

# PRIME+RETOUCH: When Cache is Locked and Leaked

Jaehyuk Lee  
Georgia Institute of Technology  
jaehyuk@gatech.edu

Fan Sang  
Georgia Institute of Technology  
fsang@gatech.edu

Taesoo Kim  
Georgia Institute of Technology  
taesoo@gatech.edu

**Abstract**—Caches on the modern commodity CPUs have become one of the major sources of side-channel leakages and been abused as a new attack vector. To thwart the cache-based side-channel attacks, two types of countermeasures have been proposed: detection-based ones that limit the amount of microarchitectural traces an attacker can leave, and cache prefetching-and-locking techniques that claim to prevent such leakage by disallowing evictions on sensitive data. In this paper, we present the PRIME+RETOUCH attack that completely bypasses these defense schemes by accurately inferring the cache activities with the metadata of the cache replacement policy. PRIME+RETOUCH has three noticeable properties: 1) it incurs no eviction on the victim’s data, allowing us to bypass the two known mitigation schemes, 2) it requires minimal synchronization of only one memory access to the attacker’s pre-primed cache lines, and 3) it leaks data via non-shared memory, yet because underlying eviction metadata is shared.

We demonstrate PRIME+RETOUCH in two architectures: predominant Intel x86 and emerging Apple M1. We elucidate how PRIME+RETOUCH can break the T-table implementation of AES with robust cache side-channel mitigations such as Cloak, under both normal and SGX-protected environments. We also manifest feasibility of the PRIME+RETOUCH attack on the M1 platform imposing more restrictions where the precise measurement tools such as core clock cycle timer and performance counters are inaccessible to the attacker. Furthermore, we first demystify undisclosed cache architecture and its eviction policy of L1 data cache on Apple M1 architecture. We also devise a user-space noise-free cache monitoring tool by repurposing Intel TSX.

## I. INTRODUCTION

Cache side-channel attacks have recently gained increasing attention due to their broad impacts [8, 19, 25, 40, 41, 45, 54]. The majority of cache attacks rely on the observable timing differences between a cache hit and a miss caused by the access latency of memory hierarchies. By carefully manipulating a target cache set, the attacker can force timing differences that lead to leakage of the victim’s secret data.

As cache side-channel attacks have continuously broken carefully designed systems, various detection and mitigation techniques targeting them have been proposed [15]. Practical and widely experimented defense mechanisms are based on the assumption that cache side-channel attacks pose observable side effects themselves as well. That is, detectable attacker efforts, commonly forcing cache evictions, are required to capture meaningful victim memory access activities. Therefore, several mitigation proposals aim to detect such traces imprinted on the cache by directly monitoring abnormal microarchitectural behaviors [9, 11, 12, 36, 55]. Others seek to conceal the victim’s

access pattern by preloading all necessary data into the cache before issuing sensitive memory operations [4, 6, 13, 29, 48]. Recently [17] demonstrated that locking the preloaded data in the cache using transactional memory can further eliminate cache evictions of preloaded data.

In this paper, we introduce a new attack, PRIME+RETOUCH, that can leak information while maintaining the victim cache state and minimizing microarchitectural traces, completely bypassing the known mitigation techniques. The core idea of PRIME+RETOUCH is to accurately infer the cache accesses from the metadata of the cache replacement policy. More specifically, after reverse engineering the Tree-based Pseudo Least-Recently Used replacement policy (Tree-PLRU) of the Intel and Apple M1 processors, we discover that the attacker can learn the memory access history of a co-located victim from the states of the Tree-PLRU’s tree-shaped metadata without causing evictions of sensitive data and code.

PRIME+RETOUCH has three unique properties compared with the popular forms of cache-based attacks [43, 54]. 1) no eviction of sensitive data and code is required, 2) no shared memory with the victim is required, and 3) the memory access pattern is leaked through the metadata of the cache replacement policy. The PRIME+RETOUCH attack proves that it is possible to distinguish actual memory accesses without accessing or evicting preloaded data, which completely breaks the assumption of prefetch-based cache side-channel mitigations. The PRIME+RETOUCH attack is also challenging to detect, as the victim cannot identify the source of cache accesses while PRIME+RETOUCH only requires single synchronized cache access to one of the attacker-primed entries. Total mitigation might even require hardware modifications regarding the L1 cache replacement policy. To our best knowledge, the PRIME+RETOUCH attack is the first cache side channel attack that abuses L1 replacement policy to leak the victim’s access pattern on Intel x86 and Apple M1 architectures.

However, it is non-trivial to reliably demonstrate PRIME+RETOUCH in a modern CPU for three reasons: First, as PRIME+RETOUCH does not directly rely on timing information, it is hard for the attacker to synchronize with the victim’s operations and interfere at the correct moment. Second, analysis activities during attacks waste processor cycles and further exacerbate the above challenge. Lastly, as Tree-PLRU is sensitive to the order of cache accesses, unsynchronized cache accesses from the attacker render undesired states of the Tree-PLRU metadata that introduce no meaningful information except noise.

We successfully overcome those challenges by postponing memory access analysis after the active attack phase using the Tree-PLRU metadata and the novel technique of PLRU Aware RETOUCH §V-C, which is the first of its kind comparing to noisy realtime measurements adopted by traditional side-channel attacks. We demonstrate realistic PRIME+RETOUCH attacks in action (*e.g.*, attacking AES T-Table) and show that the attack successfully bypasses all prefetch-based countermeasures. We further extend the PRIME+RETOUCH attack to Apple’s M1 platform. We have reported the PRIME+RETOUCH attack to Intel and got acknowledged about the attack method.

**Summary.** This paper makes the following contributions:

- We introduce a novel method that allows noise-free L1 cache monitoring using Intel TSX in user space.
- We first study the cache architecture of Apple’s M1 processor using the undocumented performance counter and expose the underlying Tree-PLRU policy of its L1 cache.
- We propose PRIME+RETOUCH, a metadata-based L1 cache side-channel attack that only requires a single synchronized memory access without evicting the victim’s data, and show how PRIME+RETOUCH completely breaks the assumption of prefetch-based mitigations.
- We provide thorough analysis of the unique properties of the Tree-PLRU eviction policy and develop techniques to enable the attacker to synchronize with the victim and make the PRIME+RETOUCH attack practical.
- We demonstrate that the PRIME+RETOUCH attack successfully leaks victim secrets (*e.g.*, access patterns in AES T-Table) not only under settings assumed by traditional side-channel attacks, but also SGX-protected environments
- We demonstrate that the PRIME+RETOUCH attack can be extended to the M1 platform where no fine-grained timer is available by purely leveraging the Tree-PLRU policy.

## II. BACKGROUND

### A. Cache Architecture

CPU caches are small but fast memory chunks located between CPU cores and RAM. It serves to store data that the CPU is most likely to need next. CPU caches are categorized hierarchically according to their affinity to CPU cores. Traditionally, low-level caches (L1 and L2) are private to individual CPU cores, smaller in size and closer to the processor and thus faster, while the last-level cache (LLC) is bigger in size and shared among cores. When simultaneous multi-threading is enabled (*e.g.*, hyperthreading in Intel), two logical cores can share L1 and L2 caches.

CPU caches commonly adopt the W-way set associative design, where the cache memory is divided into sets and each set holds W lines of usually 64 bytes of data. To locate desired data in a cache, bits of a memory address are divided into different sections – offset to locate specific cache line, index to determine the cache set, and tag to flag whether data is cached – and utilized by addressing algorithms to derive the exact location.

### B. Cache Attacks and Mitigations

Cache attacks aim to leak information about whether specific cache lines have been accessed by a victim program. Among

various sources of leakage, many of them utilize the time difference between a cache hit and a cache miss. Since a cache miss takes significantly longer to retrieve the desired data, the attacker can infer memory access patterns of victim programs by carefully manipulating cache lines mapped to the memory.

**PRIME+PROBE.** PRIME+PROBE attack and its variants [3, 14, 43] monitor victim access to cache lines within a specific cache set throughout two phases. The attacker keeps accessing the target cache set so that it is completely filled with the attacker’s data (*prime* phase). Then, the attacker waits for a predetermined amount of time and again accesses the data he has loaded previously while measuring the load latency (*probe* phase). If the victim has accessed the target cache set, some of the attacker-primed data will be evicted, causing the reloading of the attacker’s data to take longer due to cache misses. As a result, the attacker can infer the victim’s memory access activities at a cache set granularity. However, PRIME+PROBE style attack incurs a significant amount of cache miss events, making the attack more easily detectable.

**Preloading and cache pinning.** For decades, researchers have been trying to mitigate cache side-channel attacks in various ways [32, 38, 39, 52, 56]. Recently, [4, 6, 13, 29, 48] have discussed preloading sensitive data into the cache to eliminate cache traces left by the victim’s data accesses. Furthermore, Gruss *et al.* [17] propose Cloak, which utilizes Intel TSX to preload and pin sensitive data and code in transactional memory during execution. If the attacker evicts corresponding cache lines, Intel TSX allows the victim process to be immediately interrupted and capture the malicious behavior. Note that Cloak can provide stronger cache defense compared to naive preloading which cannot prevent attacker from interfering between prefetch and genuine accesses. Cloak is also applied to Intel SGX to mitigate cache side-channel attacks targeting enclave programs. Notwithstanding the defenses posing a limit to the level of leakage through the cache status, PRIME+RETOUCH demonstrates that attacker can still leak information through the underlying eviction policy without causing evictions of the victim’s cache lines.

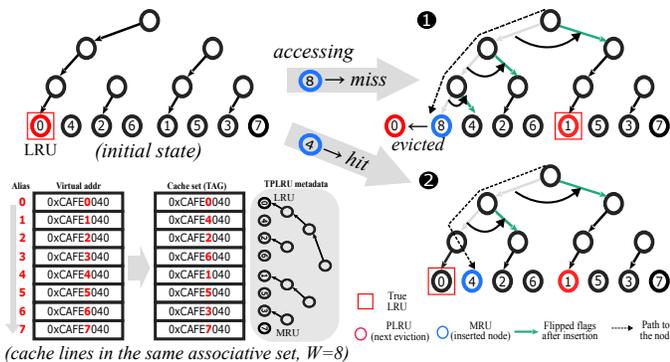
### C. Cache Replacement Policies

If entire cache lines of one set are filled, a next cache miss in that set will *evict* one of its present cache lines in order to host the newly fetched data. *Cache replacement policy* determines which cache line to evict. In concept, the replacement algorithm keeps track of certain metadata of the cache lines according to the history of cache accesses, on which the selection of the best cache way to evict is based.

**LRU.** The Least-Recently Used cache replacement policy is based on the assumption that it is more likely for the processor to use the more recently fetched or accessed data than the stale ones. To preserve the temporal locality, the LRU algorithm keeps track of the age of each cache line and chooses to evict the least recently used (oldest) cache way upon cache misses. Due to the expensive memory requirement and latency in storing age information and updating LRU records, an approximation of the LRU policy called Pseudo Least-Recently Used (PLRU) is often used.

