2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00037

TimeCache: Using Time to Eliminate Cache Side Channels when Sharing Software

Divya Ojha
Dept. of Computer Science
University of Rochester
Rochester NY, USA
dojha@cs.rochester.edu

Sandhya Dwarkadas
Dept. of Computer Science
University of Rochester
Rochester NY, USA
sandhya@cs.rochester.edu

***Abstract*—Timing side channels have been used to extract cryptographic keys and sensitive documents even from trusted enclaves. Specifically, cache side channels created by reuse of shared code or data in the memory hierarchy have been exploited by several known attacks, e.g., evict+reload for recovering an RSA key and Spectre variants for leaking speculatively loaded data.**

In this paper, we present TimeCache, a cache design that incorporates knowledge of prior cache line access to eliminate cache side channels due to reuse of shared software (code and data). Our goal is to retain the benefits of a shared cache of allowing each process access to the entire cache and of cache occupancy by a single copy of shared software. We achieve our goal by implementing per-process cache line visibility so that the processes do not benefit from cached data brought in by another process until they have incurred a corresponding miss penalty. Our design achieves low overhead by using a novel combination of timestamps and a hardware design to allow efficient parallel comparisons of the timestamps. The solution works at all the cache levels without the need to limit the number of security domains, and defends against an attacker process running on the same core, on a another hyperthread, or on another core.

Our implementation in the gem5 simulator demonstrates that the system is able to defend against RSA key extraction. We evaluate performance using SPEC2006 and PARSEC and observe the overhead of TimeCache to be 1.13% on average. Delay due to *first access misses* adds the majority of the overhead, with the security context bookkeeping incurred at the time of a context switch contributing 0.02% of the 1.13%.

I. Introduction

Shared memory resources expose timing side channels that can reveal information even in the presence of security measures such as process isolation and enclave separation. Cache side channels leveraging shared memory have been shown capable of extracting cryptographic keys, sensitive documents, and data even from cryptographically secured enclaves [6]. Several classes of cache side channel attacks and defenses have been developed in the literature [24].

In this paper, we focus on cache side channels created by the reuse of shared software (code and data) in the memory hierarchy. Shared software is an essential component to keeping system costs low. For instance, shared libraries (code) are an important optimization in modern computing systems to help keep the memory footprint low. Likewise, services providing access to large data stores result in data being shared across untrusted client requests. Access to the shared code or data leaves a footprint in the memory hierarchy, which has been exploited by several known attacks [49] [11] [23] [46] [9] [6] [18].

A typical cache side channel attack when sharing software involves evicting the shared data (e.g., code from a shared library) from the cache hierarchy and re-accessing it after the victim's execution (using evict+reload or flush+reload [46]). A fast re-access is indicative of an access to the shared location by the victim. If the shared library access is indexed by a secret data, the attacker can infer the victim's secret. This attack model is used in attacks to leak cryptographic keys [46], in Spectre I, Spectre II [18], NetSpectre [33], in cross-tenant attacks to leak data in clouds providing Platform-as-a-service [49], and in discovering key strokes [38].

There is another class of cache side channel attacks that do not require shared memory and is not the focus of this work. These attacks are referred to as contention or conflict-based side channel attacks. Contention attacks are mitigated using randomizing caches as in CEASER [27], CEASER-S [28], and ScatterCache [13], or using very efficient multiple hashing techniques like RPCache [40]. However, these techniques are unable to prevent reuse attacks in shared memory. TimeCache can work in conjunction with these techniques to provide a holistic defense.

Reuse attacks on shared memory are more precise and a handy tool for constructing more sophisticated attacks. They are less noisy and are a preferred covert channel for leaking speculatively loaded data [5], [18], [37]. Preventing reuse attacks on shared memory will not only make the attacker's work difficult but also allow system providers to deploy deduplication or copy-on-write sharing (e.g., unix-style process fork operations or Docker-style containers) for increased performance and reduced space utilization. Deduplication evaluation in the literature shows that its use can reduce memory needs by a factor of 2-4x [15], [34] and increase performance by up to 40% [34].

Existing solutions for reuse attacks partition the cache [7], [17], [26], [39], [40] or implement constant time algorithms [20], [30], [31], both resulting in increased latency. Partitioning has been seen to be associated with higher overheads due to both reduction in the effective cache size for individual processes, and due to potential aliasing of the shared memory in the cache, depending on the system design.

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00037

Partitioning techniques also might have restrictions on the number of supported security domains [17], [40]. For instance, DAWG [17] supports only 16 security domains at a time. Some other solutions protect accesses only to the LLC [22], [44].

In this work, we design and evaluate a low-overhead hardware-software solution to defend against reuse attacks. Our goal is to retain the benefits of a shared cache of allowing each process access to the entire cache and of cache occupancy by a single copy of shared software. We protect every level of cache without limiting the number of supported security domains.

TimeCache creates a "per-process view" of cache occupancy by delaying (treating as a miss) the *first access* by any process to a resident (i.e., a cache hit) cache line. Delaying the first access gives each process timing isolation from other processes when sharing data by giving every process the impression that data is brought into the cache by its own access. This approach breaks the fundamental premise of a reuse attack using shared software. The reduction in performance due to the delay can be considered elemental to the design of a secure cache while avoiding a potential $O(n)$ space consumption for n processes sharing data in a partitioned cache. Delay is incurred only when data is evicted and reloaded, so that the performance of steady-state in-cache sharing is unaffected. As a consequence of this defense, systems can choose to deploy memory deduplication techniques to reduce memory footprint [1], [15] without the fear of creating an avenue for cache side channels through a shared software stack.

We implement our defense using a novel combination of timestamps and a hardware design to allow efficient parallel comparisons of the timestamps, coupled with per-cache-line access bits and software support at context switches. In software, a process's caching context is saved along with the "context-switch" timestamp when it is context switched out. Hardware-implemented *bit-serial* comparison logic allows fast parallel timestamp comparisons for the context being restored and is used to update the stale caching context being restored.

The security and performance of TimeCache on the gem5 simulator [4] demonstrates its effectiveness against attacks using microbenchmarks and an RSA attack. Our defense is able to prevent the classic RSA attack used to demonstrate `flush+reload` attacks. Performance evaluation using SPEC2006 and PARSEC shows an average overhead of 1.13% and 0.8%, most of which is due to delayed accesses from shared software. The overhead due to the security context bookkeeping at context switches adds about 0.024%.

