Synchronization Storage Channels (S^2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions

Jiyong Yu and Aishani Dutta, *University of Illinois Urbana-Champaign;* Trent Jaeger, *Pennsylvania State University;* David Kohlbrenner, *University of Washington;* Christopher W. Fletcher, *University of Illinois Urbana-Champaign*

https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jiyong

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Synchronization Storage Channels (S^2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions

Jiyong Yu
University of Illinois Urbana-Champaign

Aishani Dutta
University of Illinois Urbana-Champaign

Trent Jaeger
Pennsylvania State University

David Kohlbrenner
University of Washington

Christopher W. Fletcher
University of Illinois Urbana-Champaign

Abstract

Shared caches have been a prime target for mounting cross-process/core side-channel attacks. Fundamentally, these attacks require a mechanism to accurately observe changes in cache state. Most cache attacks rely on timing measurements to *indirectly* infer cache state changes, and attack success hinges on the reliability/availability of accurate timing sources. Far fewer techniques have been proposed to *directly* observe cache state changes without reliance on timers. Further, none of said 'timer-less' techniques are accessible to userspace attackers targeting modern CPUs.

This paper proposes a novel technique for mounting timer-less cache attacks targeting Apple M1 CPUs named Synchronization Storage Channels (S^2C). The key observation is that the implementation of synchronization instructions, specifically Load-Linked/Store-Conditional (LL/SC), makes architectural state changes when L1 cache evictions occur. This by itself is a useful starting point for attacks, however faces multiple technical challenges when being used to perpetrate cross-core cache attacks. Specifically, LL/SC only observes L1 evictions (not shared L2 cache evictions). Further, each attacker thread can only simultaneously monitor one address at a time through LL/SC (as opposed to many). We propose a suite of techniques and reverse engineering to overcome these limitations, and demonstrate how a single-threaded userspace attacker can use LL/SC to simultaneously monitor multiple (up to 11) victim L2 sets and succeed at standard cache-attack applications, such as breaking cryptographic implementations and constructing covert channels.

1 Introduction

The increasing complexity of modern processors has led to a plethora of micro-architectural side channels that can be exploited to infer sensitive information. Despite being one of the earliest targets to mount such attacks, the shared cache is still the most prominent—due to its shared use among all tenants on the same processor, the relative ease with which it can be monitored, and the richness of information that can be gleaned through it. By measuring cache usage within a victim process, an attacker obtains information about the victim's memory access pattern, which can be useful in breaking cryptographic implementations [38,43,68], key logging [16,33,50], browser fingerprinting [55], model stealing [67], and aiding transient execution attacks [31,36,58].

A fundamental requirement of any cache side-channel attack is a way to accurately measure cache state changes. Most existing attacks *indirectly* observe the cache state by measuring memory access latencies with high-resolution timers. These can be used to deduce the cache level an address resides in, and to further deduce the presence of the victim's address(es) in those cache levels. Even if only provided a low-resolution timer, techniques have been proposed to enhance the effective resolution [32,53,54,66]. Regardless, precise timing measurement is prone to (or can be aggravated by adding) noise [23,39], requires micro-architecture-specific profiling (e.g., to ascertain cache latencies), and can be fully mitigated by blocking the use of explicit timers. The attacker can also craft 'implicit timers' with a counter incremented by sibling threads in the absence of explicit timers [34,53,65], but this requires additional attacker capabilities such as running multiple attack threads concurrently.

To circumvent the limitations/defenses associated with timers, an attacker ideally would like a way to *directly* and *precisely* measure cache state without relying on timers. Yet, there is scant literature on such *timer-less* attacks, and at present all known methods have limitations. For instance, cache storage channels [17] rely on uncacheable memory, which can only be exploited by privileged attackers; Prime+Abort [10] and its variant [30] exploit Hardware Transactional Memory, which is a rare and even deprecated feature implemented only by specific vendors such as Intel [25]. Currently, no general-purpose primitive is available that allows userspace attackers to *directly* and *precisely* observe the shared cache state on modern CPUs.

This paper's key insight is that the implementation of *hardware synchronization primitives* on modern CPUs, specifi-

cally Load-Linked/Store-Conditional (LL/SC) instructions on the Apple M1, can be exploited to directly measure whether cache evictions have occurred.

LL/SC are general-purpose instructions in many common ISAs (such as ARM and RISC-V) for implementing synchronization/mutual exclusion. In a nutshell: LL loads an address and marks it as 'exclusive'[1] in the memory system. SC is a store that a) 'succeeds', i.e., performs the store, if and only if its address has been marked 'exclusive' (by some older LL) and b) writes to a register whether it succeeded. If any store (including an SC) from any processor core writes to an address marked exclusive, the exclusive state is cleared. That is, LL-SC implements atomic read-modify-write when there isn't an intervening write to the target address, and performs a NOP when there is an intervening write/atomicity violation. *In both cases, it makes architectural state changes (by writing the success bit to a register).* In the normal use of LL-SC, if an SC fails, the thread using LL-SC will retry the LL-SC sequence until it reports success (i.e., "lock acquired").

Ideally, only shared variables will be exclusive, and the variable's exclusivity status is maintained regardless of where the variable is cached in the memory hierarchy. However, we found that the implementation of LL/SC in the recent Apple M1 drops the exclusive semantic when the address is evicted from the L1 data cache, causing the later SC to fail conservatively. In practice, this design does not compromise correctness, as a benign use of LL/SC involves retrying when SC fails. However, it *does* enable attackers to directly measure whether a local L1 eviction has taken place—using only a single user-space attacker thread and without the use of any timer—by monitoring the result of the SC.

Following this idea, we propose *Synchronization Storage Channels (*S^2C*)*, the first timer-less cache side-channel attack technique on the Apple M1, and also the first micro-architectural attack to exploit hardware synchronization instructions (specifically LL/SC). At a high level, S^2C aims to build a cross-core side channel to monitor memory accesses of other processes sharing the same cache as the attacker.

The technical core of the paper proposes techniques to overcome two main limitations in LL/SC semantics that impede shared cache attacks. First, LL/SC on the M1 only indicates L1 evictions, but to perform cross-core attacks, notifications from the shared L2 cache (L2 evictions) are required (both for building L2 eviction sets and performing the actual attack). Second, LL/SC only allows the attacker to monitor evictions on a single cache line, as opposed to traditional timer-based cache attacks that pre-occupy a cache set with multiple attacker-controlled lines and monitor all those lines. Sometimes, the attacker may also need to simultaneously monitor evictions happening to multiple cache sets.

To enable S^2C-based attacks that can monitor a single L2 set, we reverse-engineer a variety of new features related to the L2 cache on the M1 (for example, that the L2 is inclusive, implements the AutoLock optimization [64] and uses a dynamic replacement policy). We then develop techniques that exploit said features to enable single-threaded, unprivileged Prime+Probe-like attacks through the L2 using only LL/SC.

To enable S^2C-based attacks that can simultaneously monitor multiple L2 cache sets, the insight is to view LL/SC *as a general-purpose single-bit communication channel*, as opposed to *just* a means to monitor whether a victim accessed the particular cache set. Following this, we construct a weird circuit [12] that micro-architecturally computes the logical-OR of whether the victim accessed at least one of an attacker-specified set of L2 cache sets—and communicates the single-bit result of this logical-OR through LL/SC to the attacker's architectural state. While weird circuit constructions are not the main focus of the paper, our weird circuit is relatively simple conceptually (relies purely on out-of-order execution and LL/SC as opposed to requiring speculative execution/Intel's TSX [12]), and thus may be of independent interest.

In summary, the paper makes the following contributions.

- We present the first timer-less cross-core cache attack technique that exploits hardware synchronization instructions, namely Load-Linked/Store-Conditional (LL/SC) on the Apple M1. We call this technique Synchronization Storage Channels (S^2C).
- We identify that LL/SC serves as a *direct* and *precise* architectural observation channel to monitor micro-architectural events (L1 evictions).
- We conduct a detailed reverse-engineering of the M1's L2 caches, which is necessary to exploit S^2C and may benefit future attacks against the M1.
- We develop techniques to overcome challenges in using LL/SC to perform cache attacks. In particular, we develop methods to monitor L2 evictions using LL/SC and methods to simultaneously monitor evictions on multiple L2 sets (despite LL/SC natively tracking L1 evictions for only a single line at a time).
- We show that S^2C can simultaneously monitor up to 11 L2 sets with high accuracy, and can be used for building covert channels as well as attacking cryptographic implementations such as T-table AES.

The source code of the attack implementation as well as the evaluation can be found in: https://github.com/FPSG-UIUC/S2C.

Responsible Disclosure. We responsibly disclosed our findings to Apple, who acknowledged our findings.

[1] Not to be confused with the Exclusive (E) state in modern cache coherence protocols, e.g., MESI, which is a related but different concept.

2 Background

2.1 Cache Side-Channel Attacks

The goal of a cache side-channel attack is to infer a victim program's secret-dependent memory access pattern by monitoring the victim's use of a shared cache. In general, cache attacks can be categorized into the following two types.

