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’ 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 (S²C), 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, S²C aims to build a cross-core side channel to monitor architectural state. At a high level, S²C aims to build a cross-core side channel to monitor 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 (S²C).

- We identify that LL/SC serves as a direct and precise architectural observation channel to monitor microarchitectural events (L1 evictions).

- We conduct a detailed reverse-engineering of the M1’s L2 caches, which is necessary to exploit S²C 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 S²C 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/FPS-G-UIUC/S²C.

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

**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].

### 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.

<table>
<thead>
<tr>
<th></th>
<th>Level</th>
<th>Ways</th>
<th>Sets</th>
<th>Line Size</th>
<th>Total size</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-core</td>
<td>L1D</td>
<td>8</td>
<td>256</td>
<td>64 B</td>
<td>128 KB</td>
</tr>
<tr>
<td></td>
<td>L2</td>
<td>12</td>
<td>8192</td>
<td>128 B</td>
<td>12 MB</td>
</tr>
<tr>
<td>E-core</td>
<td>L1D</td>
<td>8</td>
<td>128</td>
<td>64 B</td>
<td>64 KB</td>
</tr>
<tr>
<td></td>
<td>L2</td>
<td>16</td>
<td>2048</td>
<td>128 B</td>
<td>4 MB</td>
</tr>
</tbody>
</table>

Table 1: Apple M1 cache parameters (from Table 2 of [49])
will succeed, forcing all other threads to lose their exclusive with its address for every core. This ensures that when multi-
implementation may piggyback on top of the coherence protocol, (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 S²C 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.
failures. We also conduct experiments to verify that the be-

Algorithm 1: Code for testing the correlation between

<table>
<thead>
<tr>
<th>Algorithm 1: Code for testing the correlation between strex failures and L1 evictions of the exclusive address.</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Input:} addr: a selected target exclusive address</td>
</tr>
<tr>
<td>1 for i = 1 to 1000 do</td>
</tr>
<tr>
<td>2 \hspace{1em} Generate a set of random addresses $S$</td>
</tr>
<tr>
<td>3 \hspace{1em} val = LDREX[addr]</td>
</tr>
<tr>
<td>4 \hspace{1em} traverse $S$ // may or may not evict addr from L1</td>
</tr>
<tr>
<td>5 \hspace{1em} latency = measure latency of load[addr]</td>
</tr>
<tr>
<td>6 \hspace{1em} fail = STREX val,[addr]</td>
</tr>
<tr>
<td>7 \hspace{1em} evicted_from_L1 = latency &gt; L1_Latency</td>
</tr>
<tr>
<td>8 \hspace{1em} print fail, evicted_from_L1</td>
</tr>
<tr>
<td>9 // An example of counting the output:</td>
</tr>
<tr>
<td>10 // fail = True.evicted_from_L1 = True: 518</td>
</tr>
<tr>
<td>11 // fail = False.evicted_from_L1 = False: 0</td>
</tr>
<tr>
<td>12 // fail = False.evicted_from_L1 = True: 482</td>
</tr>
<tr>
<td>13 \end{algorithm}</td>
</tr>
</tbody>
</table>

We test the above code on both P-cores and E-cores, by con-

3.3 Result and Takeaway

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 ad-

havior 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 al-

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 pri-

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\textsuperscript{2}C 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

---

3We 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 $S$

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

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

1 Function Check_AutoLock (addr, $S$):
2 Split $S$ into two L1 set and $|S| − n$ on the other L1 set
3 [Core 1] load [addr]
4 [Core 2] traverse $S$
5 [Core 1] $t =$ measure latency of load [addr]
6 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 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.

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 $S$

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

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

1 Function Check_AutoLock (addr, $S$):
2 Split $S$ into two L1 set and $|S| − n$ on the other L1 set
3 [Core 1] load [addr]
4 [Core 2] traverse $S$
5 [Core 1] $t =$ measure latency of load [addr]
6 return $t$
we change line 4 in Algorithm 3 such that the eviction set Algorithm 3 can also be utilized to investigate the L2 replace-
guaranteed to evict addr 
addr 
S 
S
is used. We make the following observation:
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 S²C 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.