The key contributions made by the paper include:

- Preventing reuse attacks on shared software while allowing access to the entire cache by each process and maintaining a single cached copy of the shared software.
- Disallowing the *first access* by a process to a cache line from experiencing a cache hit when the cache line has been brought into the cache by a different process.
- Proposing a timestamp-based solution to creating a per-process view of cache line occupancy across context switches to prevent reuse attacks.
- Developing a fast *bit-serial* timestamp-parallel comparison logic to compare timestamps for all cache lines simultaneously.
- Using a simulation-based evaluation to demonstrate that our proposed solution prevents real-world attacks and analyzing the potential overheads of timing isolation.

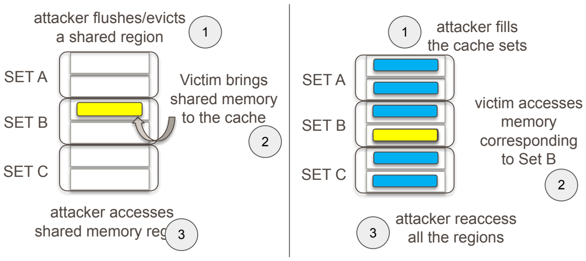
II. Background

A. Cache Side Channels

Information leaked as a result of shared cache utilization is collectively referred to as *cache side channels*. Mechanisms to exploit cache side channels were first exposed as early as in 1992 [12], and different classes of attacks relying on cache access timing have been developed since then.

Two types of information leak are possible depending on whether or not there is shared software between the attacker and the victim. With no shared software between the attacker and the victim, the attacker can only learn the cache set accessed by the victim using a "Prime+Probe" [25] style of attack, commonly referred to as contention-based attack. In the presence of shared software, an attacker can learn the line accessed by the victim using an "evict+reload" or "flush+reload" style attack [10], [46]. Figure 1 depicts both the attacks. Defenses against "prime+probe" attacks such as caches using randomized placement [27] [21] are not effective against "evict+reload" or "flush+reload" style attacks. The latter is a low-noise, high-bandwidth, and more efficient form of attack. This work addresses this second style of attacks.

Fig. 1: Reuse and contention attacks


attacker flushes/evicts a shared region
1
SET A
SET B
SET C
Victim brings shared memory to the cache
2
attacker accesses shared memory reg
3
1
attacker fills the cache sets
SET A
SET B
SET C
victim accesses memory corresponding to Set B
2
3
attacker reaccess all the regions

B. Shared Software Attacks

Shared libraries have commonly used subroutines, which can be mapped directly into a process's user-level address space. The same physical memory may be mapped into different processes' virtual address spaces. Shared libraries help reduce memory footprint and improve memory hierarchy efficiency. However, they create the potential for leaked memory access patterns through cache side channels, whether due to access to the shared code or to shared data.

Side channels exist due to shared hardware and software, and improve in precision in the presence of shared software. Side channels using shared software were earlier assumed to affect only cryptographic routines. The advent of more recent attacks have shown that they are a handy gadget for more sophisticated attacks like Spectre [18], [19], [33]. They are also capable of leaking keystrokes from another process [38], leaking passwords in cloud environments such as an Amazon EC2 server, and leaking data across Virtual Machines [49].

C. Prior Solutions

There have been a number of defenses around cache side channels but the strict performance requirement of caching structures has remained a challenge. One class of defenses mitigate only contention-based attacks (e.g., randomizing caches [27], [28], [41], SHARP [44], RPCache [40]). Defenses that include reuse attacks either resort to cache partitioning [7], [17], [40] or implement constant time algorithms [20], [30], [31]. Partitioning defends against both reuse and contention attacks but reduces the effective cache available for each process execution, effectively reducing performance. Constant time algorithms, likewise, incur a significant performance penalty [20], [30], [31]. Some defenses restrict the number of possible security domains [17], [40]. Others work only for the last-level cache [24], [29]. FTM [29] uses directory presence bits at the last-level cache to detect first accesses, requiring untrusted processes to be spatially isolated on separate physical cores.

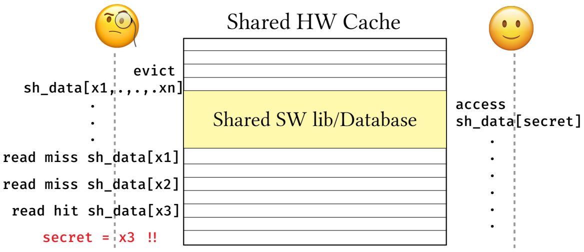
In terms of detecting the presence of side channels, hardware description languages that check for information flow in the hardware specification [47] help to detect the existence of side channels and help specify security labels in the processor for building side-channel-resistant processors. Likewise, Checkmate uses relational model finding to detect the presence of side channels in processor specification [36].

III. Threat Model

The threat model under consideration has a separate attacker and victim process sharing some software stack in addition to sharing caches. They could be running simultaneously on the same (hyperthreaded) or different cores, or interleaved in time, and the attack can be conducted from any level of the cache. Figure 2 shows a reuse attack when accessing the shared software stack in a cache shared between attacker and victim. The attack has the following sequence:

1) The attacker and victim share software and a hardware cache. The access to the shared software is dependent on or is indexed using the victim's secret data.
2) The attacker evicts a shared location from the cache hierarchy.
3) It then waits for the victim's execution.
4) The attacker subsequently reloads the same shared location and determines that the shared location was also accessed by the victim if it hits in the cache, determined by timing the access.

Fig. 2: Reuse attack in cache



This attack model is self-sufficient in the sense that it has been demonstrated to be capable of leaking RSA keys when using the GnuPG shared library [11], [46]. It is also a low-noise, high-bandwidth tool for building more sophisticated attacks like Spectre-I & II [18], SpectreRSB [19], and Netspectre [33]. It is a preferred covert channel for these more recent attacks for leaking shared library access patterns. Other contention-based covert channels could also be used for a similar purpose but are not as precise and hence would make the attack more difficult.

IV. Per-Process Caching Context

TimeCache eliminates reuse-based cache timing side channels by implementing techniques to allow per-process cache line visibility of a shared cache line. Unlike partitioning-based approaches, TimeCache does not use isolation in space; rather, TimeCache ensures that accesses to a shared cache line by different processes are isolated in timing. This allows different processes to share access to the same cache line without revealing to one another if the cache line was made available in the cache by another process. Compared to solutions that rely on cache partitioning (example, Intel's cache allocation technology [22] in [7]), this approach does not restrict the size of usable cache for any process. It is also applicable to any level of the cache hierarchy from L1 to LLC.