The first type, known as *flush-based*, works by the attacker flushing a line corresponding to a shared address from the cache and monitoring whether the victim re-reads said address (refilling the cache with the corresponding line). This type includes techniques such as Flush+Reload [19, 68] and its variants Evict+Reload [16] and Flush+Flush [15]. These attacks are only capable of learning memory accesses to data that is shared between the attacker and the victim. Nonetheless, they have the advantage of inferring the precise cache line-granular address accessed by the victim.

The second type, known as *contention-based*, relaxes the requirement for shared memory by monitoring how the victim's cache lines (addresses) contend for space in the cache with the attacker's cache lines (addresses). Our attack falls into this category. All contention-based attacks, such as Prime+Probe [27, 29, 38, 43], Prime+Abort [10, 30], Reload+Refresh [7], and Prime+Scope [45], follow a similar attack procedure. First, the attacker *primes* the cache by filling a target shared cache set with attacker-controlled lines. Later, the attacker *probes* the same cache set to observe whether any of its lines have been evicted, and from this deduces if the victim line(s) has been accessed. By monitoring shared cache contention, contention-based attacks do not require shared memory, but they only learn a subset of a victim address bits (i.e., those bits used to choose the cache set).

An accurate method for determining the location of a specified cache line within the cache hierarchy, is crucial for any cache attack. Most methods are based on timing measurements (*timer-based* attacks), yet several techniques achieve this without relying on timing (*timer-less* attacks).

Timer-based attacks Most cache attacks rely on high-resolution timers to measure the latency of accessing a specific cache line, by either reading the cycle count register directly (e.g., via `rdtscp` in x86), or exploiting special instructions that interact with the cycle counter register (e.g., `monitor`/`mwait` in x86 [70]). When only low-resolution timers are accessible, the attacker can also leverage existing techniques to amplify small access latency differences so that they are detectable by the timer [32, 53, 54, 66]. Despite these efforts, a fuzzy or inaccessible timing source can still impede attacks that rely on timers [23, 32, 39]. When an explicit timer is absent, a counter incremented by a sibling thread can serve as an *implicit* timer [34, 53, 65]. However, this requires the attacker to have additional capabilities, such as spawning and concurrently running multiple threads. This may not be feasible in practice. For instance, Javascript programs in browsers are single-threaded and sandboxed [52, 54].

	Level	Ways	Sets	Line Size	Total size
P-core	L1D	8	256	64 B	128 KB
	L2	12	8192	128 B	12 MB
E-core	L1D	8	128	64 B	64 KB
	L2	16	2048	128 B	4 MB

Table 1: Apple M1 cache parameters (from Table 2 of [49])

Timer-less attacks The limitations of timers can be overcome if the attacker has the ability to directly measure cache state, i.e., directly convert micro-architectural state changes in the cache to architectural state changes in the register file. However, existing methods for doing so are limited and scarce. One method is cache storage channels [17], which directly returns to the attacker whether its data is cached or not. This primitive, however, is not available to normal user-space attackers as it requires configuring non-cacheable memory. Hardware Transactional Memory is another feature that communicates specific types of cache misses/evictions as transactions abort when their data is evicted from the shared cache [10, 30]. Yet, this feature is only available on some Intel products and has been deprecated. Intel has also disabled the use of TSX by default on CPUs that support it through a microcode update [26].

2.2 Apple M1

Apple has recently started using a new processor architecture on its laptop, desktop, and tablet devices. The new processor design, including the M1 and the latest M2, is based on the ARMv8-A ISA. We have confirmed that our findings about LL/SC in Apple processors that are later discussed (§3) apply to both the M1 and the newest M2, but in this paper, we mainly focus on the M1.

An M1 CPU consists of four performance-oriented cores (P-cores) and four energy-oriented cores (E-cores). Based on prior works [49], each P-core/E-core has its own private L1 data cache (L1). There are two separated L2 caches, one shared among all four P-cores, and the other shared among all four E-cores. The associativity, number of sets, line size, and total size of each cache is shown in Table 1. The M1 supports regular pages of size 16 KB and 32 MB huge pages natively. Notice that the M1 does not implement Simultaneous Multi-Threading (SMT), thus each core only runs one thread.

Previous studies [9, 34] have highlighted two difficulties in performing cache attacks on ARM processors. By adopting the ARM ISA, the M1 also inherits both difficulties. First, the cycle count register on the M1 is only accessible to privileged software, unlike x86 processors where the corresponding register can be read by unprivileged instructions. Second, ARM does not have dedicated instructions for flushing a specific ad-

```
Retry: lock_val = LDREX [lock_addr]
if (lock_val == FREE) {
  fail = STREX BUSY, [lock_addr]
  if (fail) { goto Retry; }
}
```

Figure 1: A test-and-set-style lock implemented with ldrex/strex. Even when all threads perform ldrex simultaneously and see a FREE lock, only one thread will successfully perform strex and acquire the lock. All other threads will encounter failed strex and retry.

dress from caches (like clflush on x86), making attack methods that rely on those instructions, such as Flush+Reload [68] and Flush+Flush [15], ineffective on the M1.

2.3 Load-Linked/Store-Conditional in ARM64

Load-Linked/Store-Conditional, also referred to as Load-Exclusive/Store-Exclusive (ldrex/strex) by ARM [6], are used in common RISC ISAs such as ARM, MIPS, and RISC-V for implementing synchronization and mutual exclusion. A ldrex loads a specified address while marking it as *exclusive with respect to the current core* in memory. This means that an address can be exclusive to multiple cores at a given time when multiple sibling threads running on different cores execute ldrex concurrently (e.g., for competing for a lock like Figure 1). One core can track *at most one exclusive address*, meaning a younger ldrex will overturn the exclusive address marked by an older ldrex on this core.

A strex only performs the store operation when the address is *exclusive to the current core* and returns a bit in its output operand indicating whether the store is performed. Whether a strex succeeds or not, it clears all exclusive states associated with its address for every core. This ensures that when multiple threads simultaneously perform ldrex, such as competing for a lock in Figure 1, only the first thread to perform strex will succeed, forcing all other threads to lose their exclusive access to the lock and retry the procedure. Lastly, a regular store behaves like an always-succeeding strex and also changes the address back to non-exclusive for all cores.

The exclusive states of cached addresses and how they respond to different memory operations are managed by an *exclusive monitor* [4]. The exclusive monitor tracks exclusiveness at a granularity called the *Exclusive Reservation Granule (ERG)*. Whether the exclusive monitor is implemented as a dedicated unit or integrated with existing cache logic such as cache coherence protocol, as well as the ERG size, is design-specific.

Exclusive address vs. the Exclusive cache-coherent state. Modern cache-coherence protocols, e.g., MESI, use an Exclusive (E) state to reduce bus invalidations when Shared+Clean (S) data is Modified (M). Although the exclusive monitor implementation may piggyback on top of the coherence protocol, it has different semantics when compared to the E cache-coherence state. E in cache coherence means "clean, owned by a single core". However, in the context of ldrex/strex, a single address can be marked exclusive simultaneously by multiple cores. For the rest of the paper, we refer to the target address of ldrex/strex as the *exclusive address*.

3 New Attack Primitive on M1 using LL/SC

This section introduces the attack primitive associated with the behavior of ldrex and strex on the Apple M1, which enables the S^2C attack technique proposed in this work.

3.1 Micro-architectural strex Failures

As explained in §2.3: A strex instruction, immediately following a ldrex, might return 'failed' when multiple threads are competing to write to the same shared data. This requires the implementation of the exclusive monitor to track the address's exclusiveness regardless of its location in the cache hierarchy. However, the official ARM specification states that in some implementations, strex might return fail for micro-architectural reasons, e.g., cache evictions [5]:

> *An implementation might clear an exclusive monitor between the ldrex and the strex, without any application-related cause. For example, this might happen because of cache evictions.*

We investigated whether such an implementation exists, looking specifically at the widely-used Apple M1. Our questions are: *Can a single-threaded program, that executes ldrex-strex to its own private data, see strex failures due to cache evictions—even if it performs no action that is known to revoke the exclusive state (from §2.3)? If so, from what level(s) of the cache can said evictions lead to strex failures?*

The rest of this section answers these questions. To summarize: in a ldrex-strex sequence, even when ldrex and strex instructions are applied to private data, strex can fail when the exclusive address accessed by ldrex is evicted from the L1 cache before strex executes.

3.2 Experiment Design and Methodology

To answer the questions in §3.1, we designed the experiment shown in Algorithm 1. The code contains a ldrex (line 3) and a strex (line 6) to addr, which points to a local variable. Between the ldrex and strex, we traverse a set of random addresses S (S never includes addr) which may evict addr from the L1 cache (line 4). To identify the location of addr after the possible cache eviction, we load addr right before strex (line 5), and compare its access latency with the L1 access latency L1_Latency using timers. The experiment, therefore, studies how strex failures correlate with L1 evictions.