**Tree-PLRU.** The Tree-based Pseudo Least-Recently Used policy (Tree-PLRU) [47] uses a binary tree structure as its

metadata to encode the Pseudo Least-Recently Used (PLRU) relationship within a cache set. The cache lines in the same set are divided recursively in binary to sub-groups until it reaches 2-way groups. As a result, the division produces a binary tree (see Figure 1). Each leaf node of the binary tree represents the physical location of a cache way (TAG). Each intermediate node has a flag that indicates the less recently accessed sub-group represented as a sub-tree in the binary tree. To find the PLRU cache way that best approximates the LRU way, Tree-PLRU starts from the root node, and traverses along the less recently accessed sub-trees. The finally reached leaf node then contains the best approximated PLRU cache way under Tree-PLRU. In addition, when a cache line is accessed, all the flags on the reaching path from the root node to the leaf node associated with the accessed line are updated to lead away from the path, making the cache way the most recently accessed (MRU).



**Fig. 1: Tree-PLRU in action.** It illustrates how the internal metadata of Tree-PLRU changes on a cache miss and hit on a cache line of the same associative set.

**Tree-PLRU in action.** Upon a cache hit or a cache miss, Tree-PLRU updates its metadata that represents the Pseudo-LRU ranks of cache lines per associate set. We define the *PLRU rank* of a cache way as the placement of the cache way regarding the eviction order determined by the specific PLRU policy. The sooner a cache way can get evicted under the specific PLRU policy, the higher placement in the PLRU eviction order the cache way has and thus, the higher PLRU rank. In the case of a cache miss (① in Figure 1), Tree-PLRU evicts the PLRU node (e.g., ① in ① reached by chasing the pointers from the root node) and flips all the flags along the path to lead away from the evicted node, indicating the node as the most recently accessed one (i.e., no pointers now directing to the node). In the case of a cache hit (② in Figure 1), it simply flips all the flags along the reaching path to lead away from the accessed cache line, which similarly indicates it as the now most recently accessed one. One significant side effect we leverage in our attack is how the LRU status is approximated in Tree-PLRU as PLRU ranks. By virtue of the approximation, the LRU entry will not match the entry marked as the highest PLRU rank in some circumstances. For example, in Figure 1-②, Tree-PLRU considers ① as the next eviction entry (PLRU) after ④ is accessed due to the tree-based metadata that approximates the PLRU ranks of cache ways, although ① is the true LRU entry.

#### D. Intel TSX

Intel Transactional Synchronization Extensions (TSX) implements a hardware-based transactional memory that allows multiple hardware threads to run critical sections concurrently by detecting conflicts on shared data. When a transaction is aborted due to a conflict, all monitored data will be discarded, thereby restoring to the initial state (i.e., rolled back).

### III. DEMYSTIFYING L1 EVICTION POLICY

In this section, we explain how we examine the details of the undocumented L1 cache eviction policy of modern Intel processors and Apple’s M1 processor. To understand the underlying cache architecture, we use Intel TSX as a noise-free monitor to capture specific L1 cache eviction events on Intel processors. For the same purpose on the M1 processor, we use the undisclosed Apple’s performance counter. Although we provide systematic approaches to collect cache traces, for recovering the policy from the trace, we refer to [2, 51].

**In-order memory accesses.** The internal states of an eviction policy can highly depend on the order of cache accesses. However, modern processors adopt out-of-order execution and memory speculation. To retain the expected order of cache accesses, we craft in-order memory operations and time measurement primitives by carefully inserting memory barrier instructions and using pointer chasing [53].

#### A. Intel Processor

Reverse engineering the cache eviction policy is not a new problem in the x86 architecture [2, 10]. However, achieving it without noise in user space is challenging. For example, Briongos *et al.* [10] encounters false positives due to the adoption of timing based measurements, while nanoBench [2] leverages hardware Performance Monitoring Extension (PMU) that requires root privilege to achieve accurate results, restricting its scope of application. We devise a user-space noise-free reverse engineering technique utilizing Intel TSX, and uncover that the underlying eviction policy is Tree-PLRU. We used the same environment setting described in §VI.

**Different behaviors of TSX read-set and write-set.** Intel TSX, within a transactional region, tracks memory addresses written into as a write-set and memory addresses read from as a read-set. To prevent conflicting accesses where another logical processor either reads memory from the write-set or writes to memory of either the read- or write-set, Intel TSX monitors such events at different cache levels. When a cache line mapped to an address in the write-set or the read-set is evicted from the L1 data cache or the L3 cache, respectively, it triggers TSX abort resulted from the conflicting access.

#### Algorithm for reversing the L1 cache with TSX.

The essence of TSX-assisted reversing is utilizing different condition of conflicting access on the read-set and the write-set. Regardless of memory write or read operations, the cache lines are fetched to the L1 data cache. Nevertheless, a TSX abort will be triggered only when a cache line in the write set is evicted from the L1 cache. Note that an abort will not be triggered when a cache line in the read-set is evicted from the L1 or L2 cache. Therefore, we can reliably measure how many fresh cache misses, thus evictions, are required to evict a specific

---

**Algorithm 1:** TSX-supported eviction policy reversing

---

**Input:**  $X[nWays]$ ,  $Y[nWays]$ : Addresses in set  $X$  and  $Y$  are mapped to the same set, but  $X$  and  $Y$  are disjoint.  
**Output:**  $evictSeq[nWays]$ : Num of required evictions per cache line

```
1 for  $target \leftarrow 0$  to  $nWays$  do
2   do
3     for  $numEvict \leftarrow 1$  to  $nWays+1$  do
4       beginTSX
5       for  $i \leftarrow 0$  to  $target$  do
6         memRead( $X[i]$ )
7       memWrite( $X[target]$ )
8       for  $j \leftarrow target + 1$  to  $nWays$  do
9         memRead( $X[j]$ )
10      memRead( $X[read\_primed]$ )
11      for  $k \leftarrow 0$  to  $numEvict$  do
12        memRead( $Y[k]$ )
13      endTSX
14    while TSX abort
15    if TSX aborts then
16      evictSeq[target] = numEvict
```

---

cache line tracked by the write-set from the L1 data cache. Note that this number of eviction represents the internal decisions of the cache replacement policy. In each round of Algorithm 1, we load the  $target - 1$  number of the cache lines (Line 5-6) to the read-set, the  $target$  cache line to the write-set (Line 7), and the rest of the cache line to the read-set (Line 8-9). At this stage, since the associative set contains all the data (from set  $X$ ) we loaded, an additional cache line accesses to the same associative set will incur an eviction (Line 11-12). Then, we count how many *new* cache lines (from set  $Y$ ) can be loaded until the  $target$  cache line is chosen for eviction by the underlying eviction policy. Note that any L1 data cache eviction from the read-set would not cause the TSX to abort. Furthermore, to understand how the underlying eviction policy is affected by the L1 cache hit events, we enumerate different combinations of read operations on the primed data (Line 10). Therefore, we expose the eviction orders of the cache lines within the same associative set resulting from all combinations of cache hits and learn that the changing eviction order conforms to the behavior of Tree-PLRU.

### B. Apple M1 Processor

It is challenging to explore the microarchitectural behaviors of the newly released Apple M1 processor, as its internal states and interfaces are not previously studied. Nonetheless, we successfully reveal that the M1 processor adopts the Tree-PLRU eviction policy for the L1 data cache. To reach the obtained results, we utilized the Apple’s hardware PMU to monitor its L1 data cache events because the M1 processor is not equipped with transactional memory. In detail, we utilize the fact that L1 cache hit and miss events have different latency. We expect that TSX-style reverse engineering (§III-A) is possible with Transactional Memory Extension [21] deployed in ARMv9 [24]. We used the same environment setting described in §VII.

**Reversing with the undocumented PMU.** The first challenge we faced in reversing the M1 architecture is the lack of a publicly available interface as a measurement tool, such as hardware PMUs including a high-resolution timer. Although the M1 architecture expands the ARMv8, we found that the M1 does not deploy the standard ARMv8 PMU [23] (*i.e.*,

reading and writing system registers of ARMv8 PMU halts the system). Therefore, we reverse-engineered the interface of the Apple’s proprietary PMU and related events. Although no official document from Apple discloses such information, we found that Apple’s XNU kernel partially utilizes their proprietary PMU. Based on the analysis and experiment, we found that the M1 processor can concurrently monitor at most 10 PMU events per core, and each PMU can track one of the 255 undocumented events (0 to 254). We also empirically found two undocumented PMU events:  $0x2$  that tracks the core clock cycles (*e.g.*,  $rdtsc$  as in x86) and  $0xa3$  that monitors the L1 data cache miss event. Those two PMU events are enough to uncover the replacement policy of the L1 data cache.

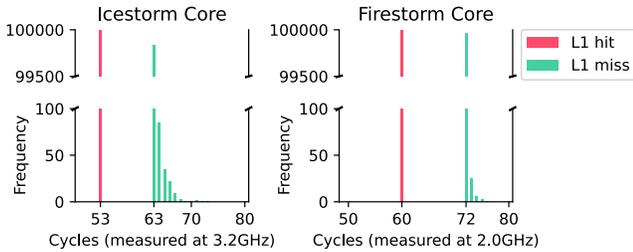
**Cache hierarchies of M1.** Information about cache hierarchies such as set associativity, number of sets, and cache line size is essential for reversing the replacement policy. Unfortunately, to the best of our knowledge, there is no public documentation specifying the M1 processor’s cache hierarchies. Therefore, we for the first time retrieve such information by utilizing the system registers  $CCSIDR_{EL1}$  and  $CSSELR_{EL1}$ . As the M1 architecture adopts two different types of cores (Firestorm and Icestorm), we collect two sets of cache hierarchies respectively. All the detailed information is described in Table I.

CPU Type	FireStorm		IceStorm	
	L1 DCache	L2 DCache	L1 DCache	L2 DCache
Number of Set	256	8192	128	2048
Set Associativity	8	12	8	16
Cache Line Size(Byte)	64	128	64	128

**TABLE I:** Information of data cache hierarchies in M1 processors

**L1 data cache set mapping.** To measure the L1 data cache activities in a particular set, we should be able to manipulate the addresses mapped to a particular set. For example, we need at least 9 addresses mapped to a particular set as the L1 data cache is 8-way set associative in both types of the cores. We found that M1 employs different cache set mappings depending on the core type. For all virtual address  $x$ , its L1 data cache set  $s$  is determined by the following equations:  $s = (x \bmod 16384)/64$  on Firestorm core and  $s = (x \bmod 8192)/64$  on Icestorm.

**L1 cache hit and miss latency.** To measure the L1 hit latency, we first prefetch one memory address with  $ldr$  instruction and then measure the latency of loading the same entry once again. If the entry has been prefetched before being accessed in the measurement, it will always result in an L1 hit. Although we have no information about the eviction policy yet, we can still measure the L1 miss latency because we are aware that each L1 data set consists of 8 cache lines (Table I). In detail, we access 16 memory addresses mapped to a particular L1 data set and measure the latency to access the first entry once more. Note that the first 8 entries are evicted from L1 caches after the following 8 accesses occur. Therefore, the accesses to the first 8 entries will not be served by the L1 data cache. As shown in Figure 2, we can clearly distinguish L1 hits from L1 misses in both types of the cores. We implement the reversing logic as a kernel driver and launch measurements on a designated processor using  $smp\_call\_function\_single$  to eliminate the noise introduced by context switching.