A. First Access

A process's *first access* refers to the first time it accesses a resident cache line that was brought into the cache by another process. A resident cache line can experience as many *first access* misses as the number of processes accessing it (minus one for the initial cache line fill). If a cache line is evicted and later brought back into the cache by a process, all other processes accessing the cache line at a later time will experience a *first access* miss.

The importance of the *first access* lies in the construct of the attack. If the attacker times its *first access* after evicting a data from the cache hierarchy and is able to detect a cache hit, the attacker is able to infer the victim's memory access patterns. Beyond the *first access*, a fast access or a cache hit does not provide any clue about the data access pattern of another process. Using this key observation, TimeCache enables space sharing (avoids partitioning) by enforcing misses on *first access* to provide timing isolation.

In the baseline cache design, processes experience delays in access due to cold, capacity, conflict, and coherence misses, whether due to its own actions or due to those of other processes. Critically, it may also experience hits in the cache due to data brought in because of another process's access. It is this latter timing side channel that TimeCache targets. In TimeCache, we create a new kind of miss: a *first access* miss.

B. Distinguishing First Accesses

TimeCache is based on the observation that the attack under consideration exploits the caching benefits due to another process. Hence, we propose to identify when a process first accesses a resident cache line and to treat the access as a miss. By treating this *first access* as a miss (essentially incurring the delay of a miss), an attacker will be unable to infer another process's memory access patterns via cache residency.

Whether a cache line has already been accessed by the currently executing context is represented by a per-hardware-context security bit (*s-bit*). When a cache line is brought into the cache on a miss, the *s-bit* for the loading context is set and the *s-bits* for all other hardware contexts are reset. On a cache hit, the *s-bit* of the accessing context is checked. When the *s-bit* is set, the access is allowed to proceed as a hit. Otherwise, the access is recognized as a *first access*, treated as a miss, and the context's *s-bit* is set so that the future accesses can proceed as a hit (see Figure 4(b)). *First access* misses are handled by sending the request down the memory hierarchy but not filling the cache with the received data, as the data in the cache is the most recent copy. This mechanism is implemented at every level of cache in the memory hierarchy.

C. Handling Context Switches

s-bits represent the caching context (footprint) of a hardware execution context and are specific to the process executing in the context. At the time of a context switch, *s-bits* must be carefully managed in order to ensure that they are neither stale nor contain information on the caching context of a different process.

TimeCache ensures timing isolation at the time of a context switch using a combination of software and hardware. In order to retain caching behavior across context switches and still provide timing isolation, software saves the *s-bits* for the process being preempted, along with current time as its "context-switch" timestamp (*Ts*). Software also restores the *s-bits* for the process being scheduled (saved from the last time the process executed), to the cache in the corresponding hardware context where the process is set to resume execution.

Hardware ensures that the stale restored *s-bits* (reflecting the state of the cache when the process last executed) are updated to reflect the current cache content. The s-bit for a cache line that has been filled after time *Ts* must be reset, detected by maintaining (and comparing *Ts* against) *Tc*, the time at which the cache line was last filled.

The *s-bit* save and restore can be performed by any trusted computing base library at the time of context switch. In our design, we allow the operating system to save and restore the process-specific *s-bits*. The ability to save, restore, and update the caching context allows TimeCache to enjoy fast access as long as the data is not evicted from the cache. Our design leverages locality across context switches while providing timing isolation, something that cannot be achieved by simply flushing the cache on a context switch. The mechanism described here may be designed as a processor feature that can be turned off if the processes are trusted and reuse-based cache attacks are not a concern.

V. TimeCache: Design and Implementation

Figure 3 provides an overview of the hardware modifications, depicted for a cache consisting of 8 cache lines accessed by two hardware contexts. The hardware support added to a conventional cache is as follows:

- A per cache line, per hardware context, security bit (*s-bit*).
- A per cache line timestamp Tc to indicate the time at which the line was filled (became resident).
- A bit-serial, timestamp-parallel comparison logic with transpose gate and bitline peripherals, to compare timestamps efficiently.
- A shift register to hold Ts, the timestamp indicating the time when a process that is about to resume execution due to a context switch, last executed.

The following subsections describe the implementation details of each hardware modification and the software support required for the defense.

A. First Access Delay Mechanism

On a traditional cache access, the requested data is returned to the processor if a tag and state lookup succeeds. Otherwise, the access incurs a miss and the request is passed on to the next level in the memory hierarchy. With our cache design, the *s-bit* for the cache line is checked in addition to the state and tag bits. An access is considered a hit only if in addition to the above, the *s-bit* of the cache line is set, in which case data is returned to the processor from the cache.

The flowchart in Figure 4 represents the actions taken by TimeCache to maintain timing isolation for a process during

Fig. 3: TimeCache hardware, depicted for a cache with 8 lines accessed by a core with 2 hardware contexts

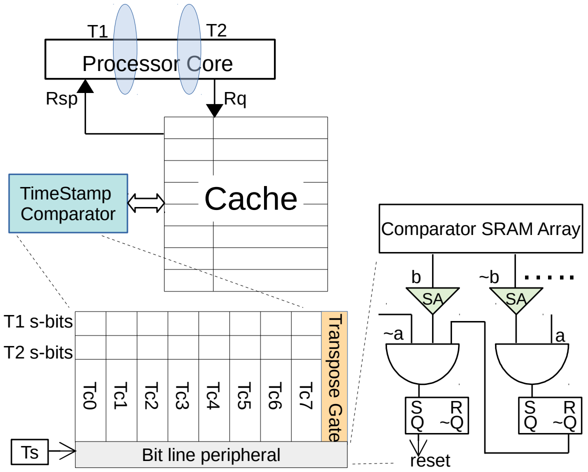
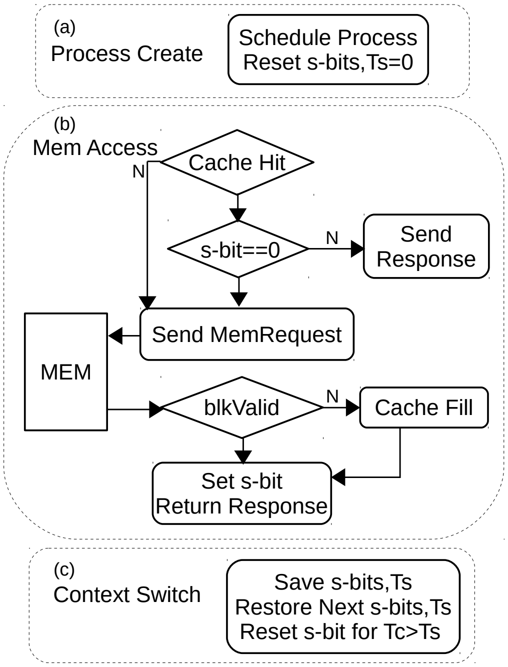