Algorithm 1: Code for testing the correlation between strex failures and L1 evictions of the exclusive address.

```
Input: addr: a selected target exclusive address
1  for i = 1 to 1000 do
2      Generate a set of random addresses S
3      val = LDREX [addr]
4      traverse S // may or may not evict addr from L1
5      latency = measure latency of load [addr]
6      fail = STREX val, [addr]
7      evicted_from_L1 = latency > L1_Latency
8      print fail, evicted_from_L1
9  // An example of counting the output:
10 // fail = True, evicted_from_L1 = True: 518
11 // fail = True, evicted_from_L1 = False: 0
12 // fail = False, evicted_from_L1 = True: 0
13 // fail = False, evicted_from_L1 = False: 482
```

We test the above code on both P-cores and E-cores, by configuring the thread quality of service [3]. The load latency is measured by reading the M1's cycle count register before and after the load and computing the difference. We use a custom kernel extension (similar to the prior Pacman Attack [49]). Note that we read timers in this experiment, not in the actual attack technique. L1_Latency can be obtained by executing two consecutive loads to the same address (with a memory barrier in between for maintaining ordering) and measuring the latency of the second load.

3.3 Result and Takeaway

Line 10-13 in Algorithm 1 shows an example of counting the output of the experiment. We further ran Algorithm 1 over 10 times, and consistently observed that fail correlates perfectly with evict_from_L1. In other words, strex to a local exclusive address never fails when the address still remains in the L1, and always fails when the address resides only in lower levels of the cache. This observation aligns with the information in ARM's documentation (§3.1). Therefore, we can confirm that on Apple M1 CPUs, when a ldrex-strex pair is applied to a local address, strex can fail if the address is evicted from the L1 in between when the ldrex and strex are performed. We discuss the potential implementation of the exclusive monitor on the M1 that can produce this behavior in §8.

We performed another experiment to determine the ERG size. The experiment consists of only a single ldrex followed immediately by a strex, but when the two have different address operand values. We observed that strex can succeed even if ldrex and strex addresses differ. Also, strex only succeeds when the two addresses are on the same 64-byte-aligned block. The fact that the ERG size is 64-byte (the L1 line size) not 128-byte (the L2 line size) implies that the exclusive monitor may be implemented to only track exclusiveness for L1 lines, which is consistent with our above finding about strex failures. We also conduct experiments to verify that the behavior of ldrex/strex follows ARM's specification (§2.3). For example, we find that each core can indeed only monitor one exclusive address at a time.

Microarchitectural factors influencing the result of strex might seem like a bug that affects correctness. However, in practice, this behavior is acceptable. Regular programs always use ldrex and strex for thread synchronization purposes, which are inherently non-deterministic due to unpredictable thread interleavings. Programmers will normally use the output of strex to create higher-level synchronization primitives, for example by retrying the code until the strex succeeds as shown in Figure 1. Therefore, a well-written program should not break given this additional cause of strex failures, apart from possibly wasting cycles from unnecessary code retries.[2]

From the security perspective, this observation is relevant in the context of cache side-channel attacks. As explained in §2.1, the ability to directly and accurately observe cache state circumvents the defenses and other challenges associated with timer-based attacks. Utilizing ldrex and strex instructions, an attacker can detect L1 evictions, which can be caused by contention in lower-level shared caches, such as the L2 cache in M1 processors. On the other hand, such detection is limited to one exclusive address at a time, since each core can only monitor one exclusive address. The rest of the paper builds upon this idea and presents S^2C, a timer-less attack technique aimed at the M1's shared L2 caches.

4 Reverse-Engineering M1's Shared L2 Cache

The goal of the S^2C attack technique is to leverage ldrex and strex to expose information leakage through the shared L2 cache, which requires detailed knowledge about the M1's L2 cache. Given the lack of such information in existing research, in this work, we present the first detailed reverse-engineering of the M1's shared L2 caches, which encompasses various key characteristics of the L2 cache, such as the inclusion policy (§4.3), the replacement policy (§4.4) and the set index mapping (§4.5). Due to the observability of ldrex/strex being limited to the L1 (§3.3), these details are essential for a successful S^2C-based attack. The information in this section may also benefit other cache attacks.

We describe the reverse-engineering process with a primary emphasis on the caches used by P-cores, as the results obtained from E-cores are similar. This section's reverse engineering makes use of all capabilities required, such as making extensive use of the privileged cycle count register, and executing experimental code on multiple cores simultaneously. For the actual attack technique, starting in §5, we limit attackers

[2] Note that livelock can occur when too many memory accesses are placed between ldrex and strex, causing the exclusive address to be constantly evicted from L1. There is no guarantee that this won't occur, but programmer guidance (which says to minimize the use of instructions between a ldrex and strex pair, plus several other rules of thumb) tries to minimize its likelihood in practice [5]. We discuss this further in §8.

to single-threaded execution without access to timers.

Terminology We define addresses mapped to the same cache set as *congruent* addresses, and further define addresses mapped to the same L1 set as *L1-congruent*. Correspondingly, *L2-congruent* addresses are mapped to the same L2 set. An *L1/L2 eviction set* for a target address is defined as a set of addresses that are L1-/L2-congruent to the target address, with the size of at least the L1/L2 associativity. Such sets can be used to evict the target address from the L1 or L2, respectively.

4.1 Reverse Engineering Private L1s

Before reverse engineering the L2 cache, we first investigate the private L1 cache. We suspect that, like most modern CPUs, the M1's L1 cache is also virtually-indexed/physically-tagged, with a Least Recently Used (LRU) replacement policy. In this case, the L1 set index will correspond to the regular page offset subtracting the L1 line offset bits.

To test this, we first choose a random address and create a candidate L1 eviction set comprised of eight addresses (since the L1 is 8-way-associative) that share the same L1 set index bits as the target address. We start by accessing the target address, then traverse all 8 addresses in the candidate eviction set, and lastly measure the access latency to the target address and compare the latency with the L1 latency obtained from §3.2.

The results show that the candidate L1 eviction set always evicts the target address out of the L1 cache. Also if we drop one address from the eight-element eviction set, the target address will never be evicted. This confirms our hypothesis:

> L1 sets are indexed by the page offset bits subtracting the L1 line offset bits. The L1 cache adopts an LRU-based replacement policy for choosing evicted lines.

4.2 L2 Eviction Set Generation with a Timer

Our L2 cache reverse-engineering process relies heavily on L2 evictions. Since ARM does not provide cache flush instructions, we generate L2 eviction sets for inducing L2 evictions, similar to previous studies [9, 13, 22, 34]. However, identifying L2-congruent addresses for building the L2 eviction set is challenging given that the L2 set index may depend on physical page number bits.

Vila et al.'s algorithm [59] is the most popular eviction set generation algorithm that overcomes this challenge. The algorithm starts by adding randomly generated virtual addresses to a candidate set, and tests whether the candidate set can evict the previously loaded target address from the L2, until enough L2-congruent addresses are included and the eviction appears. At this moment, the candidate set is an *L2 eviction set superset*, meaning that it contains a valid L2 eviction set, but also a significant number of redundant, non-L2-congruent addresses. One thing to notice is that the L2's inclusion policy is not yet known. Assuming the L2 is exclusive of the L1, the first loaded target address will only be cached in the L1, not L2, therefore a valid L2 eviction set cannot evict the target address from the L2 unless it evicts the target address from the L1 first. So we force Vila et al.'s algorithm to generate only addresses with identical L1 set indices as the target address, ensuring that a valid L2 eviction set can always evict the target address from the L1 to the L2 first, and subsequently from the L2. Once obtaining the L2 eviction set superset, the algorithm prunes the superset iteratively and checks whether the remaining is still capable of evicting the target address until the set size reaches the L2 associativity, and the superset is now reduced down to a minimal L2 eviction set.

4.3 L2 Cache Inclusion Policy and AutoLock

Understanding the cache inclusion policy is crucial for cache attacks, especially our new S^2C attack technique, where the attacker can only observe the L1 cache state. Case in point, if the shared L2 is inclusive of the private L1, contention within the L2 will cause L1 evictions, allowing the attacker to detect cross-core activities by monitoring the L1.

One straightforward method for determining whether the L2 cache is inclusive of the L1 works as follows. First, we randomly select an address and create its L2 eviction set using the technique in §4.2. Next, we access this address on a processor core, and then traverse the eviction set on a different core. If the L2 cache is inclusive, traversing the eviction set will evict the address from the L2, and correspondingly from L1. If the cache is non-inclusive/exclusive, this L1 eviction will not happen. Hence whether the L2 is inclusive can be determined by measuring the access latency to the address in the end and comparing it with `L1_Latency`.

After running this experiment with different addresses, we always observe an L1 hit. In fact, a recent attack on the Apple M1 by Hetterich et al. [22] also observed this phenomenon and suggested that it could be due to several factors, namely an exclusive/non-inclusive policy or a hardware optimization used in some ARM CPUs called *AutoLock*. *AutoLock* is a common optimization used by ARM CPUs in conjunction with inclusive caches [64]. When evicting data from the shared inclusive L2, AutoLock prioritizes data that is not present in L1 caches, effectively *locking* those that are present in the L1, since they are likely to be frequently reused. However, Hetterich et al. did not confirm whether the M1's L2 cache is inclusive with AutoLock or exclusive/non-inclusive [22]. The rest of §4.3 explains how we determine that the L2 cache is actually inclusive with AutoLock, by leveraging a technique called *eviction set splitting*.