4.4 L2 Replacement Policy
Algorithm 3 can also be utilized to investigate the L2 replace-
resulting in a 2-bit L2 slice index, and each slice owns 8192/4 = 211 sets.
Algorithm 3 but traverse S on Core 1 (the same core as load [addr]) instead of Core 2 and only iterate over S once.
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.

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.

4.5 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 S²C 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.
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 S²C Monitoring a Single Cache Set
We now present a protocol that enables the attacker (the receiver) to use S²C 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

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 replace-
resulting in a 2-bit L2 slice index, and each slice owns 8192/4 = 211 sets.
Algorithm 3 but traverse S on Core 1 (the same core as load [addr]) instead of Core 2 and only iterate over S once.
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.

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 replace-
resulting in a 2-bit L2 slice index, and each slice owns 8192/4 = 211 sets.
Algorithm 3 but traverse S on Core 1 (the same core as load [addr]) instead of Core 2 and only iterate over S once.
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.

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.
We explain the attack phase first in §5.2 assuming minimal L2 which reveal the exact cache level through concrete latency

\[ \text{strex} \]

To relate this bit to the victim’s activities on a single target (§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 (\( \otimes \) denotes the XOR operation).

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, \( S^2 \text{C} \) 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 \( S^2 \text{C} \)-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 \( S^2 \text{C} \) 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.

\begin{algorithm}
\textbf{Algorithm 4:} How \( S^2 \text{C} \) monitors the victim’s access to a single target address \( addr \) using ldrex/strex.
\begin{enumerate}
\item \textbf{Function Monitor Single Addr} \((P, S^p)\):
\item [attacker on core X] \( val = LDREX [P] \)
\item [attacker on core X] traverse \( S^p \)
\item /* Now \( P \) should be the next to evict in L2, but
still cached in L1 (i.e. AutoLocked) */
\item [victim on core Y] may or may not access \( addr \)
\item /* The access to \( addr \) should evict \( P \) from
the L2, and also from L1 due to inclusive cache */
\item [attacker on core X] \( fail = STREX \ val, [P] \)
\item return \( fail \)
\end{enumerate}
\end{algorithm}

We now explain how \( S^2 \text{C} \) 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 evict 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, \( S^2 \text{C} \) 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. 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], S2C instead utilizes the reverse-engineering result of the L2 set index mapping and huge pages 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.

6 S2C 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)
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 ②), 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 6: An example of how \(S^2C\) 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.](image)

![Figure 7: The chance that strex fails when the victim accesses different addresses, given \(S^2C\) monitoring multiple target addresses.](image)

### 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 \(S^2C\) attacker is single-threaded and does not use timers.

#### 7.1 Monitoring Multiple Cache Sets

We now evaluate \(S^2C\)’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 \(S^2C\)’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 addr_{trans}, is used in the transmission stage. The receiver prepares an exclusive address (to be monitored by strex) which is L2-congruent to addr_{trans}, and uses it along with the remaining eviction set to detect the sender’s accesses to addr_{trans}. To transmit 1 bit, the sender accesses addr_{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 addr_{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 addr_{ready}. Different from the transmission stage, the receiver will always access addr_{ready} to indicate readiness. The sender repeatedly calls Algorithm 4 until an access to addr_{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 addr_{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 addr_{trans}, and the sender controls the L2 eviction set of addr_{trans}. To transmit a bit, the receiver first accesses addr_{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 addr_{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
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.

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.

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.

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 cryptographic 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 \(\mu\)arch state changes.** Most cache attacks observe cache states using timers \((\S\text{2.1})\). 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 \(\S\text{2.1}\), 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 \(\muWC\) 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


[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,


[60] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. Secdep: secure


