185 cycles) for its *first access* and cannot infer if the data is cached by the victim or not.

6.3 Security Evaluation of Coherence Attack

We implement the original exploit explained in the paper describing the side channel attack from coherence state

leak [39]. The attacker threads share memory and run on cores C1 and C2. One of the threads repeatedly times its accesses to the shared memory and flushes the accessed data, while the other accesses the shared memory and keeps it *exclusive* in the local cache. We evaluate the attack on the gem5 simulator using a snoopy bus coherence protocol. To simulate *E* access latency, we allow the cache with *exclusive* data to respond to data requests. Thus the attacker thread on C1 receives the requested shared data with *E* access latency. Since a response from another cache on the same level takes lesser number of cycles in a snoopy protocol, the *E* access latency is our implementation is lesser (20 cycles) than the LLC response latency (40 cycles) for data in *S* state.

Accesses from the victim thread to the same shared memory running on C3 changes the *exclusive* state of the data in the local cache of C2 to *shared* state. Further timed accesses from C1 experience *S* access latency, as the request is serviced from the LLC. Our defense works against the attack by disallowing *S* access latency on previously unaccessed data (refer Section 4), the attacker experiences memory access latency ($\sim$185 cycles) instead.

6.4 S-Bits do not Introduce Additional Side Channels

The additional hardware comprising of the SRAM comparator array and the execution context specific *s-bits* do not create additional side channels for the following reasons:

- The operations of a process executing on a hardware context is affected only by the *s-bits* associated with that hardware context, and not by the *s-bits* for any other context.
- The execution context's *s-bits* are saved and restored during a context switch so that a process does not see *s-bits* from another process that might have used the same hardware execution context.
- The *s-bits* are managed only by a trusted computing base at the time of the context switch.
- The *s-bit* save and restore is a constant time operation, thus leaks no information about bits being set or reset.

7 Related Work

The existing defense techniques to prevent side channel attacks on shared memory either from reuse or coherence state leak, resort to cache partitioning or obfuscate time, and incur significant overhead.

7.1 Cache Partitioning

Side channels in shared cache can be prevented by partitioning the cache, or disallowing sharing. However, partitioning impacts performance by reducing the cache available for execution, and generally use security domains. Dynamic partitioning can resize the partitions as required and can achieve lower overheads compared to static partitions. SecDCP [35] implements a dynamic cache partitioning between two groups of confidential and public applications. It provides coarse grain protection and works with a prior knowledge of security domains. Another dynamic cache partitioning technique utilizes page coloring to allocate pages to a secure domain [31], [43], but can incur significant copy costs for recoloring. DAWG uses security domains and partitions cache ways between a maximum of 16 security domains at a time, incurring 4-12% slowdown [13]. PLCache locks a cache line for a process, thus can be seen as a line wise partitioning. It results in a 12% performance degradation.

7.2 Last Level Cache Defense

7.2.1 Intel CAT-Based Partitioning

Intel's cache allocation technology (CAT) allows way wise partitioning of the last level cache. Both Catalyst [17] and Apparition [5] use Intel CAT to partition cache between distrusting users for cache side channel defense. However, the performance of such systems depends on its ability to minimize reassignment of partitions to other applications, keeping the cache flushes to a minimum. For instance, Apparition [5] assigns a Class of Service (CLOS) to each requesting application and flushes the entire partition across context switches, resulting in very high overhead for certain applications. Catalyst uses pinned pages and the solution is suited to cloud service providers. Both these techniques are dependent on hardware support available only at the LLC and hence cannot defend against attacks at other cache levels.

7.2.2 FTM

First Time Miss (FTM) [25] uses directory presence bits to prevent reuse of data in LLC, over data brought in by other cores. The defense only works for a cache that is shared by the attacker and the victim running on seperate cores, and is managed using the coherence directory. Not all use cases support distinct cores for attacker and victim, as identifying the attacker is necessary for separating its execution environment, nor is it necessary to have coherence directory bits.