Fig. 4: Maintaining timing isolation: per-process flowchart



process creation, execution, and preemption/resumption at the time of a context switch. Software saves and restores the *s-bits* for a process executing on a hardware context to/from memory at the time of a context switch. Additionally, software maintains *Ts* for each process, which is the time the process was most recently preempted. A newly created process has both *Ts* and *s-bits* reset when it is scheduled for the first time.

Per cache line *s-bits* are modified by the following actions:

- Restored from memory for the hardware context on which a process is scheduled/resumed.
- Reset by the timestamp comparison logic for the hardware context on which a process is being scheduled/resumed.
- Reset when a cache line is evicted or invalidated.
- Set for the requesting hardware context when a cache line is filled; reset on all other hardware contexts.
- Set for the requesting hardware context after a first access to a resident cache line.

A reset *s-bit* on a cache hit indicates that the current process has not accessed the cache line. If the *s-bit* is reset, the response to the processor is delayed by sending the request down the memory hierarchy. Once the response is received, the received data is discarded, and the data in the cache line is forwarded to the processor. The *s-bit* is set to ensure that future accesses to the cache line by the process do not result in additional traffic and are treated without additional delay.

The rationale behind sending a request down the memory hierarchy even when the data is available in the cache is to make the *first access* see a response latency equivalent to the variable access latency it would have incurred on a miss. It is possible that a context's *s-bit* is reset in a higher-level (closer to the processor) cache but set in a lower-level cache due to its larger capacity. Sending a request down the memory hierarchy ensures that if the requested data is available in a lower-level cache and has the *s-bit* set, the request is serviced with the lower cache response latency. The data received in the response is, however, discarded, as the cache has the most recent copy of the data.

When a cache line is evicted or invalidated, all *s-bits* are reset. When a cache line is filled, the *s-bit* for the hardware context loading the line is set; the *s-bits* for all other hardware contexts sharing the cache remain reset.

On a context switch, hardware compares *Tc* for each cache line against *Ts* (loaded into a special register by software) for the process context being resumed; the s-bits for lines that have *Tc* greater than *Ts* are reset for the hardware context on which the process is being resumed in order to enforce delayed first access.

The *s-bits* save and restore is done only at context switch time. All memory accesses thereafter proceed with an additional 1 bit lookup in parallel with the cache tag lookup. If the *s-bit* is not set, the access results in a miss. This is unlike conventional caches, where a response is sent back to the processor if the data is cached.

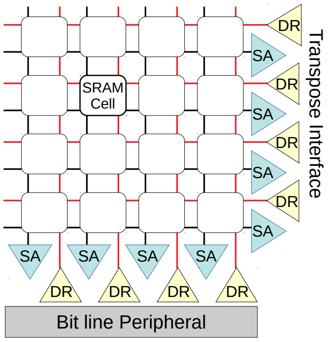
B. Per-Process s-bits Copy and Update

The *s-bits* are saved and restored on a context switch to preserve caching benefits across context switches. If the *s-bits* were not saved and instead reset on every context switch, this would be equivalent to flushing the cache on every context switch, which can impact performance heavily [7].

The number of 64-byte (cache line size) memory accesses required to save or restore *s-bits* is dependent on the cache size. A small 64KB L1 cache requires only 2 64-byte memory accesses, while a larger 8MB L3 cache requires 256 64-byte memory accesses.

Restored *s-bits* cannot be used as is since they are stale and need to be updated based on any changes in the cache. If a cache line is evicted while a process is preempted, its corresponding saved *s-bit* in memory will not be up-to-date. To update the *s-bits* for cache lines that might have been evicted, invalidated, or reloaded when the process was preempted, we use the *Ts* timestamp. *Ts* indicates the last time the *s-bits* were brought up-to-date, so any cache lines loaded after that time would not have been accessed by the process. When a process resumes execution, its restored *Ts* is compared with the *Tc* of every cache line in parallel, and the *s-bits* for all cache lines with *Tc* greater than *Ts* are reset. Timestamp comparisons are triggered *only* at the time of a context switch and prior to resuming a process. Subsequent accesses need no comparison since the *s-bits* now contain the necessary information. Comparing timestamps serially for all cache lines at the time of a context switch can consume a significant number of cycles. We discuss the comparison of *Tc* and *Ts*, and the mechanism for updating large arrays of *s-bits* in constant

Fig. 5: Transpose SRAM array for timestamps



time (proportional to the number of *Tc* bits) in the following subsection.

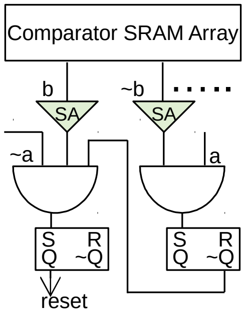
C. Bit-Serial, Timestamp-Parallel Comparison of Timestamps

A regular data access from cache is *bit-parallel*, i.e, all the bits in a cache line, along with the line's tag, timestamp, and hardware-context-specific *s-bit*, may be accessed at the same time. Accessing cache SRAM arrays in *bit-parallel* fashion implies that the time required to perform timestamp comparisons would be proportional to the number of cache lines. In order to perform parallel comparisons of cache line timestamps (Tc) and Ts, we store the per cache line *Tc* timestamps along with the cache line's *s-bits* in an SRAM array in a transposed fashion, similar to that proposed in the neural cache work [8]. The result is computation performed in a *bit-serial* [3] and word-parallel (timestamp-parallel) manner.