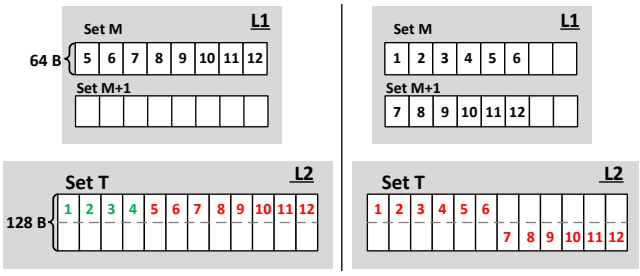


Figure 2: How eviction set splitting circumvents AutoLock. Every number represents a different address, and the values represent the access order. Green marks addresses without L1 copies (non-AutoLocked). Red marks addresses with L1 copies (AutoLocked). The right part shows eviction set splitting: the minimal 12-element eviction set is distributed over two adjacent L1 sets, ensuring all elements are AutoLocked.

4.3.1 Eviction Set Splitting

We first eliminate the impact of AutoLock on our analysis of the inclusion policy. A standard AutoLock mechanism chooses L2 lines with no L1 copies to evict over those with L1 copies. Therefore, for an L2 eviction set to evict the target address cached in another core's L1 (hence AutoLocked), we must ensure that the L2 eviction set addresses are also AutoLocked. In this way, the replacement logic is forced to evict AutoLocked lines to service cache fills. The original eviction set generated from §4.2 cannot achieve this, because all eviction set elements are mapped to the same L1 set, so there are always eviction set elements cached only by the L2, as shown in Figure 2 (left). Note, the L1's associativity is less than the L2's associativity (Table 1).

Yet, we notice that the L2 line size in the M1 is 128 Bytes, twice the size of L1 lines. This means that data in one L2 set can reside in two adjacent L1 sets. By toggling bit 6 in half of the eviction set addresses, as shown in Figure 2 (right), those addresses can be moved to the other half of the L2 lines, and their L1 copies are moved to the adjacent L1 set. Now, since all addresses own L1 copies, AutoLock, assuming it exists, can no longer interfere with the L2 eviction.

4.3.2 L2 Inclusion Policy

We use Algorithm 2 to determine the inclusion policy of L2.[3] The algorithm first generates L2-congruent addresses following §4.2, and chooses 1-4 addresses to form a set T. The rest of the addresses, serving as the L2 eviction set of T, undergo eviction set splitting and form S. Hence, all lines in S and T are AutoLocked. The algorithm then traverses T on core 1, and subsequently traverses S on a different core 2 and measures the latency of traversing S.

Figure 3 shows the result of the experiment. This result

[3]We note that the use of 2 cores in this experiment is not fundamental, but was done to match the methodology used in subsequent sections.

Algorithm 2: Code for testing L2's inclusion policy.

Input: T: A set of addresses mapped to the same L2 set
S: A **split** L2 eviction set of T (T, S are in the same L2 set)
Output: Cycles spent by the traversal of S

```
Function Is_Inclusive(T, S):
    [Core 1] traverse T
    [Core 2] t = measure latency of traversing S
    return t
```

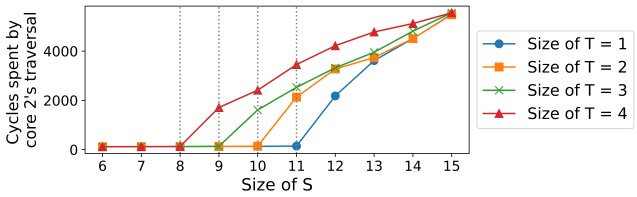


Figure 3: Time spent to traverse the split S when varying the size of S and T in Algorithm 2. This result shows that the L2 cache is inclusive of the L1.

reveals that the L1 cache is inclusive, because, from core 2's perspective, the entire S can be cached in the L1 only if the total size of T and S does not exceed the L2 associativity. In other words, the contention with T in the L2 causes S to be evicted from the L1, which is only possible in an inclusive L2.

4.3.3 AutoLock

Algorithm 3: Code for verifying AutoLock's effect.

Input: `addr`: A randomly selected address
S: A minimal L2 eviction set of `addr` generated by §4.2
Output: Latency of the 2nd access to `addr`

```
Function Check_AutoLock(addr, S):
    Split S into n on one L1 set and |S| − n on the other L1 set
    [Core 1] load [addr]
    [Core 2] traverse S
    [Core 1] t = measure latency of load [addr]
    return t
```

We further use Algorithm 3 to verify that the L2 indeed implements the AutoLock mechanism. The experiment starts by accessing addr, and then traverses its eviction set S on a different core. We explore different ways of splitting S by varying the number of addresses in S to be flipped to the other L1 set. We observe that addr can be evicted when the smaller half of S has at least 3 addresses (meaning 3 or 9 addresses are flipped). This proves the existence of AutoLock: when fewer than 3 eviction set elements reside in an L1 set on core 2, the other L1 set on core 2 can only hold 8 eviction set addresses. Hence, at most 11 addresses out of 12 in the L2 set are AutoLocked, including at most $2+8=10$ eviction set elements plus addr, thus addr is never evicted.

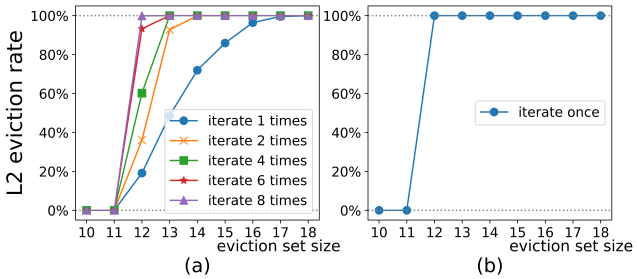


Figure 4: (a) The L2 eviction rate of addr when we run Algorithm 3 but allow Core 2 to traverse S multiple times. (b) The L2 eviction rate of addr when we run Algorithm 3 but traverse S on Core 1 (the same core as load [addr]) instead of Core 2 and only iterate over S once.

> M1's shared L2 is inclusive of private L1s. M1 also employs the AutoLock optimization [64] to prioritize data without L1 copies for eviction from the L2.

4.4 L2 Replacement Policy

Algorithm 3 can also be utilized to investigate the L2 replacement policy. After applying eviction set splitting to S, addr and addresses in S are all AutoLocked thus evictable from the L2. We examine the replacement policy by measuring how difficult it is to evict addr as we vary the size of S and the number of times the traverse function iterates over S.

The L2 eviction rate with different eviction set sizes and iteration counts is displayed in Figure 4 (a). The L2 eviction rate is always 0% when the size of S is less than the L2 associativity (due to AutoLock), and it increases to 100% as S grows. In addition, by iterating over S more times, a smaller S can guarantee to evict addr. For example, iterating over the minimal eviction set at least 8 times guarantees the eviction of addr in practice. These observations imply that a non-LRU replacement policy is being used. In fact, prior studies have shown that a pseudo-random replacement policy is adopted by the L2 cache in previous Apple ARM CPUs [20, 21, 34].

Since the experiment above uses cache lines belonging to different cores, we ask whether the replacement policy behaves differently for lines belonging to the same core. Thus, we change line 4 in Algorithm 3 such that the eviction set S is accessed by the same core as addr. Also, traverse only iterates over S once. The result shown in Figure 4 (b) indicates that even one iteration over the minimal L2 eviction set S is guaranteed to evict addr. This is possible only when an LRU-based policy is used. We make the following observation:

> The L2 replacement policy behaves as LRU when eviction candidates are lines belonging to the same core, and non-LRU (possibly pseudo-random) when lines belonging to different cores can be evicted.

4.5 L2 Cache Set Index Mapping

The L2 eviction set generation technique §4.2 is agnostic about the actual L2 set index mapping function. However, this technique is not applicable when the cache state is measured with ldrex/strex instead of timers, as shown in §5.3. The main reason is due to the fact that ldrex/strex only indicates data residency in L1, as opposed to timers that pinpoint the exact cache level that the data is at. Here, we aim to learn the L2 set index mapping which is later used by S^2C for generating L2 eviction sets.

We reverse-engineer the undocumented L2 set index hash function by inspecting the physical addresses of L2-congruent addresses, which can be retrieved by `/proc/self/pagemap` on Linux. Based on previous research on reverse-engineering Intel's undocumented set/slice mapping function [14, 24, 40], we speculate that on the M1, every L2 set index bit is also computed through a reduction operation with exclusive-or (xor) against a specific set of physical address bits. To verify this, we generate a large number of mutually L2-congruent addresses and compute the xor value of different combinations of physical address bits. When a combination is actually used for computing an L2 set index bit, the xor reduction will produce the same bit value for every L2-congruent physical address. We demonstrate how every L2 set index bit is formed from physical address bits in Figure 5 based on our experimental results.[4] Notice that 11 out of 13 set index bits are directly mapped from huge page offset bits. The other 2 bits are computed over a complex xor operation involving the huge page number bits.

5 S^2C Monitoring a Single Cache Set

We now present a protocol that enables the attacker (the receiver) to use S^2C to monitor a single L2 cache set that the victim may access. §6 will describe how to generalize the protocol to simultaneously monitor multiple L2 sets.

5.1 Attacker Model and Overview

We assume an attacker who co-locates with a victim process on the same Apple M1 processor, and shares the same L2

[4] This pattern resembles the address mapping in Intel CPUs, in which the LLC set index is directly mapped from huge page offset bits (subtracting cache line offset bits) and LLC slice index bits are xor-ed from the high-order page number bits [24, 40]. It is possible that the M1 also divides the L2 cache into as many slices as cores, resulting in a 2-bit L2 slice index, and each slice owns $8192/4 = 2^{11}$ sets.