**Fig. 2:** L1 hit and miss latency for Icestorm and Firestorm core. The measured latency includes the memory barrier instruction’s latency: 51 cycles and 56 cycles, respectively, on Icestorm and Firestorm core. Therefore, on two different cores, the actual L1 hit latency is 2 cycles and 4 cycles, and L1 miss latency is around 12 cycles and 16 cycles, respectively. Note that each core runs at different clock frequency.

**Reversing M1 eviction policy.** To reverse engineer the eviction policy of the L1 cache on the M1 platform, we follow the similar reversing logic described in Algorithm 1. Based on the latency difference measured in Figure 2, we can determine which cache line has been evicted and recover the eviction policy from the trace. Also, we utilize the `0xa3` PMU events together to monitor L1 cache activities for better accuracy. Due to the lack of space, implementation details are described in §A.

#### IV. PRIME+RETOUCH

PRIME+RETOUCH has three distinctive characteristics compared with known cache side-channel attacks (see §II-B).

① **Stealthy.** PRIME+RETOUCH does not incur eviction of the victim’s prefetched data to leak the access patterns. Since PRIME+RETOUCH only accesses the cache lines primed by the attacker to manipulate eviction metadata (*i.e.*, internal states of Tree-PLRU), it does not interfere with the victim’s execution or destruct cache state. This property is particularly important because existing defenses against PRIME+PROBE and FLUSH+RELOAD rely on preloading and locking the cache sets [17] to prevent cache evictions.

② **Minimal synchronization.** When an attack requires multiple cache accesses, it is more challenging to synchronize those accesses with the victim’s access. Thus, incorrect synchronization incurs noise and makes the attacks unreliable. Furthermore, when a prefetch-based defense is deployed, especially for PRIME+PROBE style attacks, the targeted windows tend to be narrower because the attacker should evict the prefetched entries before they are consumed. As a result, the attacker has to average out the noise by repeatedly launching the procedure a significant number of times [10, 14, 17], thus leaving a detectable level of microarchitectural traces for detection-based countermeasures [9, 55]. In contrast, PRIME+RETOUCH requires only one synchronized memory access to the attacker’s pre-primed cache lines, minimizing both noise from synchronization efforts and microarchitectural traces left.

③ **Leakage via non-shared memory.** Attacks such as FLUSH+RELOAD and RELOAD+REFRESH [10] require shared memory between the attacker and the victim to introduce changes on cache status, such as flushing the victim’s cache line. However, the requirement highly limits the range of application scenarios. In contrast, PRIME+RETOUCH does not require any shared memory resource to leak the victim’s access pattern.

Since the PRIME+RETOUCH attacker has complete knowledge about the shared eviction policy, the attacker can precisely reveal access patterns of a target cache line through the eviction metadata instead of the cache itself.

#### A. Attack Model

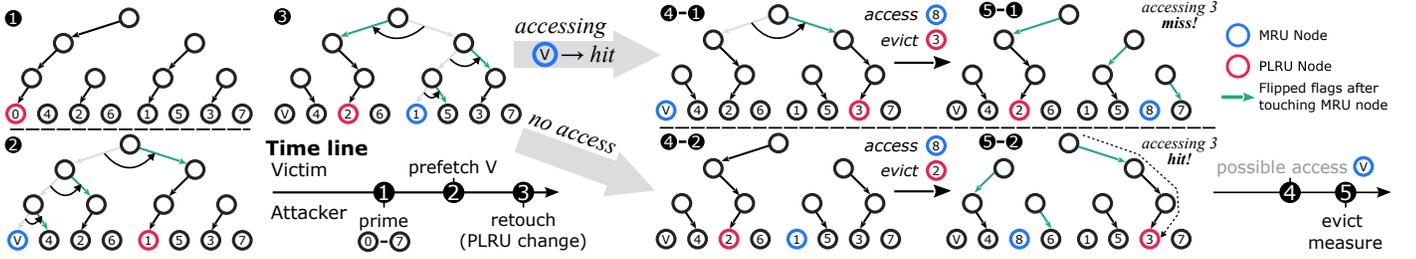
PRIME+RETOUCH is a general attack applicable to any architecture meeting the below assumptions. Note that further details will be described in §VI, §VII based on the target architecture. Regarding the execution environment, we assume that the L1 data cache is shared between attacker and victim. We assume that the victim process prefetches data before accessing to conceal the access pattern. We also assume that cache evictions of sensitive data are not allowed. Considering the attacker’s capability, we assume that the attacker can replay the security-sensitive code unlimited times without introducing unexpected interruptions to the victim process. The attacker is not expected to have any shared memory with the victim. Lastly, we assume that the attacker can freely allocate virtual memory to pick addresses that are mapped to desired L1 cache sets.

#### B. Leaky Tree-PLRU

The Tree-PLRU cache replacement policy aims to mimic a true LRU policy by using a tree-based data structure, as explained in §II-C. One of the most distinct properties of Tree-PLRU is that the eviction metadata used to decide the eviction order is encoded as a binary tree structure when entries are placed into the cache. Therefore, the exact eviction order for LRU is not accurately tracked and approximated by sub-trees (Pseudo-LRU). In detail, one cache way is grouped with other cache ways by sub-trees represented in the eviction metadata, and the PLRU rank of a cache line (*i.e.*, eviction order) can be affected by where an access to or an eviction of cache lines in the same sets occurs in the tree structure. We found that such a property can be carefully manipulated by an attacker to produce eviction metadata that leak the cache activities of the victim without causing evictions. We now explain how this can be achieved.

**Indistinguishable back-to-back accesses.** Consecutive accesses to one cache line back-to-back are observed as single memory access occurring under Tree-PLRU since the Tree-PLRU approximates the order of accesses as a binary tree. As we recall from §II-C, when a cache line is accessed, all the flags on the reaching path from the root node to the associated leaf node are updated to lead away from the path, making the cache line the most recently accessed (MRU). However, if the cache line is accessed again back-to-back, the corresponding flags will then remain unchanged, as they are already updated and indicating the correct less recently accessed sub-trees. As a result, the fact that one cache line has been accessed multiple times cannot be revealed directly from the final tree state if the accesses occur back-to-back. In the case of PRIME+RETOUCH, since a prefetch operation essentially accesses the prefetched data, we cannot discover a back-to-back victim access to the prefetched data directly from the final tree state either.

**Distinguish victim’s access from prefetch operation by retouching.** To distinguish back-to-back accesses to the same cache line solely from final tree states, the attacker should be



**Fig. 3:** An overview of PRIME+RETOUCH against the L1 cache with the Tree-PLRU policy. Changes on Tree-PLRU are demonstrated following the timeline. Depending on the presence of the victim’s access following the prefetching, two different PLRU cache ways will be produced.

able to map one access to exactly one time of observable tree state change. Therefore, by checking whether an additional change in the tree state has occurred, the attacker can discover the victim’s access following the prefetch. We define an observable tree state change as a change of the node indicating the PLRU cache way (*i.e.*, next eviction entry). Note that a change of the PLRU cache way is observed only when the flag of the root node is flipped. In Figure 3, attacker primes all cache lines of one set (1), and waits until the victim prefetches (V) (2). To discover the victim’s access following the prefetch, attacker retouches the PLRU cache way (1). Consequently, it flips the corresponding flags, including the root node flag, and produce a new PLRU node (2), causing the root node flag to direct toward the prefetched entry (3). If a subsequent victim access to the prefetched entry occurs (4-1), the root node flag will be flipped again to lead away from the prefetched entry, announcing a new PLRU cache way (3). In contrast, 2 remains as a PLRU cache way when there is no victim access after preloading (4-2). If the attacker had not retouched the PLRU cache way, both 4-1 and 4-2 would have produced the identical PLRU tree, which prevents discovering the victim’s access following the prefetch.

**Probe the latest PLRU cache way and match with cache activities.** When another miss event is introduced to the set, PLRU cache way will be evicted and allow a new entry to be cached. However, note that different cache ways will be evicted depending on which entry in the set is the PLRU entry, which is determined by the possible victim access following the retouch. We produce a cache miss by accessing (8), a new cache line associated with the current set, to evict the current PLRU cache line in both cases. Note that the evicted cache line belongs to attacker’s primed set and does not affect the victim’s cache way. Consequently, different entries will reside in the set (2 remain in 5-1, and 3 remain in 5-2). We can discover whether there were victim’s access following the prefetch by measuring the access latency of (3). A cache miss latency indicates that (3) was indeed evicted and confirms that the victim has accessed the prefetched entry while a L1 cache hit latency indicates that (2) had been evicted instead, indicating that there was no further access from victim.

### C. Synchronization in PRIME+RETOUCH

In the previous section §IV-B, we describe the attack only when the attacker’s retouch happens after the victim’s prefetch (A and B in Table II). However, as shown in Table II, five different operation sequences can happen depending on the order of the victim’s and attacker’s memory accesses during

	Ⓐ $P \rightarrow R \rightarrow A$	Ⓑ $P \rightarrow R$	Ⓒ $P \rightarrow A \rightarrow R$	Ⓓ $R \rightarrow P \rightarrow A$	Ⓔ $R \rightarrow P$
Retouch ①	PLRU:3	PLRU:2	PLRU:2	PLRU:3	PLRU:3
PLRU Aware	PLRU:3	PLRU:2	PLRU:2	PLRU:2	PLRU:2

**TABLE II:** PLRU cache ways produced by all five possible operation sequences. After applying PLRU Aware RETOUCH (§V-C, the sequence Ⓐ can be distinguished among all five sequences.

PRIME+RETOUCH attack (*i.e.*, synchronization). In this section, we will demonstrate the challenges imposed by synchronization in PRIME+RETOUCH.

**Retouching ① can produce false positive results of distinguished victim accesses.** If the attacker cannot adequately achieve precise synchronizations, unwanted operation sequences are produced and result in noisy Tree-PLRU metadata, posing *false positives*. Particularly, when an attacker’s retouch occurs prior to the victim’s prefetch, two extra operation sequences (D and E) (we call them unsynchronized sequences) can occur.

$$\text{D) } Retouch_{Attacker} \rightarrow Prefetch_{Victim} \rightarrow Access_{Victim}$$

$$\text{E) } Retouch_{Attacker} \rightarrow Prefetch_{Victim}$$

As shown in Figure 4, although synchronized sequence (A) and unsynchronized sequence (E) represent opposite victim cache activities (access vs. no access), they result in the same Tree-PLRU eviction metadata. Note that unsynchronized sequence (D) results in the same Tree-PLRU metadata as (E) because the following victim’s access occurs immediately after the prefetch. In fact, the mixture of all five possible sequences makes the previously unique PLRU cache way produced by the desired sequence (A) completely indistinguishable, as shown in Table II. Therefore, without techniques to differentiate such conflicting cases, final tree states resulting from unsynchronized retouch cannot render useful information about victim cache activities.