1) Transpose Interface: The transpose memory unit [8] uses 8-T bit cells and two sets of sense amps and drivers to access data in both regular and transposed modes. While access times will be higher compared to a 6-T SRAM cell, accesses can be made in parallel with the much larger cache data arrays. Figure 5 shows the timestamp array and comparison logic, constructed with the 8-T multi-access SRAM cells. The 'transpose interface' is used for the regular operation of the cache, which is when timestamps are updated and *s-bits* of other contexts are reset, or an *s-bit* needs to be looked up or set. The 'regular' bit-line peripheral interface is used for *s-bit* saves and restores, as well as for parallel timestamp comparisons and *s-bit* resets.

After the process-specific *s-bits* are loaded into the SRAM array in the *s-bits* for the corresponding hardware context, they need to be updated with the information about the cache lines that have been evicted while the process was preempted. This is done by comparing the *Tc* and the restored Ts. The transposed timestamps allow a *bit-serial* and timestamp-parallel comparison, taking time linear in the number of bits in the timestamp (32 in our experiments). The impact of timestamp rollover is discussed in Section VI-C. The logic required for the timestamp comparisons and reset of *s-bits* is shown in Figure 6.

Fig. 6: Bit-line peripheral



2) Bit-Serial Comparison Logic: Bit-serial computation allows us to simplify the comparison logic. The greater of two unsigned integers can be determined by comparing their bits sequentially starting from the MSB (most significant bit): one of the two numbers can be declared as larger when the first bit that differs is encountered: the larger number will have the bit set in its binary representation where the other number has the bit set to 0. We codify the above algorithm in the following scheme iterating from the MSB:

- If the bit position under consideration has a 1 for only one of the two numbers, that number can be marked as greater and the comparison is complete. This behavior can be checked by performing an `xor` of the two bits.
- If the bit position under consideration has a 0 for both the numbers, the next bit position is considered.
- If the bit position under consideration has a 1 for both the numbers, the next bit position is considered.

For instance, the greater of the two numbers '1100' and '0101' can be determined as the first number '1100' by looking at the MSB.

Ts is loaded into a shift register. For each of 32 iterations (the size of our *Tc* timestamp), the *Tc* timestamps are read from the SRAM array 1 bit at a time using the 'regular' bit-serial peripheral interface, at the same time as the shift register is shifted left to feed the comparison logic.

- If *Tc[i]* is 0 and Ts[i] is 1, $Tc < Ts$, the *s-bits* need not be updated and the comparison should stop. We latch this output and use it to ignore further bit comparisons.
- If *Tc[i]* is 1 and Ts[i] is 0, $Tc > Ts$, i.e., the cache line is newer than the Ts. The bit-line peripheral latches a '1' and the latch output is used as the reset for the *s-bit*.

Figure 6 shows the peripheral circuit attached to each SRAM bitline. It requires 2 SR latches, which are reset prior to initiating the timestamp comparisons, and 1 3-input `and` gate, and 1 2-input `and` gate for the comparison operation, with the *Tc* bit being fed to 'b' and the *Ts* bit to 'a'.

The comparison should stop if *Tc* is determined to be smaller than Ts, which is the result of the `and` gate on the right. To ignore further bit comparison, the result is latched using an S-R latch, and Q is fed to the `and` gate on the left.

At the end of the 32 iterations, if it is determined that $Tc > Ts$, as latched in the left-hand S-R latch, the bitline drivers for which the S-R latch has been set, and the wordline for the *s-bit* corresponding to the hardware context, are enabled, to write a 0 into the *s-bits*.

VI. Evaluation

We implemented TimeCache in the gem5 cycle-accurate simulator [4] 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 IV. 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 I specifies the real and simulation system parameters used for the evaluation.

TABLE I: 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

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

A. Security Analysis

The attack depends on a fast reload due to another process. The attack can be broken if no process is allowed a cache hit due to another process. If the *first access* by a process to an existing cache line is never a cache hit, the attacker remains oblivious of the data being cached beforehand and cannot learn if some shared data was accessed by another process. The second access is of no significance to the attacker. Allowing unaltered access beyond the *first access* is sufficient to ensure security while not significantly compromising performance. A reuse attack in TimeCache is prevented as follows:

- Attacker evicts a shared location
- Attacker waits for the victim process to execute, resulting in shared data being cached
- Attacker accesses the shared data but does not experience a cache hit due to the victim's caching

The additional information tracked for the defense includes timestamps and the *s-bits*, which is saved and restored by trusted software, and protected from unprivileged access.

1) Microbenchmark functionality evaluation: In order to confirm the correct operation of the timestamp-based approach, we created a microbenchmark attack consisting of a pair of child and parent processes accessing a shared memory-mapped array of size equal to 256 cache lines. The parent process acts as the attacker, i.e., flushes the shared array and yields the processor. The victim's execution follows, where it writes a value repeatedly to the shared array. The parent process then wakes up and performs timed reads of the entire array. A hit is considered a successful attack. The attacker does not see any hit with our defense simulation enabled in gem5.

```
if parent
    flush shrd_mem;
    sleep;
    read shrd_mem; // cache hit
else
    read shrd_mem;
```

2) Attacking RSA: We use the `flush+reload` technique to attack the GnuPG version of RSA, as described in the original paper [46]. The attack was tested both on real hardware and the gem5 simulator, both running Linux. The attacker is an independent program, sharing the same machine and hence the caches.

On a real machine, we install a non-stripped GnuPG library and locate the offsets for the Square, Multiply, and Reduce functions. The shared library has the encryption algorithm for exponentiation, which performs a sequence of Square-Reduce-Multiply-Reduce for processing a key bit value 1 and a sequence of Square-Reduce when processing a clear bit. RSA encryption is an example where the control flow through the shared library is indexed using secret information, i.e., in this case, bit values from the secret key.

In the original attack, the attacker flushes the cache and then accesses the memory location for the Square, Multiply, and Reduce functions in a loop, using the time to process a 1 or 0 bit coupled with whether or not accesses hit in the cache to extract information about the key being used. In our evaluation, we simplify the attack and assume a cache hit in the attacker process represents a successful attack.

We calculate the time required for a cached and uncached access on the experimental real machine and set that as the threshold for the cache hit. The attacker program is an independent program running a loop to flush and read memory. Reading the timestamps must be fenced/ordered with respect to the memory access being timed to avoid speculative loads. The attack goes through, i.e., the independent attacker program gets hits for its accesses as a simultaneously running victim process performs an encryption. We are able to launch the attack both on a real machine and in gem5 *full-system simulation* mode.