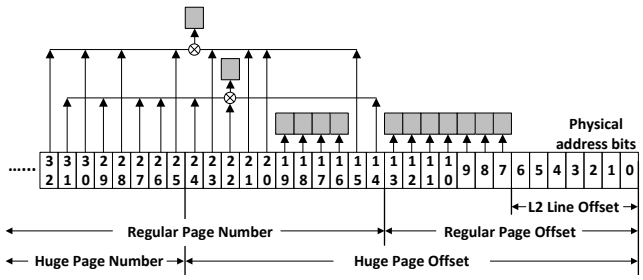


Figure 5: L2 set index bit mapping based on our reverse engineering (§4.5). Eleven L2 set index bits are directly mapped from huge page offset bits. Two L2 set index bits on the top are XOR-ed from multiple regular page number bits (⊗ denotes the XOR operation).

cache. The attacker can run arbitrary unprivileged code, but the attacker has no access to any timing source (such as `PMC0`) and is limited to using a single core. Such an attacker cannot use timing measurement techniques mentioned in §2.1, including the implicit timer since no two attacker-controlled threads can run concurrently. The attacker can allocate huge pages, which is required for generating eviction sets in §5.3.

The attacker's goal is to monitor the victim's memory access patterns, specifically, whether/how the victim accesses a target address that maps to a specific L2 set. As a contention-based cache attack, S^{2}C cannot differentiate victim addresses that are mapped to the same L2 set, as mentioned in §2.1.

Similar to existing cache attacks [10, 15, 16, 43, 45, 68], a complete S^{2}C-based attack has two phases: a preparation phase when the attacker generates the L2 eviction set for the target victim address, and an attack phase when the attacker detects the victim's access to the target address in real-time. We explain the attack phase first in §5.2 assuming minimal L2 eviction sets for the target address are available, and describe how to generate the L2 eviction set using ldrex/strex instead of relying on timing measurements in §5.3.

5.2 Attack Phase

A strex leaks 1-bit of information about whether an attacker-controlled exclusive address is evicted from the L1 cache. To relate this bit to the victim's activities on a single target address, an S^{2}C attacker can simply co-locate the exclusive address with the victim target address in the same L2 set, with additional efforts to ensure that the strex fails if and only if the victim reads/writes to the victim target address.

This strategy faces two unique challenges. First, strex only indicates data's presence in L1, unlike timing measurements which reveal the exact cache level through concrete latency numbers. To avoid false positives, the exclusive address visited by ldrex must remain in the L1 until the expected L2 set contention occurs, which implies a victim's access to the target address. Second, unlike normal Prime+Probe which can measure the access latency to multiple addresses, ldrex/strex only observes one specific address. This necessitates that our attack performs a very delicate balancing act: we must ensure that the exclusive address is not only evictable (not constrained by AutoLock), but that *it* is the line that gets evicted by the victim's access to the target address.

Algorithm 4: How S^{2}C monitors the victim's access to a single target address `addr` using ldrex/strex.

Input: P: exclusive address
S^P: remaining eviction set of `addr` excluding P
Output: Boolean value indicating if `addr` is accessed

1 **Function `Monitor_Single_Addr`** (P, S^P)**:**
2 [attacker on core X] `val` = LDREX $[P]$
3 [attacker on core X] traverse S^P
4 `/* Now` ***P*** `should be the next to evict in L2, but still cached in L1 (i.e. AutoLocked) */`
5 [victim on core Y] may or may not access [`addr`]
6 `/* The access to` [**`addr`**] `should evict` ***P*** `from the L2, and also from L1 due to inclusive cache */`
7 [attacker on core X] `fail` = STREX `val`, $[P]$
8 **return** `fail`

We now explain how S^{2}C addresses both challenges, using Algorithm 4 as a guide. The attacker can obtain an exclusive address P that is L2-congruent with the target address addr, by choosing an arbitrary address from addr's minimal L2 eviction set S. The rest of the eviction set S^P is traversed on the same core after ldrex completes. Importantly, we must apply eviction set splitting (§4.3) to this L2 eviction set S. This guarantees that every L2 eviction set element, including P and all of S^P, are cached in the L1. This addresses the first challenge. For this exact same reason, P and S^P are AutoLocked and occupy the entire L2 set, meaning P can be chosen by the L2 replacement logic, according to the mechanism of AutoLock §4.3.3. Additionally, our study of the L2 replacement policy in §4.4 points out that the L2 uses an LRU-based policy for evicting L2 cache lines belonging to the same core. Since ldrex happens strictly before traversing S^P, the L2 line where P is located will become LRU after line 3 since P and S^P are from the same core. The attacker then waits for the victim's action by spinning a loop a number of times. Whenever addr is accessed, P will be evicted from both L1 and L2, causing the strex on line 7 to fail.

5.3 Preparation Phase

During the preparation phase, S^{2}C generates L2 eviction sets for the target address utilizing ldrex/strex only. As mentioned in §4.2, Vila's algorithm (or similar techniques such as [56]) is widely used by cache attacks due to the weak assumptions it makes on the attacker — no knowledge about the address bits beyond the regular page offset bits is required.

However, we found that using Vila's algorithm (or any similar techniques based on pruning eviction set supersets) by replacing timing measurement with ldrex/strex is unfeasible due to AutoLock. To start, ldrex/strex cannot distinguish

between L1 evictions caused by L1 cache contention and those caused by L2 evictions. Therefore, building the L2 eviction set superset should not use candidate addresses that are L1-congruent with the target address addr. Given addr is AutoLocked and hence can only be evicted from the L2 when the other 11 lines in the same L2 set are also AutoLocked, the traversal of an L2 eviction set superset S can evict addr only if at one point, all 12 L2-congruent addresses in S are cached in L1. This is clearly impossible: they compete for one single L1 set, meaning at most 8 addresses can be AutoLocked with addr in the L2 set.[5] The outcome is that we can never identify an L2 eviction set superset, let alone reduce it to obtain the actual L2 eviction set.

Inspired by previous works [14,27,38], S^2C instead utilizes the reverse-engineering result of the L2 set index mapping and huge pages[6] to directly compute the minimal L2 eviction set (§4.5). Since 11 out of 13 L2 set index bits are huge page offset bits, after allocating a huge page, the attacker can easily identify addresses within this huge page that share those 11 set index bits as the target address. Although the remaining two bits cannot be determined due to the unknown huge page number, we can easily determine whether two arbitrary addresses within the huge page share the same value for these two bits, because the huge page number is identical.

With this observation, our L2 eviction set generation works as follows. Given a target address, we allocate one huge page, and collect all $2^7 = 128$ addresses on this page that share the same regular page offset and address bits [19:16] as the target address. All these address bits except the L2 line offset bits are required to match the target address to achieve L2-congruence. Next, we group the 128 addresses into four groups, such that addresses within each group share the same two XOR-ed bits. Since the huge page number of these addresses as well as the target address is unknown, we cannot determine which address group has the exact same L2 set index as the target address, but it is guaranteed that one of these four address groups will be an L2 eviction set of the target address.

We leverage Algorithm 4 to identify which group is actually the L2 eviction set. For each group, we choose 12 addresses from the total 32 addresses as the candidate minimal eviction set, and apply eviction set splitting so that those 12 addresses are distributed to two sibling L1 sets evenly. As for Algorithm 4, one address is chosen as the address for ldrex/strex, and the remaining are traversed after ldrex. Unlike the attack, when testing eviction sets, the attacker must trigger access to the target address (which could be done by a victim-provided API call that is known to access the target address). Only when the tested address group is L2-congruent with the target address will the strex fail.

[5] Even if the attacker owns multiple cores to traverse the candidate set separately, it is still infeasible for a candidate L2 eviction set S to evict addr: the L2-congruent addresses constitute only around $1/2^6 \approx 1.6\%$ of S according to Figure 5. Therefore, it is almost impossible to see 12 L2-congruent addresses cached in the L1.

[6] Although allocating huge pages requires only user-level privilege, it may not be available when the attacker is limited to a sandboxed environment, e.g., browsers. In those scenarios, the attacker cannot proactively allocate huge pages via system calls. Instead, the attacker must rely on the runtime allocating huge pages automatically, e.g., via OS features such as Transparent Huge Pages in Linux.

6 S^2C Monitoring Multiple Cache Sets

We now generalize the protocol from §5 to enable the attacker to simultaneously monitor multiple victim L2 cache sets. The attacker model is otherwise the same as that presented in §5.1: the attacker is unprivileged, runs on a single thread, etc. As with §5, we do not require modifications in the victim's code.

The challenge here is that ldrex/strex only allow the single-threaded attacker to monitor evictions on a single address/cache line at a time. To work around this limitation, the insight is to view ldrex/strex *as a general-purpose single-bit communication channel* that can communicate the result of an arbitrary 1-bit function computed in micro-architectural space. With this in mind, we construct a micro-architectural weird circuit (μWC) [12] that computes the logical-OR of whether the victim accessed at least one of several attacker-specified L2 sets—and communicates the 1-bit result of this logical-OR through ldrex/strex to the attacker's architectural state, namely the result of strex.