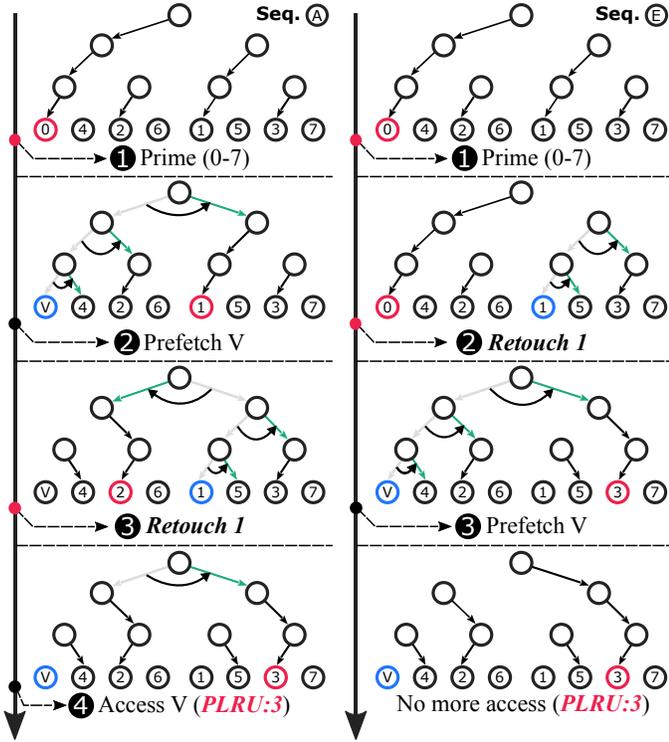
**Synchronization can eliminate false positives.** The root cause of false positives is unsynchronized retouching before the victim’s prefetching. Therefore, if the attacker is able to successfully retouch (1) after the victim entry has been prefetched, only three possible operation sequences can occur and eliminate the false positives:

$$\text{A) } Prefetch_{Victim} \rightarrow Retouch_{Attacker} \rightarrow Access_{Victim}$$

$$\text{B) } Prefetch_{Victim} \rightarrow Retouch_{Attacker}$$

$$\text{C) } Prefetch_{Victim} \rightarrow Access_{Victim} \rightarrow Retouch_{Attacker}$$

Each sequence results in different tree states following Tree-PLRU policy, where the PLRU cache way resulting from sequence (A) is different from the other two (B, C). Since we know the sequence (A) produces a unique PLRU cache way among the three, we can probe whether sequence (A) has occurred by forcing a cache miss and check whether indeed the unique PLRU cache line got evicted.



**Fig. 4:** When RETOUCH ① is applied, the identical Tree-PLRU metadata is produced as a result of two different operation sequences: synchronized (left) and unsynchronized (right).

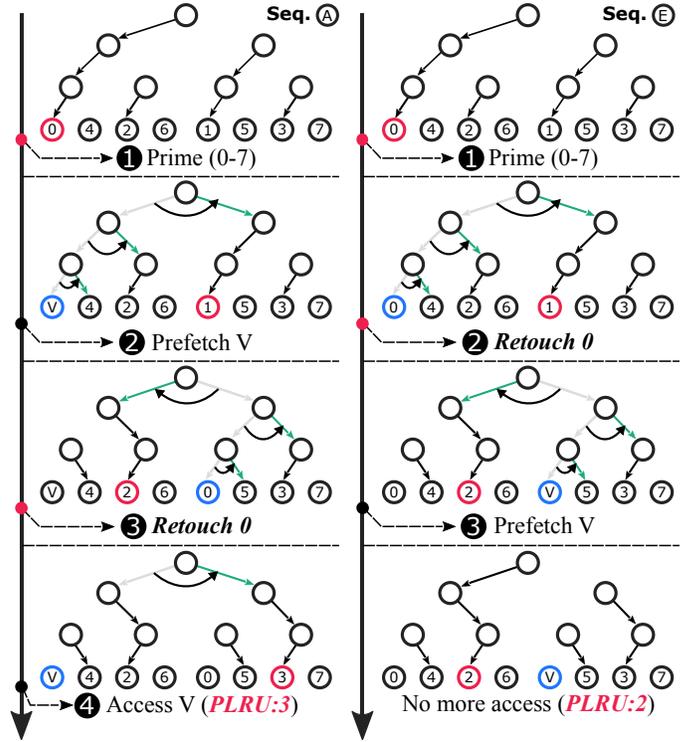
## V. PSEUDO SYNCHRONIZED PRIME+RETOUCH

In this section, we introduce the pseudo synchronized PRIME+RETOUCH attack, which does not require precise synchronization but still achieves the same leakage by purely relying on Tree-PLRU metadata. We first look at characteristics of Tree-PLRU metadata that create *false positives* in distinguishing the victim access from prefetch when assistance for precise synchronization is not available. We then propose a new technique called *PLRU Aware* RETOUCH, which cleverly solves the challenge introduced by unsynchronized RETOUCH and shows that when applied, the attacker can achieve the same leakage as with precise synchronization.

### A. Limited Precise Synchronization Techniques

Various synchronization techniques to timely introduce an attacker’s interference have been discussed under various threat models [14, 49]. Nonetheless, it is not always the case that such assistance is available to achieve precise synchronization.

**Hardware transactions.** PRIME+ABORT [14] allows the attacker to accurately observe a victim’s cache access using TSX abort. Intuitively, a PRIME+RETOUCH attacker might also rely on such a technique to precisely synchronize retouch with the victim’s prefetch. Unfortunately, TSX-assisted synchronization could not work in our threat model. From the latest stepping R0 [26, 27], Intel has deployed hardware changes as mitigations against microarchitectural side channels [26, 34, 37, 46, 50]. This mitigation prevents a hyperthreaded core from handling TSX transactions while its sibling core runs an SGX enclave process. Consequently, even if the co-located attacker initiates a TSX transaction before the victim executes, the transaction



**Fig. 5:** When PLRU Aware RETOUCH (§V-C) is applied to the case in Figure 4, by probing the PLRU cache way represented in the final tree states, the attacker is able to distinguish the two operation sequences.

will immediately abort upon the victim’s entering the enclave without capturing any cache activity. Furthermore, TSX is disabled by default in modern operating system; and user-level attacker in PRIME+RETOUCH is prohibited from utilizing it.

### B. Unsynchronized RETOUCH

**The attacker’s dilemma.** Here comes the attacker’s dilemma: The PLRU cache way produced after the victim’s prefetch and attacker’s retouch operation, should belong to the same sub-tree of the victim’s cache line (root-to-left or root-to-right sub-tree); however, different orders of the two operations make the PLRU cache way to be located in different sub-tree. The fact that the PLRU cache way resides in the same sub-tree as the victim’s cache line guarantees that the victim’s access following the prefetch operation to be traced in the Tree-PLRU structure. Therefore, at the time of the two operations finished, if the victim’s entry and PLRU cache way are not located in the same sub-tree, PRIME+RETOUCH can misinterpret the Tree-PLRU structure.

As depicted on the right side of Figure 4, the attacker unexpectedly retouches ① when it is not synchronized with the victim’s prefetch operation (sequence E). The accessed cache line does not reside in the sub-tree that contains the PLRU cache way (*i.e.*, root-to-left sub-tree), so the PLRU cache way under Tree-PLRU does not change. However, the PLRU ranks of other cache lines within the opposite sub-tree (*i.e.*, root-to-right sub-tree) has been changed (*e.g.*, promoting sub-tree containing ③ and demoting sub-tree containing ①). Consequently, the effect on the eviction candidate is preserved and postponed until the next root node flip, *i.e.*, the next access to the less recently

accessed sub-tree. As a result of the following victim’s prefetch operation (③), PLRU rank changes captured in the Tree-PLRU structure, which was incurred by ②, makes the ③ as the PLRU cache way. On the other hand, the synchronized (left) attacker retouches ① when it is in the PLRU cache line after prefetch (②), causing ③ to have the highest plru rank in the sub-tree (③). As a result, both the following victim access when synchronized and the victim prefetch when unsynchronized (④) flip the root flag and produce ③ as the next PLRU cache way. To solve the dilemma under unsynchronized settings, retouch needs to do more than just mimic an access.

### C. PLRU Aware RETOUCH

Recall that the root cause of the attacker’s dilemma is the nondeterministic location of the PLRU cache way represented in the Tree-PLRU structure when not synchronized. The attacker wishes to always retouch the PLRU cache line to cause an immediate observable tree state change, but might unexpectedly retouch a cache way with low PLRU rank (high MRU rank) prior to victim prefetch and cause an indistinguishable final tree state. Our solution is called PLRU Aware RETOUCH, a special type of RETOUCH that always accesses the PLRU cache line. In order to always retouch the PLRU cache way, the attacker should either evict the cache line in the PLRU cache way, or directly access the PLRU cache way. PLRU Aware RETOUCH solves the dilemma by always retouching a cache line that satisfied both conditions, the resulting PLRU cache line immediately after the attacker prime. Note that the attacker knows the resulted PLRU cache line, as he is aware of the eviction policy. The reason behind is that retouching such a cache line will either bring back the cache line to the PLRU cache way after being evicted by victim prefetch by a cache miss caused by the following attacker retouch (left side in Figure 5), or access the PLRU cache way itself before victim prefetch (right side in Figure 5). Equivalently, the attacker then is always retouching the PLRU cache way, no matter which side the sub-tree resides on in the Tree-PLRU structure.

An example where the attacker applies PLRU Aware RETOUCH to the case in Figure 4 is shown in Figure 5. As we can see, the attacker chooses to retouch ① instead of ① as in Figure 4. When the synchronized sequence ④ takes place on the left side, ① is first evicted by the victim prefetch (②) as the resulting PLRU cache way immediately after attacker prime (①), and then brought back to the PLRU cache way by the attacker retouch as a cache miss (③). On the other hand, when unsynchronized sequence ⑤ takes place on the right side, ① is accessed by the attacker retouch (②) as the resulting PLRU cache way immediately after attacker prime (①). In both cases, the attacker successfully retouches the PLRU cache way. As a result, sequences ④ and ⑤ are nicely distinguished by examining the PLRU cache way in the final tree states.

**Reduction to synchronized PRIME+RETOUCH.** The insight brought by PLRU Aware RETOUCH is that prefetch is essentially a retouch with an eviction of the PLRU cache way in the perspective of the Tree-PLRU structure. Essentially, PLRU Aware RETOUCH, as demonstrated in Figure 5, reduces unsynchronized operation sequence ⑤ to synchronized sequence ④. Similarly, unsynchronized operation sequence ⑥ is reduced to unsynchronized operation sequence ④. Consequently, the final tree states can be analyzed in the same

way as the synchronized case §IV-B. Furthermore, the reduction from unsynchronized PRIME+RETOUCH to synchronized PRIME+RETOUCH offloads the synchronization analysis to a post-transaction phase and therefore eliminates any runtime checking of victim prefetch that can introduce extra noise to the attacker.