The threat model in Timecache is stronger and hence provides a flexible and scalable defense. The defense does not depend on identifying or isolating the attacker processes. TimeCache protects against attacks at all levels of cache, including the higher-level caches. Recognizing a first-time miss in the presence of directories, and in the absence of hyperthread or context switches is straightforward. Doing so in a shared higher-level cache, in the presence of context switches is enabled by the use of time to recognize *first-accesses*, and by our novel bit-serial, timestamp parallel comparator. TimeCache can also mitigate cross-process Spectre variants, by eliminating the reuse side channel used by Spectre.

7.3 Obfuscating Time & Constant Time Algorithm

The ability to time accesses from user space without any privilege can also be seen as the vulnerability that enables the attacker to launch timing side channel attacks. Removing this ability has been suggested as a defense mechanism in the past. Some techniques suggest making timing instructions privileged, or returning approximate time to defeat the attack. Both these techniques can be rendered ineffective as a defense, by using alternate timing primitives for recovering clock with a fine resolution [28].

As timing side channel reveal process specific secrets when the access to the shared memory is dependent on those process specific secrets, accessing the shared region

with non secret data can add noise enough to prevent the attack. This technique is used in constant time implementations of algorithms [16], [26], [27], to defend against timing side channels. Since the transformed program adds many more accesses to the memory, the associated overhead is impractical for defending accesses to large shared libraries.

7.4 Prior Defense Against Attacks Due to Coherence State Leak

DAWG replicates shared memory in partitions, and requests to the same line from different domain IDs are filled by the memory controller, there by not allowing coherence state leak across security domains [13]. Another suggested defense is to make the LLC the owner or responder rather than the cache with E or M, which requires sending all the updates to LLC [40]. TimeCache does not require this additional update to the LLC.

8 Conclusion

We implement and evaluate a defense against side channel attacks resulting from reuse or coherence state leak of shared memory. TimeCache provides entire cache access to all executing applications. The design does not constrain system configuration or use, i.e., it works against an attacker running simultaneously with the victim or preemptively, either cross-core or on the same core.

TimeCache defends against reuse attacks on shared data by maintaining per-process caching contexts. TimeCache creates, saves, and restores caching contexts to retain the speed of operation over shared memory across context switches. A novel bit-serial timestamp comparator is used to update the stale caching context upon a context switch by comparing against per-cache-line last-fill timestamps. We evaluate the security of the defense against microbenchmark attacks using the classic `flush+reload` attack and using an exploit that uses coherence state leak in the gem5 simulator. The performance overhead is evaluated using standard benchmarks from SPEC2006, PARSEC, and SPLASH. The average slowdown on SPEC2006 is 1.13%, most of which is from delayed first accesses, with caching context book-keeping adding 0.024%. Multicore simulation of PARSEC and SPLASH shows an average overhead of 0.46%. Our defense retains the benefits of shared memory like memory pressure reduction, and allows safe deduplication.

Acknowledgments

We also thank Sreepathi Pai for his feedback during the initial discussions of the ideas in this work.