We remark that while weird circuit constructions are not the main focus of the paper, our weird circuit is relatively simple conceptually compared to the original proposals in [12], and may be of independent interest. In particular, our construction relies solely on out-of-order execution, as opposed to some form of speculative/transient execution (e.g., speculative instruction execution, Intel's TSX). We also remark that while we compute logical-OR due its useful semantics, other functions are of course possible (e.g., one to compute a hop in a binary search to *localize* which victim cache set was accessed). We leave such investigations to future work.

6.1 μWC Construction

We explain the μWC assuming the attacker wishes to monitor two victim target addresses a and b located at L2 sets A and B for simplicity, and generalize to monitoring N L2 sets $A_0, A_1, \ldots, A_{N-1}$ at the end.

To start, the attacker uses the procedure from §5.3 to construct minimal L2 eviction sets for target addresses a and b, and also a third eviction set for another address x allocated by the attacker itself. The attacker ensures that x is located at an L2 set X different from A and B, and X is not used by the victim. The attacker further builds a linked list $Pa \rightarrow Pb \rightarrow Px$, where Pa, Pb, Px are addresses randomly chosen from the eviction sets generated for a, b, and x, respectively.

The attacker is interested in whether the victim accesses a line in sets A or B. At a high level, it can infer activity on these sets by observing the eviction of Px with ldrex/strex, and using out-of-order execution to create a race between a)

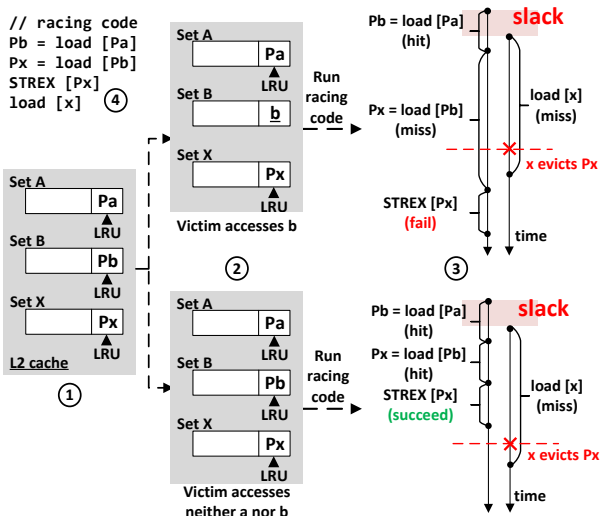


Figure 6: An example of how S^{2}C constructs a μWC to monitor victim addresses (called a and b) that map to L2 cache sets A and B, respectively. See text in §6.1 for a detailed walkthrough.

traversing the linked list $Pa \rightarrow Pb \rightarrow Px$ and b) accessing x. Depending on the outcome of this race, Px could be evicted by x when it is accessed by the attacker's strex, which indicates whether the victim displaced data in sets A or B.

In more detail: In the "Prime" step (Figure 6 ①), the attacker uses the techniques in §5.2 to bring Pa, Pb, Px into the cache, position each in the LRU position of each respective L2 set and ensure that each is evictable (using eviction set splitting; §4.3.1). It further evicts x from all levels of cache and monitors Px using ldrex. It then waits for the victim to make an access (see Figure 6 ②), same as in §5. In the "Probe" step, the attacker begins the race (Figure 6 ③) by simultaneously a) traversing $Pa \rightarrow Pb \rightarrow Px$ and b) making an access to x. x will always result in a miss. It accesses Pa, Pb with normal loads and accesses Px using strex. There are two possible outcomes, depending on the victim's access pattern:

- If the victim accessed neither A nor B (Figure 6 ②-③, bottom), traversing $Pa \rightarrow Pb \rightarrow Px$ will result in all hits and complete *before* x fills the cache. Since Px will still be cached, strex returns 0 (success).

- Otherwise (Figure 6 ②-③, top), traversing $Pa \rightarrow Pb \rightarrow Px$ will result in at least one miss and complete *after* x fills the cache. Since Px will be evicted (by x), strex returns 1 (fail).

Monitoring N victim addresses. The above generalizes to monitoring N victim addresses $a_0, a_1, \ldots, a_{N-1}$ located at different L2 sets using N eviction sets, a separate set X (which serves the same function as before) and a linked list that traverses $P_{A_0} \rightarrow P_{A_1} \rightarrow \cdots \rightarrow P_{A_{N-1}} \rightarrow Px$.

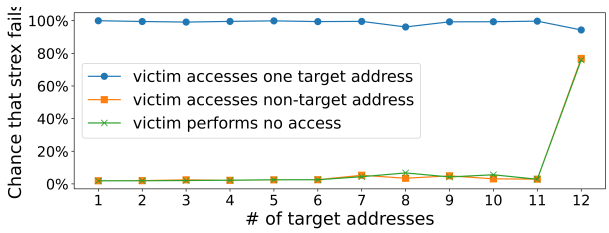


Figure 7: The chance that strex fails when the victim accesses different addresses, given S^{2}C monitoring multiple target addresses.

Implementation considerations. Our implementation matches closely with the above description. That said, we needed to place load[x] after the strex in program order (Figure 6 ④). This is because strex is not performed until it reaches the head of the reorder buffer, and thus would not execute until load[x] is completed if the load was placed before it. Another important factor is that, as N increases, it is necessary to delay the load to x to ensure that traversing the linked list is faster than accessing x, if all linked list accesses are hits. This delay, called 'slack' in Figure 6, is implemented by spinning in a loop a specific number of times.

7 Evaluation

Our evaluation is performed on an M1 Mac Mini, running Asahi Linux (Linux version 6.1.0). We cannot conduct the full evaluation on MacOS since the M1 version of MacOS does not support huge pages. Following the attacker model described in §5.1, the S^{2}C attacker is single-threaded and does not use timers.

7.1 Monitoring Multiple Cache Sets

We now evaluate S^{2}C's ability to simultaneously monitor multiple cache sets. Here the victim may access a set of addresses that map to different L2 cache sets, and the attacker uses the method described in §6 to detect accesses to those sets. Ideally, any victim access to those sets should result in a 100% strex fail rate; if the victim accesses none of the aforementioned sets, we expect a 0% fail rate, i.e., the difference in the strex fail rate should be close to 100%.

Figure 7 shows the probability that the attacker's strex fails when the victim 1) accesses one target address, 2) performs no memory accesses, and 3) accesses addresses belonging to other L2 sets. When monitoring less than 12 addresses, we can always find a suitable slack so that the difference in the strex fail rates between the victim accessing the target address versus accessing no/other addresses is close to 100% (the difference is at least 93%). This showcases S^{2}C's effectiveness in monitoring up to 11 L2 sets. For more than 12 target addresses, we cannot find a slack value that achieves a favorable difference in the strex fail rate. This results in

significant false positives, as shown in Figure 7.

7.2 Covert Channel

S^2C, like previous cache side channels, can be used to establish cross-process covert channels. However, as the strex output only conveys 1-bit of information, the channel only transmits 1-bit at a time. Another challenge in building covert channels with S^2C is that it requires synchronization between two processes without relying on timing measurements, which are used by prior cache attacks [15, 38, 45, 72].

The transmission of each bit consists of two stages: the standby stage and the transmission stage. Both stages employ Algorithm 4, but in opposite directions. The sender and the receiver each designate an address located in different L2 sets, and both generate an L2 eviction set for the other party's address. The address owned by the sender, dubbed $\texttt{addr}_{\texttt{trans}}$, is used in the transmission stage. The receiver prepares an exclusive address (to be monitored by strex) which is L2-congruent to $\texttt{addr}_{\texttt{trans}}$, and uses it along with the remaining eviction set to detect the sender's accesses to $\texttt{addr}_{\texttt{trans}}$. To transmit 1 bit, the sender accesses $\texttt{addr}_{\texttt{trans}}$ (and otherwise makes no access). The receiver pauses by iterating over a busy-waiting loop eight thousand times to give the sender sufficient time to access the targeted L2 set, and then checks whether $\texttt{addr}_{\texttt{trans}}$ was accessed using strex.

During the standby stage, the sender waits for a signal from the receiver indicating that the receiver is ready. Likewise, this is achieved by the sender monitoring the receiver's access to the address $\texttt{addr}_{\texttt{ready}}$. Different from the transmission stage, the receiver will always access $\texttt{addr}_{\texttt{ready}}$ to indicate readiness. The sender repeatedly calls Algorithm 4 until an access to $\texttt{addr}_{\texttt{ready}}$ is observed, at which point the sender can proceed to the transmission stage.

After tuning the covert channel code, we achieve a bandwidth of approximately 185 Kbits per second with a 98.5% accuracy. The small error rate is due to the sender's access to $\texttt{addr}_{\texttt{trans}}$ overlapping with the receiver's ldrex/strex, creating a *blindspot* that has been studied in previous works such as Prime+Scope [45]. Since our synchronization also relies on S^2C instead of a precise timing source, the blindspot can also cause de-synchronization. For example, if the receiver sends a ready signal that is missed by the sender, the sender will get stuck waiting. In the next round, the receiver will interpret that as the sender transmitting 0, at which point it will send another ready signal (which, hopefully, the receiver will now see). This de-synchronization has a 0.2% chance of affecting each bit transmission.