Our defense in gem5 disallows any cache hit in the attacker process since the attacker's timed access is preceded by a flush. The defense allows a cache hit in a process only if it has suffered a cache miss for its *first access*. Since the access after the flush to a cached data is the *first access*, which is delayed, the attacker does not perceive a hit. This attack was the key demonstration for the `flush+reload` attack and our defense successfully breaks the attack.

3) S-bits Do Not Introduce Additional Side Channels: The additional *s-bits* do not introduce additional side channels for the following reasons:

- a process executing on a hardware context will not see *s-bits* associated with other hardware contexts
- *s-bits* are saved and restored at a context switch so that a process will only ever be able to access its own *s-bits*
- *s-bits* are saved in process-specific data structures in software, accessed only at a context switch, and are only accessible to a trusted computing base
- *s-bit* saves and restores at a context switch are constant time operations and therefore do not leak information

B. Performance Evaluation

1) First Access Delay: We evaluate the performance overhead of our first-access delay mechanism 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. Figure 7 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 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%.

Figure 8 shows *first accesses* 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 in wrf and perlbench is due to their larger shared instruction memory footprint. An interesting observation is that both perlbench and wrf have higher first access MPKI in the last-level cache when run with the same benchmark. However, when 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 further evaluate the overhead due to first access delay in the last-level cache when running pthread-based PARSEC benchmarks using 2 threads on 2 separate cores. We do this using system emulation mode in the simulator, where the clone syscall is emulated to allocate the new thread to another core. The geometric mean of the overheads due to TimeCache is 0.8%, as shown in Figure 9a. Since the threads execute on different cores, L1I and L1D for both cores have no *first access misses*, as shown in Figure 9b. In the case of PARSEC, each thread incurs *first access misses* for accesses across different execution contexts at the LLC for both code and data.

The exact overheads and the change in the number of misses per thousand instructions (MPKI) for the last-level cache is presented in Table II. 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.

TABLE II: SPEC2006 and PARSEC 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
fluidanimate	1.029	0.1317	0.1583
raytrace	1.0015	0.2833	0.2836
blackscholes	1.0013	0.0466	0.0511
x264	1.0052	0.8264	0.8634
swaptions	1.0025	0.0051	0.0053
facesim	1.0086	3.3585	3.3589
average	1.008	0.1702	0.1808

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 (Figure 10). 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 [14], resulting in fewer *first access misses* after a context switch. These numbers indicate that the defense scales well with larger caches.

Fig. 7: 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.

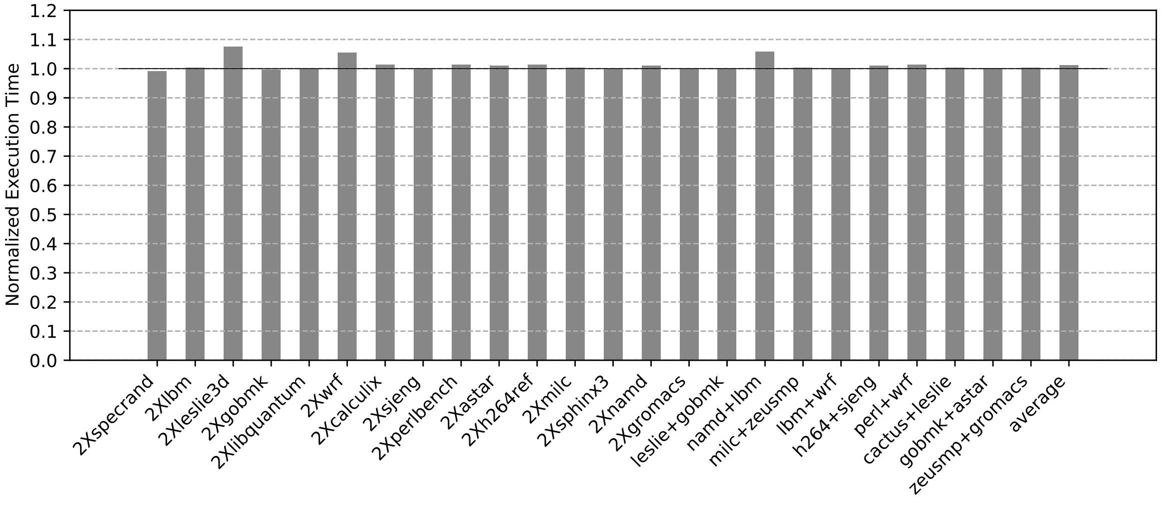
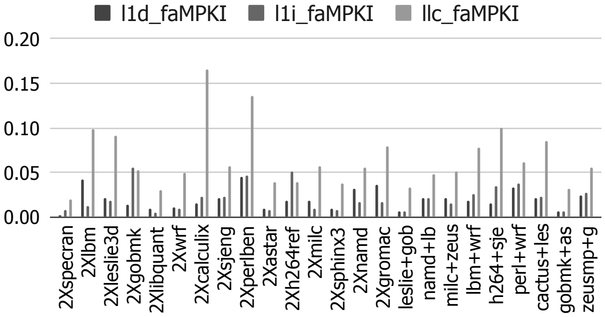


Fig. 8: Delayed access MPKI at each cache level for single-core experiments



C. 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 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.

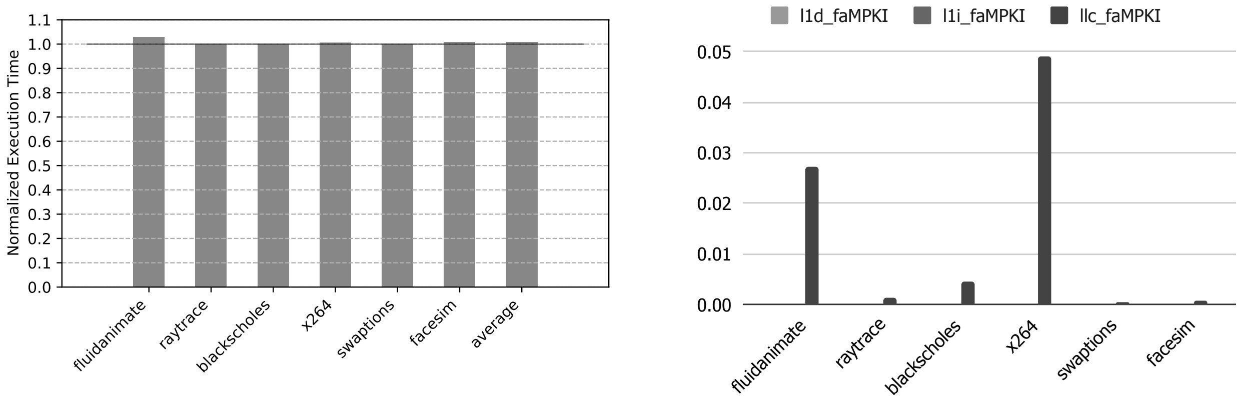
In our evaluation, we use 32-bit *Tc* timestamps to keep the area overhead low. The number of bits used for the timestamp counter has an impact on the frequency of timestamp rollover and is a parameter that can be tuned by the chip maker. A timestamp rollover can result in an additional miss after 2^{32} 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 [50]. 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.