References

[1] Kernel samepage merging (memory deduplication), 2017. [Online]. Available: https://kernelnewbies.org/Linux_2_6_32#Kernel_Samepage_Merging_.28memory_deduplication.29
[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Proc. Int. Symp. Comput. Archit.*, 1988, pp. 280–289.
[3] N. Binkert et al., "The GEM5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[4] C. Canella et al., "Fallout: Leaking data on meltdown-resistant cpus," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 769–784.
[5] X. Dong, Z. Shen, J. Criswell, A. L. Cox, and S. Dwarkadas, "Shielding software from privileged side-channel attacks," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1441–1458.
[6] C. Eckert et al., "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 383–396.
[7] 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*, 2016, pp. 279–299.
[8] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 897–912.
[9] 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.
[10] W.-M. Hu, "Lattice scheduling and covert channels," in *Proc. IEEE Comput. Soc. Symp. Res. Secur. Privacy*, 1992, pp. 52–61.
[11] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," 2010. [Online]. Available: http://www.glue.umd.edu/ajaleel/workload
[12] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proc. Israeli Exp. Syst. Conf.*, 2009, pp. 1–12.
[13] 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. Microarchit.*, 2018, pp. 974–987.
[14] P. Kocher et al., "Spectre attacks: Exploiting speculative execution," *Commun. ACM*, vol. 63, no. 7, pp. 93–101, 2018.
[15] 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 Workshop Offensive Technol.*, 2018, Art. no. 3.
[16] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "GhostRider: A hardware-software system for memory trace oblivious computation," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 87–101, 2015.
[17] F. Liu et al., "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 406–418.
[18] 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.
[19] M. Mushtaq, M. A. Mukhtar, V. Lapotre, M. K. Bhatti, and G. Gogniat, "Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA," *Inf. Syst.*, vol. 92, 2020, Art. no. 101524.
[20] 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. Archit.*, 2021, pp. 375–387.
[21] 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.
[22] D. Page, "Partitioned cache architecture as a ėide-channel defence mechanism," 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.460.9926&rep=rep1&type=pdf
[23] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proc. IEEE/ACM 51st Annu. Int. Symp. Microarchit.*, 2018, pp. 775–787.
[24] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proc. ACM/IEEE 46th Int. Symp. Comput. Archit.*, 2019, pp. 360–371.
[25] K. Ramkrishnan, S. McCamant, P. C. Yew, and A. Zhai, "First time miss: Low overhead mitigation for shared memory cache side channels," in *Proc. 49th Int. Conf. Parallel Process.*, 2020, pp. 1–11.
[26] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 431–446.
[27] A. Rane, C. Lin, and M. Tiwari, "Secure, precise, and fast floating-point operations on x86 processors," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 71–86.
[28] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2017, pp. 247–267.
[29] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read arbitrary memory over network," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2019, pp. 279–299.

[30] P. Sharma and P. Kulkarni, "Singleton: System-wide page deduplication in virtual environments," in *Proc. 21st Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2012, pp. 15–26.
[31] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Proc. IEEE/IFIP 41st Int. Conf. Dependable Syst. Netw. Workshops*, 2011, pp. 194–199.
[32] C. Trippel, D. Lustig, and M. Martonosi, "CheckMate: Automated synthesis of hardware exploits and security litmus tests," in *Proc. IEEE/ACM 51st Annu. Int. Symp. Microarchit.*, 2018, pp. 947–960.
[33] S. van Schaik et al., "RIDL: Rogue in-flight data load," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 88–105.
[34] D. Wang et al., "Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries," in *Proc. 26th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2019.
[35] 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. 53rd Annu. Des. Autom. Conf.*, 2016, Art. no. 74.
[36] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 494–505, 2007.
[37] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "SCATTERCACHE: Thwarting cache attacks via cache set randomization," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 675–692.
[38] 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. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 347–360.
[39] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 168–179.
[40] 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.
[41] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 719–732.
[42] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 503–516, 2015.
[43] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, 2009, pp. 89–102.
[44] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 990–1003.
[45] H. Zhao, A. Shriraman, and S. Dwarkadas, "SPACE: Sharing pattern-based directory coherence for multicore scalability," in *Proc. Int. Symp. Parallel Architectures Compilation Techn.*, 2010, pp. 135–146.

Divya Ojha received the bachelor's degree in electronics and communication. She is currently working toward the PhD degree in computer science with the University of Rochester, working with Dr. Sandhya Dwarkadas.

Sandhya Dwarkadas (Fellow, IEEE) is currently the Walter N. Munster professor and the chair of computer science with University of Virginia, Charlottesville, VA, USA. Until July, she was the Albert Arendt Hopeman professor of engineering and a professor of computer science with the University of Rochester, Rochester, NY, USA. Her research has contributed to the field of shared memory and re-configurable computing.