Comparison with Prime+Probe We also implement a similar 1-bit covert channel using Prime+Probe, after enabling the cycle count register from kernel space. For the Prime+Probe covert channel, the receiver owns the address $\texttt{addr}_{\texttt{trans}}$, and the sender controls the L2 eviction set of $\texttt{addr}_{\texttt{trans}}$. To transmit a bit, the receiver first accesses $\texttt{addr}_{\texttt{trans}}$, and the sender traverses the eviction set when the transmitted bit is 1 (and otherwise does not traverse the eviction set). The receiver pauses for a certain duration to allow the eviction set traversal to finish, and then times the access to $\texttt{addr}_{\texttt{trans}}$ to determine whether it has been evicted. Unlike S^2C which relies on cache contention during the standby stage for synchronization, the Prime+Probe covert channel utilizes the timer directly for synchronization. This not only reduces the activities during each transmission, leading to a significant increase in bandwidth, but also ensures precise synchronization between the attacker and the sender, eliminating the problem of de-synchronization. This 1-bit Prime+Probe covert channel is capable of transmitting approximately 382 Kbit per second with over 99% accuracy.

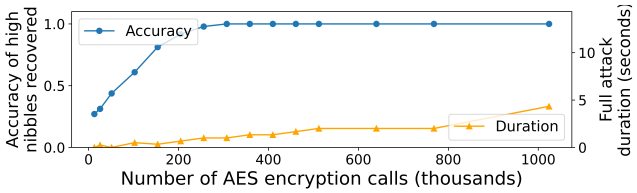
7.3 Attacking T-table AES

We now demonstrate how S^2C can be used to perform full key extraction on T-table AES, based on the classical chosen-plaintext attack due to Osvik et. al [43].

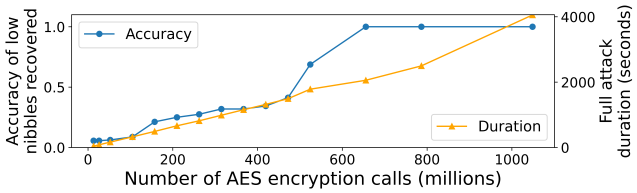
T-table AES is a popular benchmark for evaluating cache side-channel attacks because it creates secret key-dependent T-table access patterns, which can be used to infer information about the secret key [10, 15, 43, 45]. We adapt the attack by Osvik et. al [43], which proceeds in two steps: targeting high-key nibbles and low-key nibbles of key bytes, respectively. Specifically, the first round of T-table AES performs bitwise XOR between the plaintext and the private key, and the output bytes are used directly as indices for T-table lookups. To exploit this behavior, the attacker repeatedly interacts with the victim. In each interaction, the attacker tries to guess the high nibble of one key byte by creating a specific plaintext. When the guess is correct, the XOR produces an index value $< 2^4$, thereby creating an observable access to a specific cache line where the looked-up T-table is based. While this cache line may also be accessed by other rounds, it is guaranteed to be accessed when the high nibble from the plaintext byte and the target key byte match. Repeating for each key byte, this allows the attacker to recover the high nibble of each key byte.

The low nibbles can be retrieved similarly by exploiting the second round. In the second round, every T-table access index depends on four distinct key bytes, instead of one key byte in the first round. Since the high key nibbles are known, the attacker can guess possible values for the four low nibbles corresponding to the four key bytes used by the T-table index. It validates these guesses in the same way as before: by submitting plaintexts and monitoring whether the T-table target cache line is accessed during the AES operation. This process can be repeated to learn the values of nibbles in other sets of four key bytes.

Here we show that S^2C can leverage the above attack technique to infer the full 16-byte AES key. Our experiment



(a) First-round attack result.



(b) Second-round attack result.

Figure 8: Accuracy of recovered high nibbles and the attack duration of both the first-round and the second-round attack, as we increase the number of total AES calls for the full first-/second-round attack. The kink in the accuracy plot in (b) is due to measurement noise.

uses OpenSSL's T-table AES implementation [1]. The victim process maintains a secret key and exposes an AES encryption/decryption API to the attacker. The attacker process performs the chosen-plaintext attack and leverages S^2C (instead of Prime+Probe as in the original attack [43]) to monitor accesses to targeted cache lines.

Figure 8 shows the proportion of nibbles that can be accurately recovered in the first-round and the second-round attack, as we increase the number of plaintext samples used for testing each possible nibble value (or four nibble values in the second-round attack). Each data point is averaged over ten independent attacks, where each attack performs one first-round and one second-round attack to leak one randomly-generated private key. The figure also shows the attack latency as a function of the number of AES calls. The high nibbles can be recovered with 100% accuracy with around 300K encryption calls, which takes around 1 second. However, to recover the low nibbles with 100% accuracy, we need around 700M encryption calls, which is roughly 40 minutes. Notice that recovering low nibbles requires significantly more time than high nibbles because, in the second-round attack, the attacker must guess four nibbles together, and measuring a cache access only indicates if all four nibbles are guessed correctly.[7] This difference is mentioned by the original attack [43].

Since most recent cache attacks only evaluate the first-round attack [10, 15, 30, 45], we can compare the effectiveness of our version of the first-round attack to them. For instance, Flush+Flush [15] shows 250 encryptions are required to recover all high nibbles, whereas Prime+Scope [45] reports a similar number (around 200). Therefore, S^2C demonstrates a similar capability in retrieving AES key bytes albeit using a different cache measurement technique and targeting Apple CPUs.

[7] To illustrate, assume N samples are needed to verify each guess. The first-round attack, therefore, requires 16 (16 high nibbles) * 16 (16 possible values for each high nibble) * N = 256 * N AES calls, whereas the second-round attack requires 4 (4 four-nibble combinations) * 16^4 (guess values for 4 low nibbles) * N = 262,144 * N AES calls.

8 Discussion

8.1 Impact on Other Processors

Load-Linked/Store-Conditional is an ISA-agnostic primitive for performing efficient mutual exclusion and synchronization. We now discuss its support in other existing ISAs, and where we believe those implementations may enable S^2C-based attacks.

CISC/x86. CISC architectures such as x86 do not support LL/SC as part of their ISAs. We have further performed experiments to check whether native x86 hardware atomics (`cmpxchg`) are implemented using LL/SC-like microcode under the hood and haven't found evidence to support this theory.

ARM. In 2022, Apple introduced the M2 which continues using the same ARMv8-A ISA as M1. We are able to reproduce the same attack primitive that we explained in §3 on M2 CPUs. Future work is needed to reverse engineer the M2's cache configuration (akin to that in §4) to enable S^2C-based attacks (i.e., across cores). It is possible that other ARM CPUs follow the LL/SC semantics specified in ARM's manual, making them vulnerable to the base S^2C mechanism. But more reverse engineering is needed to confirm on which ARM CPUs this holds.

RISC-V. RISC-V is a relatively new and rapidly-growing RISC architecture. Different from ARM, RISC-V recognizes that allowing store-conditional to fail on cache evictions might impede LL/SC progress indefinitely. Therefore, the official RISC-V manual suggests that "reservations (exclusive addresses) are tracked independently of evictions from any shared cache" [62], which, if implemented in real life, can effectively mitigate S^2C.

MIPS. MIPS is an older RISC ISA. The MIPS manual suggests that the base observation in S^2C may apply. We quote: "...load or store may cause a cache eviction between the LL and SC that results in SC failure" [2].

8.2 Mitigations

Restricting access to the attack primitive is a straightforward yet effective method for mitigating many side-channel attacks [25, 32, 35, 39, 70]. Because LL/SC is only leveraged

by the attacker and not required to be used by the victim, one would seemingly need to summarily disallow LL/SC. However, LL/SC are basic instructions that are impractical to disable. Given this, we propose several mitigation strategies.

Changing Exclusive Monitor Implementation. Because the exclusive monitor only tracks addresses in the L1 cache, we speculate that the exclusive monitor is implemented by piggybacking on the cache coherence protocol. The implementation may use a dedicated coherence state for exclusive addresses accessed by LL instructions. The behavior of SC aligns with this design: an SC succeeds when the target address is in the new state; and like normal stores, an SC invalidates the address in other private L1 caches, forcing other threads to lose their exclusiveness to that address. To patch this vulnerable implementation, the safe exclusive monitor should keep track of exclusive addresses independent of the location of the addresses in the cache hierarchy.

Software Mitigations for Cache Side-Channels. Mitigating cache side-channel attacks have been an important topic in side-channel research. Software developers generally use constant-time programming to eliminate secret-dependent memory access patterns [8, 41, 42, 48, 51, 69, 71]. Constant-time programming has been applied to many modern cryptographic libraries for protecting critical assets such as keys [18,44]. However, many crytographic libraries, as well as general-purpose programming still opt out of the constant-time property in favor of better performance [28].

Hardware Mitigations for Cache Side-Channels. Researchers have also proposed hardware-based mitigations for defeating cache side channels, most of which are based on cache partitioning or randomization. Cache partitioning splits the cache into partitions, and each partition can be used by a security domain without any interference from other domains [11, 37, 60, 61]. Cache randomization introduces randomness into the cache set mappings, hindering the attacker from creating cache contention with victim lines [46,47,57,63]. Since LL/SC is a contention-based attack, both of these approaches would apply in principle.