Fig. 9: 2-core, 2-thread PARSEC benchmark normalized execution time (execution time with TimeCache/execution time without) and per-cache MPKI.



(a) Normalized execution time. Average performance overhead due to delayed first accesses is 0.8%

(b) Delayed access MPKI across different cores

D. 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 2.4 μ 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.

VII. Other Attacks on Shared Software

A. LRU Attack

LRU attacks exploit the cache line replacement policy and depend on eviction set creation [42]. They do not depend on shared software but can also be launched using shared software. The attack proceeds by creating an eviction set *w*, and accessing a shared cache line *l* followed by (*w-1*) cache lines of the eviction set. After a time-lapse to allow the victim’s execution, the attacker accesses the last element in the eviction set. The access replaces the oldest line, which is 1 of the (*w-1*) lines accessed by the attacker if the victim accesses *l*. This attack, like the other contention attacks that require an eviction set, can be prevented using randomizing caches.

B. Coherence Attack

Two types of attacks on shared memory due to the coherence protocol have been identified in the literature [13], [45]. One variant is know as *invalidate+transfer*, where the attacker flushes a cache line, resulting in the cache line being flushed from all the private caches, and experiences a remote cache access latency on a subsequent load if the victim running on another processor accessed the same shared memory [13]. The second variant exploits the difference in the access response time of *Exclusive* and *Shared* cached lines [45].

We can prevent these types of attacks using TimeCache by waiting for a DRAM response even when the data is available in some remote cache or Last level Cache when the accessing context’s *s-bit* is not set.

C. Flush + Flush

This attack uses the fact that the execution time of the ‘clflush’ instruction depends on whether the data is available in the cache. The ‘clflush’ instruction aborts early and takes less time if the data is not cached [9]. If shared memory is accessed by the victim, the data is cached and rather than a fast reuse, the attacker can infer this by a second ‘clflush’ taking longer. This form of attack can be prevented by making ‘clflush’ a constant time instruction. One of the ways to do so is performing a dummy write back when the data is not cached.

D. Evict + Time

Evict+Time [25] is another contention-based attack, like the LRU attack, and does not depend on shared software, but can be launched using shared software. The attacker evicts cache sets and checks to see if the evictions slow down the victim. A finer grain version of Evict+Time is possible on shared memory where the attacker flushes (using 'clflush') a shared cache line and times the victim's execution. This attack remains noisy and less practical unless the attacker communicates with the victim to trigger and time a specific access.

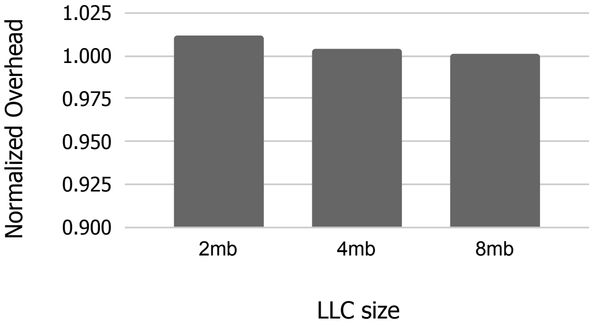
VIII. Related Work

Existing solutions for protecting against cache side channel attacks that exploit shared code and data either resort to cache partitioning or remove timing information from the accesses. Both approaches incur significant overhead.

A. Cache Partitioning

While partitioning the cache can prevent multiple types of attacks, it comes at a performance cost due to reduced cache availability. Statically partitioning caches causes significant performance deterioration as some parts of the cache become unavailable to other processes [26], whereas dynamic cache partitioning can achieve lower overheads by reallocating space as needed. SecDCP [39] is one such dynamic partitioning technique, which broadly categorizes applications as either 'confidential' or 'public' and prevents any information leakage from confidential applications to public applications, but allows information flow in the other direction. Although this dynamic cache partitioning technique performs better than its static counterpart, it provides a very coarse-grained security classification [39]. Another dynamic cache partitioning technique utilizes page coloring to allocate pages to a secure domain [35], [48], but may incur significant copy costs for recoloring. DAWG [17] partitions cache ways, supporting a maximum of 16 security domains at a time, and has an associated performance overhead due to reduced cache availability of 4-12%. PLCache defends against reuse attack by locking cache lines with process IDs to prevent their eviction by other processes [40]. It can be seen as a line-wise cache partitioning and since the locked lines are not available for eviction, there is a performance degradation of about 12%.

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



B. Last Level Cache Defense

1) Using Intel CAT-based Partitioning: Both Catalyst [22] and Apparition [7] have demonstrated the use of Intel's cache allocation technology (CAT) to achieve cache partitioning for mitigating cache side channels. The performance of the systems depend on their ability to reassign caches to different applications and keep cache flushes to a minimum. Apparition [7] uses one Class of Service (CLOS) per application and flushes it across context switches. Catalyst uses pinned pages to provide a solution suited to cloud service providers. This defense mechanism is suited to preventing cross-VM attacks and attacks targeted at the LLC, and is not suited to higher-level caches. The design further requires manually tagging pages that should be pinned or need to remain secure.

2) FTM: First Time Miss (FTM) [29] prevents reuse of data in the LLC that was brought in by another core by using the directory presence bits. The defense assumes that the victim and attacker can only share a last-level cache managed using a directory protocol and must otherwise run on isolated hardware. Not all usage scenarios would allow hardware isolation of attacker and victim processes, nor for that matter might it be possible to identify all attacker processes.