### D. A poor attacker’s approach with delay reduction

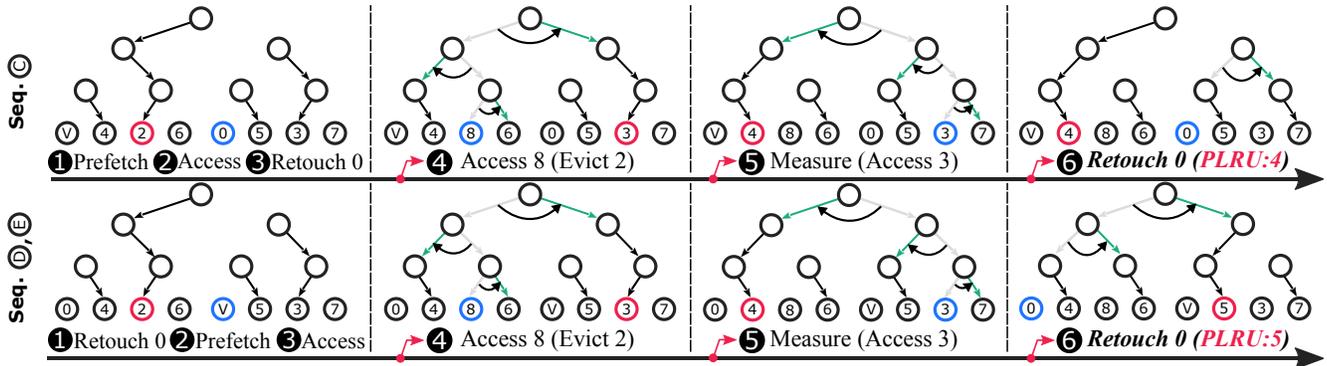
**Challenge: if retouching occurs too late, attacker can still miss the victim’s access to the prefetched data.** This is because the PLRU cache way resulting from sequences ④ and ⑤ are the same as shown in Table II, so that the attacker can miss a genuine victim access if the sequence ⑤ has occurred, producing *false negatives*. Note that the result is not dependent on retouching method. Furthermore, this challenge can be imposed even when retouching is synchronized after the victim’s prefetch. Even the synchronization support such as PRIME+ABORT only guarantee that the retouching happens after the victim’s prefetching not before the victim access if presents. Further analysis is thus required to capture missed accesses to avoid false negatives.

**A Poor man’s algorithm.** The intuition of the poor man’s algorithm is to locate the attacker’s retouching as closely as possible to the victim’s prefetching so that the victim’s access is unlikely to occur in between. The attacker first sets a long delay after the initial prime so that the attacker’s prefetching is guaranteed to occur after the victim’s access if there has been one. Then, the attacker keeps relaunching the PRIME+RETOUCH attack while reducing the delay in a fine-grained manner until the unique PLRU entry of sequence ④ is captured, which confirms a victim access, or the delay reaches zero, indicating sequence ⑤ has occurred instead. To successfully achieve this, two prerequisites are required. First, the poor man’s retouching should always be synchronized after the victim’s prefetch. Second, the delay reduction should be fine-grained enough to make retouching intervene the prefetch and possible genuine access so that it can reveal the victim’s access hidden by the false negative.

**Synchronization recovery for poor man’s algorithm.** Although PLRU Aware RETOUCH eliminates false positives, it does not guarantee that the attacker’s retouch always occurs after the victim’s prefetch, a prerequisite for poor man’s algorithm in §V-D. However, as the attacker does not have any timing information or convenient tools such as Intel TSX, extra analysis is needed to tell the order of occurrence. After the desired order of occurrence can be always achieved, we can apply the poor man’s algorithm to further eliminate false negatives. In short, we need to differentiate sequence ④ from ⑤ (we omit sequence ⑥ as its behavior replicates ⑤ due to back-to-back access to ⑥):

$$\begin{aligned} \textcircled{C} & Prefetch_{Victim} \rightarrow Access_{Victim} \rightarrow Retouch_{Attacker} \\ \textcircled{D} & Retouch_{Attacker} \rightarrow Prefetch_{Victim} \rightarrow Access_{Victim} \\ \textcircled{E} & Retouch_{Attacker} \rightarrow Prefetch_{Victim} \end{aligned}$$

More specifically, if we can distinguish the two sequences we can recover the synchronization information about the victim’s prefetch and the attacker’s retouch. Although the final state of the tree remains after applying PLRU Aware RETOUCH, it is worth noting that depending on the time when retouching occurs, the victim cache line ends up in the opposite group



**Fig. 6:** Although the same PLRU cache ways are produced after applying PLRU Aware RETOUCH, victim entries (V) in sequence (C) and sequence (D) end up in opposite sub-trees in the Tree-PLRU metadata. By retouching an attacker’s entry in the opposite group, e.g., (0), after probing the PLRU cache way (2), different new PLRU cache ways are produced. Consequently, the two sequences are differentiated by probing the new PLRU cache ways again (4 vs. 5).

of the Tree-PLRU structure, Figure 6, after probing the PLRU cache way (5). Namely, the root flag directs towards the victim entry under sequence (C) and leads away from the victim entry under sequence (D). The same situation applies to the Eviction Aware RETOUCH-ed cache line symmetrically. This fun fact can later be utilized by the attacker to recover the occurrence order of the victim’s prefetch and the attacker’s retouch, avoiding the noise brought by probing the eviction metadata during runtime for such information.

As shown in Figure 6, after probing the PLRU cache way (4 and 5) following the outcomes in Figure 5, victim entry (V) resides in opposite Tree-PLRU groups (5) with the same new PLRU cache way (4). By retouching the previously Eviction Aware RETOUCH-ed cache line, (0) in this case, sequence (C) will not cause the next PLRU cache way (PLRU:4) to change, as the root flag leads away from (0), while sequence (D) alters the next PLRU cache way (PLRU:5) as the root flag directs towards (0). We then can differentiate the two sequences by probing the eviction candidate again, and recover the occurrence order.

## VI. PRIME+RETOUCH ON X86

**Responsibility disclosure.** We have reported PRIME+RETOUCH to Intel and received an acknowledgment.

**Execution environment.** Our evaluation platform is equipped with an Intel Core i9-9900K (Coffee Lake) CPU @ 3.60 GHz, running Linux Ubuntu 20.04.1 LTS with a 5.4.0-rc8 kernel. The size of the L1 data cache is 32KiB per core, and it consists of 64 cache sets with eight cache lines per set (8-Way). The size of cache line is 64Bytes. The machine uses microcode version 0xd6 and adopts *R0 stepping* (i.e., 13). Therefore, we assume that defense described in §V-A is available, and the attacker cannot utilize TSX to assist synchronization. We assume that the hyperthreading is enabled and the attacker can co-locate a spy process concurrently with the victim process so that the L1 cache is shared. Last, we assume that attacker can utilize *rdtsc* to capture the L1 miss event.

**Prefetching and locking defense mechanisms deployed.** For data preloading to L1 cache, we utilize *write* operations inside a TSX transaction. Note that *write* operation is required because of implementation of [17], not because of intentionally

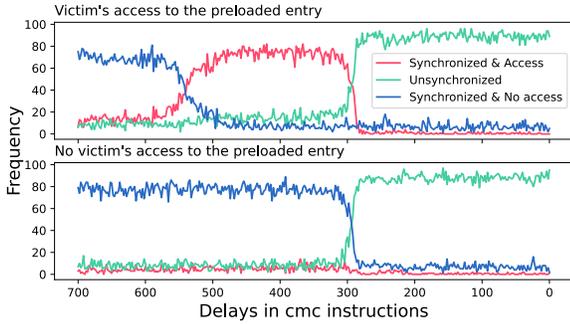
launching the PRIME+RETOUCH on a weak assumption. The victim accesses preloaded entries as he wishes using *read* operations. We assume that security-critical code and data can fit and are properly placed inside the TSX protection regions.

**PRIME+RETOUCH in SGX** For SGX environment, we exclude specific cores from scheduling by applying the *isolcpus* with *noHZ* boot parameter of the Linux kernel and use *rdpmc* instruction to capture L1 cache miss events to reduce noise. If there is no mention about SGX explicitly, all the experiment results are retrieved in a non-SGX environment.

### A. Baseline Evaluation of PRIME+RETOUCH

To effectively demonstrate the basic capability of PRIME+RETOUCH, we implemented a simple version of [17]. The victim process prefetches 64 data blocks, each mapped to a 64 different cache set inside a TSX transaction (L1 data cache consists of 64 sets in x86). We locate the victim’s access operation after the prefetch based on the experiment settings.

**PLRU Aware RETOUCH can effectively distinguish all operation sequences** We demonstrate that PRIME+RETOUCH can accurately distinguish victim’s accesses by leveraging PLRU Aware RETOUCH in Figure 7. In addition, we reveal the synchronization information retrieved from the Tree PLRU that allows the attacker to estimate when a victim’s further access occurs if it exists. In this experiment, we launched PRIME+RETOUCH 100 times per delay from 700 to 0 *cmc* range with 2 *cmc instructions* as cycle reduction granularity. Also, we iterated the identical experiment for two cases: when the victim’s access follows the preloading (above graph) and when no access follows the preloading (below graph). For each experiment, PRIME+RETOUCH successfully distinguished all three cases: (1) retouching in between prefetch and access (*Synchronized, access*), (2) retouching before prefetch (*Unsynchronized*), and (3) retouching after prefetch (*Synchronized, no access*). As shown in the upper graph, there is a distinctive increases on the *Synchronized, access* case around 550 to 330 *cmc* delays. This indicates that victim’s genuine access occurs in that interval most of the time. However, the lower graph couldn’t capture any drastic frequency changes in the same case (Red), which means no victim access was initiated after



**Fig. 7:** The only difference in these two graphs is the presence of victim’s access after the prefetch.

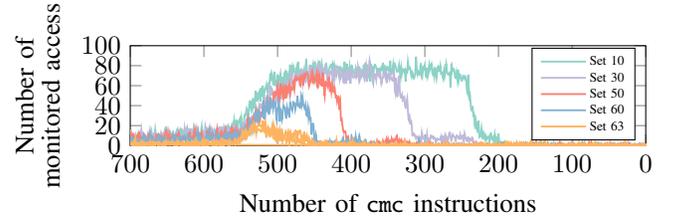
the preloading. Also, an intersection of two different lines at around 300 cmc in both graph demonstrates that the time of victim’s prefetch operation can also be pinpointed with PRIME+RETOUCH.

**Poor man’s algorithm reliably make RETOUCH to be located between victim’s prefetch and access** When the victim’s access occurs after preloading, timing delay between the preloading and actual access can determine the accuracy of PRIME+RETOUCH. It is easier to capture the access when the victim touches the entry prefetched earlier because there is sufficient timing delay that attacker can intervene for RETOUCH. For address mapped to cache set  $N$  ( $0 \leq N \leq 63$ ), it has around  $(63 - N) * \text{prefetch\_latency}$  delay between the preloading and the further access. To further evaluate the reliability of PRIME+RETOUCH, we launched PRIME+RETOUCH against five different cache sets that have various timing intervals between the prefetch and the following access. Our result shown in Figure 8 proves that the poor man’s algorithm described in §V-D reliably eliminates false negatives and accurately identifies the victim access. Although the signal telling presence of genuine accesses is getting weaker, as the time interval between the prefetch and the access decreases, it is enough to distinguish the victim accesses from the noise.