Finally, disabling huge pages does mitigate our current attack by preventing eviction set generation. For this reason, the current M1's MacOS is immune to S^2C in practice. However, previous MacOS versions on Intel CPUs do support huge pages. Given that the M1 CPU natively supports huge pages, future MacOS versions on Apple CPUs may re-adopt huge pages, making it once again susceptible.

9 Related Works

Cache attacks The attack procedure in S^2C resembles Prime+Probe [43]: the attacker establishes its lines in specific states (e.g., cached), and later watches for cache evictions to deduce the victim's behavior. Refresh+Reload [7] instead observes the LRU state changes to the attacker's lines without relying on the victim to evict the attacker's lines, making the attack more stealthy. The same idea can be applied to S^2C. Prime+Scope [45] improves Prime+Probe by making the probe step a single memory access and being windowless. Prime+Scope is special because it requires inducing private L1 evictions caused by shared cache evictions, which is also required by S^2C. However, Prime+Scope targets x86 CPUs that do not implement AutoLock like ARM.

Timer-less methods to monitor μarch state changes. Most cache attacks observe cache states using timers (§2.1). A recent attack exploits the implementation of umwait/umonitor instructions on x86 CPUs to set up a countdown clock on a specific address [70]. If a (potentially transient) write occurs within the countdown period, a thread is woken up and the carry bit (CF) set; otherwise (the countdown expires) the thread is woken up with a carry bit cleared. Both the (transient) write $\rightarrow$ CF action in that work and the cache eviction $\rightarrow$ SC fail action in our work are similar conceptually (although act on different micro-architectural state changes and impact different platforms). We note that umwait interacts with an explicit timer (the timestamp counter) to control timeout; the analog to this in our attacks is how the receiver must spin in a loop to wait for the victim to make an access (which is also conceptually a timer, albeit an implicit one).

As mentioned in §2.1, uncacheable memory [17] and hardware transactional memory [10, 30] are the only other primitives that can be used to observe cache state without relying on timing measurements. Notice that LL/SC is similar to a hardware transaction: LL/SC collectively completes a task that may fail (with some kind of feedback) and, just like transactions, can fail due to interactions with sibling threads or for purely micro-architectural reasons (cache evictions).

Finally, our μWC construction is similar to concurrent work on racing gadgets [66], which serve to amplify small timing differences so as to be detectable by low-resolution clocks.

10 Conclusion

This paper proposes Synchronization Storage Channels (S^2C), the first timer-less cross-core cache attack that exploits Load-Linked/Store-Conditional (LL/SC) instructions. The key insight is that the implementation of LL/SC on the Apple M1 enables direct observation of cache activities, i.e., whether the address tracked by LL/SC has been evicted from the L1. With several novel techniques to circumvent limitations in the single-address-observation semantics of LL/SC, we show how S^2C achieves similar attack capability as prior contention-based cache attacks but without the dependence on timing measurements.

Acknowledgments

This work was funded by the NSF under grants 1816282, 1954521, 1942888, and 2154183, as well as by an Intel RARE grant. We would like to thank the anonymous shepherd and reviewers for their insightful comments during the review process, which helped to significantly strengthen the paper.

References

[1] Openssl. https://www.openssl.org/.

[2] *MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual*. 2016.

[3] Apple. Energy efficiency guide for mac apps: Prioritize work at the task level. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx/PrioritizeWorkAtTheTaskLevel.html, 2020.

[4] ARM. Exclusive monitor system location. https://developer.arm.com/documentation/den0024/a/Multi-core-processors/Multi-processing-systems/Exclusive-monitor-system-location, 2023.

[5] ARM. Load-exclusive and store-exclusive usage restrictions. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Application-Level-Memory-Model/Synchronization-and-semaphores/Load-Exclusive-and-Store-Exclusive-usage-restrictions, 2023.

[6] ARM. Synchronization. https://developer.arm.com/documentation/den0024/a/Multi-core-processors/Multi-processing-systems/Synchronization, 2023.

[7] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. Reload+refresh: Abusing cache replacement policies to perform stealthy cache attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1967–1984, 2020.

[8] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. Fact: A flexible, constant-time programming language. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 69–76. IEEE, 2017.

[9] Shuwen Deng, Nikolay Matyunin, Wenjie Xiong, Stefan Katzenbeisser, and Jakub Szefer. Evaluation of cache attacks on arm processors and secure caches. *IEEE Transactions on Computers*, 71(9):2248–2262, 2021.

[10] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel tsx. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, 2017.

[11] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–21, 2012.

[12] Dmitry Evtyushkin, Thomas Benjamin, Jesse Elwell, Jeffrey A Eitel, Angelo Sapello, and Abhrajit Ghosh. Computing with time: Microarchitectural weird machines. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 758–772, 2021.

[13] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. Autolock: Why cache attacks on arm are harder than you think. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1075–1091, 2017.

[14] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment-Volume 9721*, pages 300–321, 2016.

[15] 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.

[16] 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.

[17] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 38–55. IEEE, 2016.

[18] Shay Gueron. Efficient software implementations of modular exponentiation. *Cryptology EPrint Archive*, 2011.

[19] 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.

[20] Gregor Haas, Seetal Potluri, and Aydin Aysu. itimed: Cache attacks on the apple a10 fusion soc. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 80–90. IEEE, 2021.

[21] Maynard Handley. M1 explained. https://github.com/name99-org/AArch64-Explore, 2022.

[22] Lorenz Hetterich and Michael Schwarz. Branch different-spectre attacks on apple silicon. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 19th International Conference, DIMVA 2022, Cagliari, Italy, June 29–July 1, 2022, Proceedings*, pages 116–135. Springer, 2022.

[23] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1(3-4):233–254, 1992.

[24] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[25] Intel. Performance monitoring impact of intel transactional synchronization extension memory ordering issue. 2021.

[26] Intel. Intel transactional synchronization extensions (intel® tsx) memory and performance monitoring update for intel processors. https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html, 2022.

[27] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604. IEEE, 2015.

[28] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "they're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 632–649. IEEE, 2022.

[29] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[30] Sowoong Kim, Myeonggyun Han, and Woongki Baek. Dprime+dabort: A high-precision and timer-free directory-based side-channel attack in non-inclusive cache hierarchies using intel tsx. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 67–81. IEEE, 2022.

[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[32] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 463–480, 2016.

[33] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Netcat: Practical cache attacks from the network. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 20–38. IEEE, 2020.

[34] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[35] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. Platypus: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 355–371. IEEE, 2021.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. Meltdown: Reading kernel memory from user space. *Communications of the ACM*, 63(6):46–56, 2020.

[37] 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.

[38] 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.

[39] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129. IEEE, 2012.

[40] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Research in Attacks,*

Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18, pages 48–65. Springer, 2015.

[41] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296. IEEE, 2018.

[42] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, volume 16, pages 10–12, 2016.

[43] 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.

[44] Thomas Pornin. Bearssl. https://www.bearssl.org/constanttime.html.

[45] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+scope: Overcoming the observer effect for high-precision cache contention attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2906–2920, 2021.

[46] 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.

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

[48] 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.

[49] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. Pacman: attacking arm pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 685–698, 2022.

[50] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212, 2009.

[51] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. *Cryptology ePrint Archive*, 2017.

[52] Michael Schwarz, Moritz Lipp, and Daniel Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *The Network and Distributed System Security Symposium (NDSS)*, volume 18, page 12, 2018.

[53] 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.

[54] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+probe 1, javascript 0: Overcoming browser-based side-channel defenses. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2863–2880, 2021.

[55] Anatoly Shusterman, Zohar Avraham, Eliezer Croitoru, Yarden Haskal, Lachlan Kang, Dvir Levi, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Website fingerprinting through the cache occupancy channel and its real world practicality. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2042–2060, 2020.

[56] Wei Song and Peng Liu. Dynamically finding minimal eviction sets can be quicker than you think for side-channel attacks against the llc. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 427–442, 2019.

[57] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. Phantomcache: Obfuscating cache conflicts with localized randomization. In *The Network and Distributed System Security Symposium (NDSS)*, 2020.

[58] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[59] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54. IEEE, 2019.

[60] 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*, pages 1–6, 2016.

[61] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 494–505, 2007.

[62] Andrew Waterman and Krste Asanovic, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, 2019.

[63] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *USENIX Security Symposium*, pages 675–692, 2019.

[64] Barry Duane Williamson. Line allocation in multi-level hierarchical data stores, 2012. US Patent 8,271,733.

[65] John C Wray. An analysis of covert timing channels. *Journal of Computer Security*, 1(3-4):219–232, 1992.

[66] Haocheng Xiao and Sam Ainsworth. Hacky racers: Exploiting instruction-level parallelism to generate stealthy fine-grained timers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 354–369, 2023.

[67] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn dnn architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2003–2020, 2020.

[68] 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.

[69] Jiyong Yu, Lucas Hsiung, Mohamad El'Hajj, and Christopher W Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *The Network and Distributed System Security Symposium (NDSS)*, 2019.

[70] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (m) wait for it: Bridging the gap between microarchitectural and architectural side channels. In *USENIX Security*, 2023.

[71] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: an oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, pages 283–298, 2017.

[72] Wu Zhenyu, Xu Zhang, and H Wang. Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *USENIX Security Symposium*, pages 159–173, 2012.