The threat model, and hence the defense mechanisms in TimeCache, is stronger than that of FTM. We make no assumptions about resource isolation between attacker and victim beyond requiring that they use separate address spaces. We protect against attacks at any level of the cache, including the shared LLC, by separating and identifying the hardware context (from a potentially hyperthreaded core) performing access to code or data. While recognizing a first-time miss is straightforward in the presence of a directory, and in the absence of hyperthreading and context switches across protection boundaries, doing so across context switches in a shared cache is enabled by our novel bit-serial, timestamp parallel comparator and our use of time to recognize a first access across context switches. Our defense is also able to mitigate cross-process Spectre variants by eliminating the cross-process and cross-core reuse cache side channel used by Spectre.

C. Removing Time & Constant Time Algorithm

The ability to time data accesses precisely can also be seen as the cause for side channel exploits. Taking this ability away from untrusted applications is not sufficient to prevent the attacks. There are several new techniques to obtain timestamps in up to microsecond granularity. These methods provide alternate timing primitives or recovery of clock resolution [32] on systems that obfuscate time by reducing the clock resolution.

Other approaches to mitigating side channels in shared memory suggest program transformation for constant time implementation [20], [30], [31]. These program transformations have impractical overhead due to making each critical access $O(n)$ [30] and are not useful for large shared libraries.

IX. Discussion

Sharing software is an important component of computing systems for efficiency and consistency. This work eliminates a channel for the leak of secret data via monitoring a victim's access to shared content using shared caches. In the absence of shared content, shared caches still allow a victim's access behavior to be monitored, but the information channel is far less accurate. In particular, a "Prime+Probe" attack fills (primes) an entire cache set, and infers the cache set accessed by the victim, based on whether the attacker's probe hits or misses. Proposed defenses for a "Prime+Probe" attack include a randomizing cache [28] [21]. These defenses do not work for attacks against shared content, which provides a more accurate/less noisy channel of information. TimeCache in conjunction with these defenses can provide a more complete defense.

Other approaches to defending against more recent attacks like Spectre either stall execution, or make speculative instructions invisible to succeeding load requests [43] [16]. They do not prevent non-speculative cache side channels. Speculative side channel attacks rely on conventional side channels for leaking speculatively loaded data to the attacker, i.e., the data is eventually leaked via a conventional side channel. By breaking conventional cache attacks, we also prevent speculative side channel leaks.

X. Conclusion

We have designed and evaluated a timestamp-based defense against timing side channel attacks that rely on reuse of shared software in caches to learn secret information. TimeCache works across context switches and prevents attacks from cross-core, same core, or SMT contexts, and at any level of cache, without the need for cache partitioning. To perform timestamp comparisons in parallel, we use an SRAM array that allows bit-serial, timestamp-parallel comparison with easy transposed access. We have evaluated the defense against microbenchmark attack programs and the classic `flush+reload` attack using the gem5 simulator. On SPEC2006 and PARSEC, the performance overhead is 1.13% and 0.8% on average, most of which is due to delaying first accesses, with copying process-specific *s-bits* at context switches adding 0.024%. Our defense against timing side channels through shared software retains the benefits of allowing processes to utilize the entire cache capacity of a shared cache and allows cache and memory pressure reduction through data deduplication and copy-on-write sharing.

XI. Acknowledgements

This work was supported in part by National Science Foundation (NSF) Awards CNS-1618497 and CNS-1900803. We thank Sreepathi Pai for his feedback during early discussions of the ideas in this paper.

References

[1] Kernel samepage merging (memory deduplication). https://kernelnewbies.org/Linux_2_6_32#Kernel_Samepage_Merging_.28memory_deduplication.29, 2017.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *International Symposium on Computer Architecture (ISCA)*, pages 280–289, June 1988.

[3] Kenneth E. Batcher. Bit-serial parallel processing systems. *IEEE Transactions on Computers*, (5):377–384, 1982.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The GEM5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[5] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 769–784, 2019.

[6] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXpectre attacks: Stealing Intel secrets from SGX enclaves via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.

[7] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L Cox, and Sandhya Dwarkadas. Shielding software from privileged side-channel attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1441–1458, 2018.

[8] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 383–396. IEEE Press, 2018.

[9] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[10] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 897–912, 2015.

[11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[12] Wei-Ming Hu. Lattice scheduling and covert channels. In *Proceedings 1992 IEEE Computer Society Symposiumhttps://www.overleaf.com/project/6056a658ee81be9ae173d875 on Research in Security and Privacy*, page 52. IEEE, 1992.

[13] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 353–364, 2016.

[14] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.

[15] Keren Jin and Ethan L Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, 2009.

[16] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. *arXiv preprint arXiv:1806.05179*, 2018.

[17] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[19] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[20] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGPLAN Notices*, 50(4):87–101, 2015.

[21] F. Liu, H. Wu, K. Mai, and R. B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, Sep. 2016.

[22] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

[23] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[24] Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapotre, Muhammad Khurram Bhatti, and Guy Gogniat. Winter is here! a decade of cache-based side-channel attacks, detection & mitigation for rsa. *Information Systems*, page 101524, 2020.

[25] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[26] D Page. Partitioned cache architecture as a ėide-channel defence mechanism. 2005.

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

[28] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 360–371. ACM, 2019.

[29] Kartik Ramkrishnan, Stephen McCamant, Pen Chung Yew, and Antonia Zhai. First time miss: Low overhead mitigation for shared memory cache side channels. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[30] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 431–446, 2015.

[31] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 71–86, 2016.

[32] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: high-resolution microarchitectural attacks in javascript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[33] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*, pages 279–299. Springer, 2019.

[34] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 15–26, 2012.

[35] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 194–199. IEEE, 2011.

[36] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 947–960. IEEE, 2018.

[37] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.

[38] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. In *NDSS*, 2019.

[39] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. Secdcp: secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference*, page 74. ACM, 2016.

[40] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Computer Architecture News*, 35(2):494–505, 2007.

[41] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: thwarting cache attacks via cache set randomization. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 675–692, 2019.

[42] Wenjie Xiong and Jakub Szefer. Leaking information through cache lru states. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 139–152. IEEE, 2020.

[43] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[44] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 347–360. IEEE, 2017.

[45] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179. IEEE, 2018.

[46] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[47] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. A hardware design language for timing-sensitive information-flow security. *Acm Sigplan Notices*, 50(4):503–516, 2015.

[48] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.

[49] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003, 2014.

[50] Hongzhou Zhao, Arrvindh Shriraman, and Sandhya Dwarkadas. SPACE: Sharing Pattern-Based Directory Coherence for Multicore Scalability. In *International Symposium on Parallel Architectures and Compilation Techniques (PACT)*, September 2010.