### B. Attacking AES T-Table

Although appropriate countermeasures [20, 31, 35, 44] have closed most of the cache side-channels presented in AES implementations, still the T-Table implementation of AES is frequently used to explore different characteristics of new side-channel attacks compared to the existing approaches [10, 17–19, 28]. Therefore, we demonstrate the effectiveness and stealthiness of PRIME+RETOUCH compared to PRIME+PROBE by attacking T-Table based AES encryption, both in regular environments and in Intel SGX enclaves. The T-Table implementation transforms AES operations (*i.e.*, SubBytes, ShiftRows, and MixColumns) in one round of encryption into 16 memory lookups to 4 different T-Tables. The T-Table accesses in the first round of encryption are stated by the equation  $T_j[p_i \oplus k_i]$  with  $i \equiv j \pmod{4}$  and  $0 \leq i < 16$ . Therefore, if the attacker can infer the values of  $p_i \oplus k_i$ , the indexes to T-Tables, they can then narrow down the possible key-bytes ( $k_i$ ) in case the plaintext ( $p_i$ ) is known [7, 42, 43].

**AES settings.** Our experiment assumes the known-plaintext attack scenario and targets the AES implementation in OpenSSL. Each T-Table consists of 256 entries of 4bytes (1KB of each)

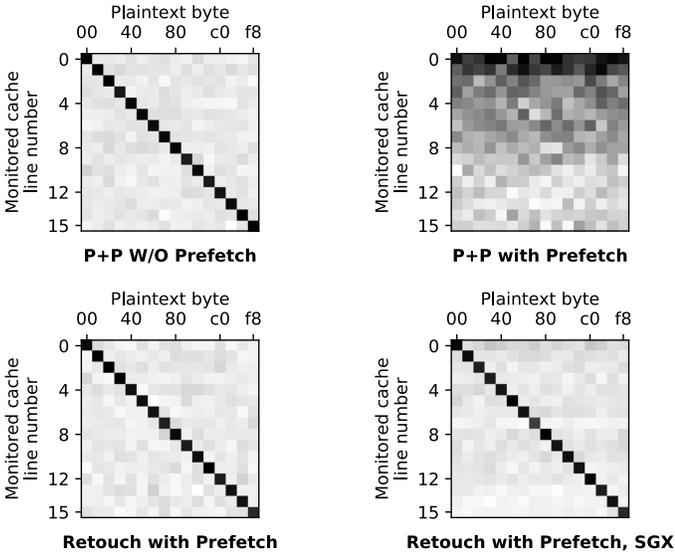


**Fig. 8:** Reliable false negative reduction achieved by poor man’s algorithm. 1 cmc instruction is set as the granularity of delay reduction. The initial delay has been set as 700 cmc instructions, and we iterated 100 times per delay until it reaches 0 cmc delay.

and maps to 16 different sequential sets of L1 cache. The first T-Table, Te0 is always mapped to the page-aligned addresses in our implementation, which map to set 0 to 15 of the L1 cache. Other T-Tables are mapped to the subsequent L1 data cache sets (16 to 63). We arranged the plaintext bytes to make the AES encryption access the cache line equal to  $p_0 / 0x10$ . The plaintext value shown in the Figure 9 indicates the first byte of plaintext ( $p_0$ ), and the random values are assigned to the left 15 bytes of plaintext ( $p_{1-15}$ ). Also, we generate a 128-bit AES key with zero-filled user key data. Note that this experiment setting is general and adopted in previous researches [17–19].

**Evaluation methods.** We launched around 300,000 encryptions asynchronously and measured the victim’s access made on the first T-Table mapped to the L1 cache set 0 to 15. For the prefetch-based defense, we first preloaded the entire four T-Tables to the Intel TSX write set and performed ten rounds of AES T-Table encryptions in a single TSX transaction. To bring the T-Table entries into a write set, we changed the T-Tables to be writable and introduced a minimum of 64 write operations, each of which strides cache line size. We measured the average time to preload 64 memory entries with write operations and organized the same amount of delay composed of cmc instructions. As a result, we could make the RETOUCH operation intervene between the preloading and actual AES encryptions. Also, we put additional random delays of less than 300 cmc to make the measurement operations initiate after the first round of AES encryption. The identical PRIME+RETOUCH attack was also conducted in Intel SGX enclaves with the prefetch-based defense applied.

**Evaluation analysis.** As shown in Figure 9, without prefetch-based mitigations, PRIME+PROBE attack is sufficient to precisely leak the entire T-Table access patterns. However, when preload operations load the entire T-Table to the L1 cache sets before the actual AES encryption, PRIME+PROBE can no longer distinguish real access patterns. The leakage captured by PRIME+PROBE under the prefetch-based defense shows as if the victim process accessed entire T-Tables during the AES encryption operation. However, we can accurately distinguish the actual T-Table accesses with PRIME+RETOUCH even when prefetch-based defenses are applied, both in regular environments and in Intel SGX enclaves. Note that the PRIME+PROBE without prefetch and PRIME+RETOUCH with prefetch have a similar result, proving that PRIME+RETOUCH is as effective as traditional cache side-channel attacks in regular environments as well as in Intel SGX enclaves. In both attacks, we can clearly see the diagonal cache access traces. Also, 40% to 60% of the victim’s accesses are captured



**Fig. 9:** Color matrix showing cache hits on AES T-Table encryption. Darker means more L1 cache hits. P+P indicates PRIME+PROBE. We conducted four different experiments on the same AES implementation. The first implementation was conducted with PRIME+PROBE without the prefetch-based defense applied. The other experiments were performed with both PRIME+PROBE and PRIME+RETOUCH, with prefetch-based mitigations applied.

for the other cache lines because we randomize the  $p_{1-15}$ . This indicates that PRIME+RETOUCH is accurate enough to capture actual access patterns even under the prefetch-based mitigation, which is not possible with eviction-based side-channel attacks. Another advantage of PRIME+RETOUCH is stealthiness. We observe that only 2.169% of the executions incur TSX faults while launching PRIME+RETOUCH. Among all TSX faults, 10% were due to cache misses (*i.e.*, 0.2% of all executions) while the rest were caused by other TSX faulting conditions, leaving minimal attack traces and making the attack hardly detectable. This indicates that most of the TSX faults were caused by running AES encryption in one transaction and irrelevant to the attack with PRIME+RETOUCH. To verify it, we launched the same AES implementation without involving the PRIME+RETOUCH attack, and observed again that around 2% of the executions incurred TSX faults. In contrast, when we launched the traditional PRIME+PROBE attack against the same AES implementation protected by the prefetch-based defense, 93.445% of the executions triggered TSX faults.

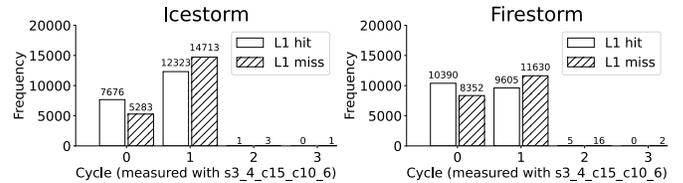
## VII. PRIME+RETOUCH AGAINST M1

**Execution environment.** We used the Apple Mac Mini with the M1 CPU consisting of four performance cores (Firestorm) and four efficiency cores (Icestorm). Each Firestorm core has 128KB of L1 data cache; and each Icestorm core has 64KB of L1 data cache. The machine is equipped with 16GiB of memory and runs a custom build Ubuntu 5.11.0-rc4+. We assume that the victim adopts naive prefetching without locking because M1 has no support for transactional memory. However, we assume that the victim’s entry should not be evicted while launching the PRIME+RETOUCH. Also, we assume that the victim and attacker processes run under time-sliced sharing setting such as pthread or user-level threading. Last, we assume that the attacker has no root privilege.

### A. Challenge of PRIME+RETOUCH on M1

To determine whether the victim has accessed the prefetched data, the attacker should be able to utilize PMU or high resolution timer as L1 cache activity monitor. However, unprivileged processes (running at ELO) are not authorized to use those measurement tools by default on the M1 platform.

**Interface of ARM system counter is hidden.** ARMv8 architecture employs a system counter [22] which provides a fixed frequency incrementing system count giving an equivalent view of the passage of time. Fortunately, the system timer is accessible to unprivileged users through the ARM-recommended register interface, CNTVCT\_EL0. However, we found that reading the system counter using this interface raises an illegal instruction fault on the M1 platform. Nonetheless, we could discover that an undocumented system register `s3_4_c15_c10_6` is used as an alias of CNTVCT\_EL0. To locate this unknown interface, we cleverly leveraged Apple’s Rosetta2 [5] which translates x86 instructions to the ARM64 counterparts. Because x86 provides a user-accessible system timer through `rtdsc` instruction, we made Rosetta2 translate the x86 binary containing `rtdsc` instruction and disassembled the translated ARM64 binary. As a consequence, we can ascertain that the `s3_4_c15_c10_6` is utilized to access system counter at ELO.



**Fig. 10:** Latency of a `ldr` measured with the coarse-grained system timer. The measurement iterated 20000 times for L1 hit and L1 miss respectively. 0 cycle indicates that the latency of load operation is too small to be measured with the system counter.

**System counter is too coarse-grained to differentiate between L1 hit and miss events.** In §III-B, we confirmed that M1 architecture exhibits a clear distinction between the L1 hit and miss latency. The latency differences are around 10 and 12 core clock cycles on Icestorm and Firestorm, respectively Figure 2. However, it is known that Firestorm core runs at 3.2GHz, but the system counter increases with a frequency in the range of 1MHz to 50MHz according to [22]. Therefore, the two events cannot be reliably distinguished due to the coarse granularity of the system timer. To validate the claim, we measured the elapsed time for accessing one memory address using the system timer. Because we have full knowledge about the eviction policy of L1 data cache, we can easily produce L1 hit and miss conditions on the measurement. As shown in Figure 10, there is no clear distinction in between the two events. When load operation finishes at 1 system clock cycle, it can be either L1 hit event or L1 miss event with 45.57% and 54.42% chance, respectively. For Firestorm, it was 45.23% and 54.76%. Note that the similar ambiguity is also present in the 0 cycle case in both cores. In summary, the latency difference between the L1 hit and miss event is too negligible to be distinguished with the coarse-grained system counter. Therefore, we need another approach to infer victim’s access in PRIME+RETOUCH under M1.

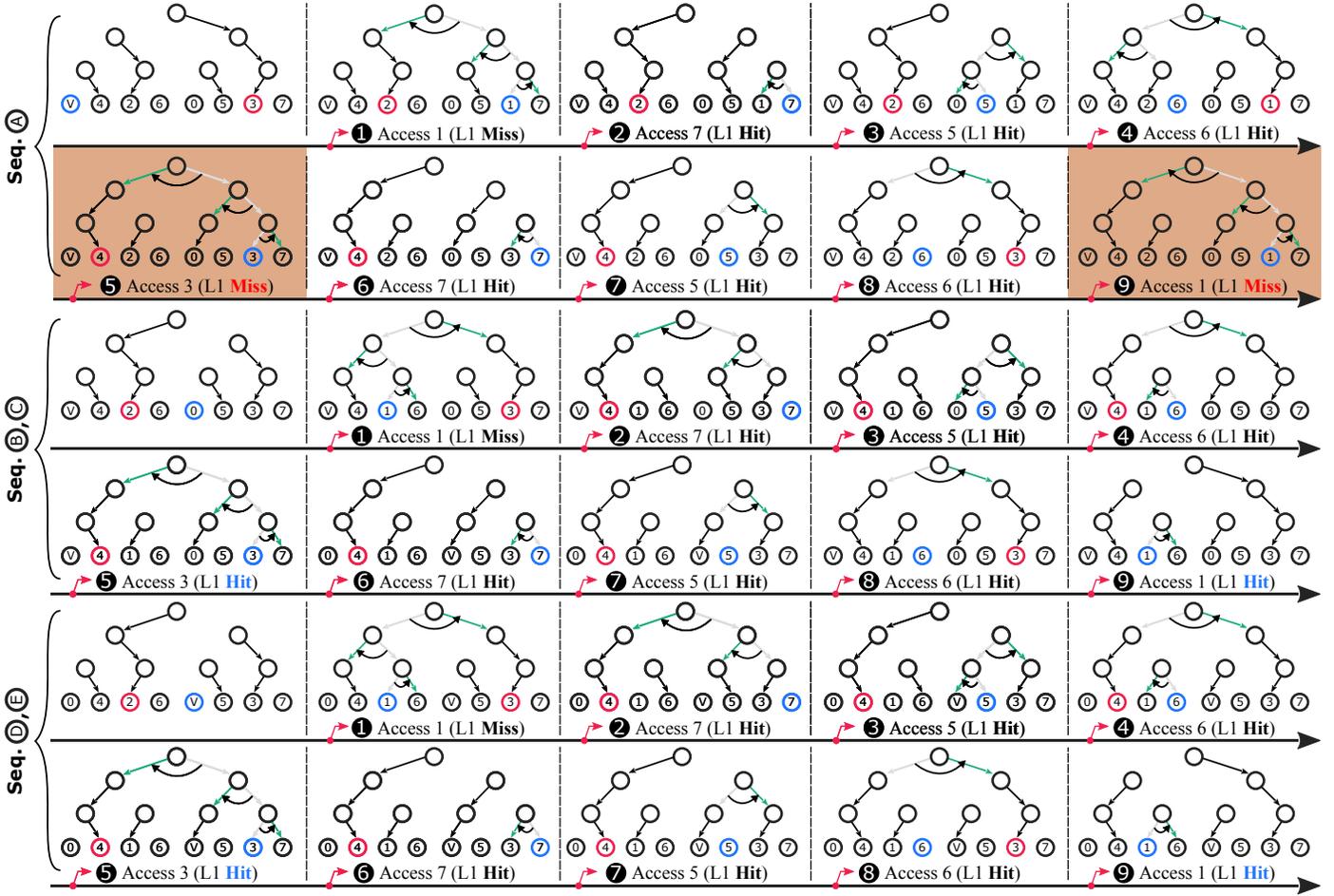


Fig. 11: Access sequence designed for introducing L1 miss events only for the operation sequence (A).

### B. Amplifying weak signal with Tree-PLRU

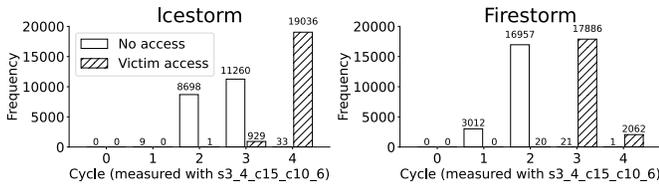
We revise PRIME+RETOUCH by leveraging characteristics of Tree-PLRU so that the attacker can deliberately produce more miss events only when a particular operation sequence happens. Although the latency of a single miss event cannot be captured, when more miss events are accumulated, it will become large enough to be measured with the coarse-grained system timer. To infer which operation sequence happens, the attacker has to make one operation sequence always produce more L1 cache miss events than the others in the measurement phase. In detail, the attacker issues a particular memory access sequence designed to introduce additional L1 miss events only for the operation sequence (A) where the victim accesses the prefetched data. It is worth noting that if the access sequence introduces the same amount of L1 misses to all other operation sequences (B) to (E), the attacker cannot tell which operation sequence has occurred.

**Exploiting Tree PLRU state to produce more L1 cache misses.** It is non-trivial to generate an access sequence satisfying such a condition because the attacker should consider additional state changes made on the other result sequence together. The major principal in constructing the proper memory sequence is maintaining at least one different entry in all possible Tree-PLRU states after every access. When we have

a different entry, we can easily generate different cache events for the same memory accesses because one operation sequence has entries that are not owned by the others and vice versa. However, as shown in Figure 11, initially, all five operation sequences have the same cache entries. Nonetheless, we can satisfy the principal because the Tree-PLRU state of sequence (A) and the others designate different entries as the next eviction target due to the PLRU aware RETOUCH. In step 1, accessing cache line 1 makes any operation sequence to produce miss events. However, it makes only the sequence (A) own different entry 2 compared to the others (note that 2 is evicted from the others and 3 remains instead). After that, through steps 2-4, we should manipulate the Tree-PLRU state of (A) so that the further L1 miss events always replace the entry recently brought to the set (1). Under such Tree-PLRU settings, fetching 3 will result in a cache miss to (A), but will result in a cache hit to the others (B-E).

### Amplified L1 miss event can distinguish victim's access.

To show that the system timer can sufficiently capture the victim access with the L1 miss signal amplification technique, we implement a simplified victim program that prefetches single data block mapped to a particular L1 data cache set. Also, to minimize the noise induced by time-slicing and show the effectiveness of PRIME+RETOUCH in M1, we utilize synchronization primitives to yield the victim program



**Fig. 12:** The access sequence comprises of 57 load operations repeatedly accessing cache entries in the following order  $\overline{75637561}$ . It makes the sequence  $\textcircled{A}$  (case of victim access) produce 14 additional misses compared to others.

for the attacker to retouch in time. The attacker launches PRIME+RETOUCH and produces additional memory access sequences described in Figure 11 in the measurement. We empirically found that 14 additional misses are enough to clearly distinguish genuine victim accesses in both types of core. We can produce one additional miss every four memory access operations. Therefore, to introduce 14 additional misses, we added 57 additional load operations including the first access  $\textcircled{1}$ . Also, we measure the latency to execute 57 additional load misses with the system timer. Note that the miss happens only when the sequence  $\textcircled{A}$  happens. Therefore, as shown in Figure 12, only when the victim accesses data after the prefetch (hatched bar plot) it incurs additional clock cycles to run 57 load operations. Compared to result shown in Figure 10 that measure only one miss event with the system timer, our results clearly distinguish the genuine victim access. Although we demonstrated PRIME+RETOUCH on the restricted environment to show its effectiveness, we believe that the attack can be reproducible against realistic examples.

## VIII. RELATED WORK

**Tree-PLRU cache eviction policy abuses.** [53] establishes covert channel by encoding one-bit information with the next eviction target based on the Tree-PLRU eviction policy. Also, [33] theoretically shows how the shared Tree-PLRU state can break the security of cache partitioning defenses. A noticeable difference between PRIME+RETOUCH and [33, 53] is that it can precisely recognize actual accesses even under the presence of preloading and locking defense. In detail, RETOUCH operation on an attacker’s primed cache line manipulates the Tree-PLRU state in such a way that a following victim’s access reveals its existence. However, under the same condition, previous attacks cannot capture the victim’s access because prefetch-based defense eliminates the leakage. Furthermore, the previous attacks have not been fully demonstrated with realistic and detailed examples regarding the Tree-PLRU eviction policy. Also, PRIME+RETOUCH provides additional synchronization information for the attacker to reduce attack noises, which has not been explored in the previous works.

**Quad-LRU cache eviction policy abuses** [10] shows that the Quad-age LRU (QLRU) eviction policy [30] can be exploited to leak the victim’s access pattern from the L3 cache. In essence, the QLRU records the number of accesses made to a specific entry with a 2-bit counter dedicated to each entry. Due to this property of QLRU, consecutive accesses to one cache line back-to-back can be clearly observed as two separate accesses. However, the Tree-PLRU can track only the access order among the cache entries, not the number

of accesses. Therefore, it is more challenging to distinguish the victim’s access when the entries are prefetched before the access. PRIME+RETOUCH shows that RETOUCH allows the attacker to capture the victim’s access despite the restriction of Tree-PLRU eviction policy. Furthermore, PRIME+RETOUCH does not evict the victim’s cache line, but [10] can induce such evictions depending on the victim’s accesses. When preloading and locking defense is deployed, eviction of the victim’s cache line will be detected immediately, which can make the reloading operations meaningless. Note that this difference makes the PRIME+RETOUCH more stealthy. Lastly, [10] requires shared memory, which is not a requirement of PRIME+RETOUCH.

**Apple M1 on Asahi Linux.** [1] is an ongoing project that aims at running Linux operating system on Apple M1 architecture. As part of the porting, they also provide a list of Apple’s system registers, including configuration registers for Apple’s proprietary PMU. However, their Linux version does not include PMU support at the time of submission. Thus, to the best of our knowledge, this is the first work that fully demonstrated how the Apple PMUs and its undocumented events can be utilized to reveal the underlying cache architecture.

**Google’s Tree-PLRU attack.** As a concurrent work, Google showed that a single read of secret data is enough to leak data efficiently by abusing the characteristic of the Tree-PLRU cache eviction strategy in their blog post [16]. Compared to Google’s work, PRIME+RETOUCH further demonstrates that Tree-PLRU cache eviction side channel can even efficiently bypass the prefetching-and-locking defenses. Also, we provide in-depth micro-architecture analysis on Intel x86 and Apple M1 architectures, lacking in Google’s article, which validates our claims. More importantly, we reported our attack to Intel and received an acknowledgment about the attack method before the Google’s publication<sup>1</sup>.

## IX. CONCLUSION

One of the most important characteristics of PRIME+RETOUCH that differentiates it from other cache side-channel attacks is full awareness of eviction policy, especially about Tree-PLRU. Based on the comprehensive understanding of Tree-PLRU policy we designed PLRU Aware RETOUCH that can completely break prefetch and locking defense. We demonstrated our attack is feasible by showing the leakage in AES T-Table encryption, which cannot be achievable in the eviction based side-channels. Furthermore, we demonstrates that PRIME+RETOUCH can leak victim’s access pattern even without access to the fine-grained timer on the M1 platform. Also, we first provide detailed analysis of L1 data cache on M1 platform.

<sup>1</sup>We first reported our attack to Intel on Nov 3, 2020, and Google posted the article on March 12, 2021.

## REFERENCES

- [1] “Asahi linux,” <https://asahilinux.org/>, [Online; accessed 22-July-2021].
- [2] A. Abel and J. Reineke, “nanobench: A low-overhead tool for running microbenchmarks on x86 systems,” *arXiv preprint arXiv:1911.03282*, 2019.
- [3] O. Aciğmez and W. Schindler, “A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl,” in *Cryptographers’ Track at the RSA Conference*. Springer, 2008, pp. 256–273.
- [4] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar, “Fine grain cross-vm attacks on xen and vmware are possible!” *IACR Cryptol. ePrint Arch.*, vol. 2014, p. 248, 2014.
- [5] Apple, “About the Rosetta Translation Environment,” <https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment>, [Online; accessed 24-May-2021].
- [6] C. Ashokkumar, B. Roy, M. B. S. Venkatesh, and B. L. Menezes, “S-box implementation of aes is not side channel resistant,” *Journal of Hardware and Systems Security*, vol. 4, no. 2, pp. 86–97, 2020.
- [7] D. J. Bernstein, “Cache-timing attacks on aes,” 2005.
- [8] F. Brasser, U. Müller, A. Dmitrienko, K. Kostianen, S. Capkun, and A.-R. Sadeghi, “Software grand exposure: SGX cache attacks are practical,” in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, Aug. 2017.
- [9] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth, “Cacheshield: Detecting cache attacks through self-observation,” in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 224–235.
- [10] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, “Reload+ refresh: Abusing cache replacement policies to perform stealthy cache attacks,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
- [11] S. Briongos, P. Malagón, J. L. Risco-Martín, and J. M. Moya, “Modeling side-channel cache attacks on aes,” in *Proceedings of the Summer Computer Simulation Conference*, ser. SCSC ’16. San Diego, CA, USA: Society for Computer Simulation International, 2016.
- [12] M. Chiappetta, E. Savas, and C. Yilmaz, “Real time detection of cache-based side-channel attacks using hardware performance counters,” *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.
- [13] H. Cho, J. Park, D. Kim, Z. Zhao, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, “Smokebomb: effective mitigation against cache side-channel attacks on the arm architecture,” in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 107–120.
- [14] C. Disselkoe, D. Kohlbrenner, L. Porter, and D. Tullsen, “Prime+abort: A timer-free high-precision l3 cache attack using intel tsx,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 51–67.
- [15] Q. ge, Y. Yarom, D. Cock, and G. Heiser, “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware,” *Journal of Cryptographic Engineering*, vol. 8, pp. 1–27, 10 2016.
- [16] Google, “A Spectre proof-of-concept for a Spectre-proof web,” <https://security.googleblog.com/2021/03/a-spectre-re-proof-of-concept-for-spectre.html>, [Online; accessed 22-July-2021].
- [17] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, “Strong and efficient cache side-channel protection using hardware transactional memory,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 217–233.
- [18] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+ flush: a fast and stealthy cache attack,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.
- [19] D. Gruss, R. Spreitzer, and S. Mangard, “Cache template attacks: Automating attacks on inclusive last-level caches,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 897–912.
- [20] S. Gueron, “Intel advanced encryption standard (aes) instructions set,” *Intel White Paper, Rev.*, vol. 3, pp. 1–94, 2010.
- [21] A. Holdings, “Transactional Memory Extension (TME) intrinsics,” <https://developer.arm.com/documentation/101028/0009/Transactional-Memory-Extension--TME--intrinsics>, [Online; accessed 24-May-2022].
- [22] —, *AArch64 Programmer’s Guides Generic Timer*, August 2019, no. ARM062-1010708621-30.
- [23] —, “Arm architecture reference manual, armv8, for armv8-a architecture profile,” 2019.
- [24] —, “Arm architecture reference manual supplement armv9, for armv9-a architecture profile,” 2021.
- [25] M. S. Inci, B. Gülmezoglu, G. I. Apecechea, T. Eisenbarth, and B. Sunar, “Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud,” *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 898, 2015.
- [26] Intel, “Intel® Transactional Synchronization Extensions Asynchronous Abort,” <https://software.intel.com/security-software-guidance/advisory-guidance/intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort>, 2019-11-12, [Online; accessed 22-September-2020].
- [27] —, “List of processors affected by transient execution attack,” <https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model>, 2020-09-11, [Online; accessed 24-September-2020].
- [28] G. Irazoqui, T. Eisenbarth, and B. Sunar, “S \$ a: A shared cache attack that works across cores and defies vm sandboxing—and its application to aes,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 591–604.
- [29] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, “Wait a minute! a fast, cross-vm attack on aes,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 299–319.
- [30] S. Jahagirdar, V. George, I. Sodhi, and R. Wells, “Power management of the third generation intel core micro architecture formerly codenamed ivy bridge,” in *2012 IEEE Hot Chips 24 Symposium (HCS)*. IEEE, 2012, pp. 1–49.
- [31] E. Käsper and P. Schwabe, “Faster and timing-attack resistant aes-gcm,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer,

- 2009, pp. 1–17.
- [32] T. Kim, M. Peinado, and G. Mainar-Ruiz, “STEALTH-MEM: System-level protection against cache-based side channel attacks in the cloud,” in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 189–204.
- [33] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, “Dawg: A defense against cache timing attacks in speculative execution processors,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 974–987.
- [34] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *40th IEEE Symposium on Security and Privacy (S&P’19)*, 2019.
- [35] R. Könighofer, “A fast and cache-timing resistant implementation of the aes,” in *Cryptographers’ Track at the RSA Conference*. Springer, 2008, pp. 187–202.
- [36] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas, “Spydetector: An approach for detecting side-channel attacks at runtime,” vol. 18, no. 4, pp. 393–422, Aug. 2019.
- [37] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- [38] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, “Catalyst: Defeating last-level cache side channel attacks in cloud computing,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 406–418.
- [39] F. Liu and R. B. Lee, “Random fill cache architecture,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 203–215.
- [40] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.
- [41] A. Moghimi, G. Irazoqui, and T. Eisenbarth, “Cachezoom: How sgx amplifies the power of cache attacks,” in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 69–90.
- [42] M. Neve and J.-P. Seifert, “Advances on access-driven cache attacks on aes,” in *International Workshop on Selected Areas in Cryptography*. Springer, 2006, pp. 147–162.
- [43] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: The case of aes,” in *Proceedings of the 2006 The Cryptographers’ Track at the RSA Conference on Topics in Cryptology*, ser. CT-RSA’06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 1–20.
- [44] C. Rebeiro, D. Selvakumar, and A. Devi, “Bitslice implementation of aes,” in *International Conference on Cryptology and Network Security*. Springer, 2006, pp. 203–212.
- [45] T. Ristenpart, E. Tromer, H. Shacham, and S. 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*, 2009, pp. 199–212.
- [46] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, “Zombieload: Cross-privilege-boundary data sampling,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.
- [47] K. So and R. N. Rechtschaffen, “Cache operations by mru change,” *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 700–709, 1988.
- [48] E. Tromer, D. A. Osvik, and A. Shamir, “Efficient cache attacks on aes, and countermeasures,” *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.
- [49] J. Van Bulck, F. Piessens, and R. Strackx, “Sgx-step: A practical attack framework for precise enclave execution control,” in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 1–6.
- [50] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “Ridl: Rogue in-flight data load,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 88–105.
- [51] P. Vila, P. Ganty, M. Guarnieri, and B. Köpf, “Cachequery: learning replacement policies from hardware caches,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 519–532.
- [52] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 494–505.
- [53] W. Xiong and J. Szefer, “Leaking information through cache lru states,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 139–152.
- [54] Y. Yarom and K. Falkner, “Flush+ reload: a high resolution, low noise, l3 cache side-channel attack,” in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.
- [55] T. Zhang, Y. Zhang, and R. Lee, “Cloudradar: A real-time side-channel attack detection system in clouds,” vol. 9854, 09 2016, pp. 118–140.
- [56] Z. Zhou, M. K. Reiter, and Y. Zhang, “A software approach to defeating side channels in last-level caches,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 871–882.

## APPENDIX

### A. Implementation of reversing L1 cache on M1

```

1 static void attack(void *data){
2     L1Cache =(u64 *) kcalloc(8U * 256 * 64, GFP_KERNEL);
3     //chaining the addresses to generate dependency
4     for (int index = 0; index < CACHE_WAYS+16; index++) {
5         *((u64 *)((char *)L1Cache + (index * PAGE_SIZE))) =
6             (u64)((char *)L1Cache + ((index + 1) * PAGE_SIZE));
7     }
8
9     u64 pmcr0 = (0x1 << 2) | (0x1 << 3) ;
10    u64 pmcr1 = ((0x1 << 2) | (0x1 << 3)) << 16;
11    //pmc2: Core clock cycle timer (0x2)
12    //pmc3: L1 cache miss event monitor (0xa3)
13    u64 pmesr0 = (0x2 | (0xa3 << 8));
14
15    asm volatile(
16        "Enable_Apple_PMUs:\n\t"
17        //Initializing PMCs as zero
18        "msr S3_2_C15_C0_0, xzr\n\t" //PMC0
19        "msr S3_2_C15_C2_0, xzr\n\t" //PMC2
20        "isb \n\t"
21        //set PMESR0 to select event
22        "msr S3_1_C15_C5_0, %6 \n\t"
23        "isb \n\t"
24        //set PMCR1 to enable PMC0 and PMC2
25        "msr S3_1_C15_C1_0, %5 \n\t" //PMCR1
26        //enable pmc2 again
27        "msr S3_1_C15_C0_0, %4 \n\t" //PMCR0
28        "isb \n\t"
29        "Prime_one_set:\n\t" //fill L1 data cache set
30        "ldr x13, [%7]\n\t" "ldr x13, [x13]\n\t"
31        "ldr x13, [x13]\n\t" "ldr x13, [x13]\n\t"
32        "ldr x13, [x13]\n\t" "ldr x13, [x13]\n\t"
33        "ldr x13, [x13]\n\t" "ldr x13, [x13]\n\t"
34        "Read_operations_to_change_LRU_state\n\t:"
35        "isb\n\t" "ldr x14, [%7]\n\t" "isb\n\t"
36        "Eviction:\n\t" //Read one more to kick out
37        "ldr x13, [x13]\n\t" "isb\n\t"
38        "Measurement:\n\t"
39        "mrs %0, S3_2_C15_C3_0 \n\t" //pmc3
40        "mrs %2, S3_2_C15_C2_0 \n\t" "isb\n\t" //pmc2
41        "ldr x14, [%10]\n\t" //Any cache line to monitor
42        "mrs %0, S3_2_C15_C2_0 \n\t" //pmc2
43        "mrs %2, S3_2_C15_C3_0 \n\t" "isb\n\t" //pmc3
44        : "=r"(pmc[0]), "=r"(pmc[1]), "=r"(pmc[2]), "=r"(pmc[3]),
45        : "r" (pmcr0), "r"(pmcr1), "r"(pmesr0), "r"(L1Cache)
46        : "memory", "x13", "x14"
47    );
48    printk("Elapsed cycle:%ld\n",pmc[1]-pmc[0]);
49    printk("L1 miss      :%ld\n",pmc[3]-pmc[2]);
50 }

```

**Fig. A.1:** Basic reversing code for L1 DCache on M1. PMC2 and PMC3 has been set to track core clock cycle counter and L1 data cache miss event, respectively

**System register configurations for Apple’s PMU.** To use the Apple’s PMU, several system registers should be correctly configured. The PMCR0 register mapped to s3\_1\_c15\_c0\_0 controls basic capabilities of PMU such as which Performance Monitoring Counter (PMC) needs to be enabled (bit 0-7), which interrupt mode of PMUs will be used (bit 8-10), and authorizing unprivileged user’s access on the PMU (bit 30). By default, the bit dedicated to enabling user access on PMU is disabled on macOS so we also disabled that bit. We checked the default PMCR0 register value on the latest macOS, BigSur 11.4, by loading the kext module. Also, because our measurement is done in a very short time window, there is no need for setting interrupt-related information for PMU control. Therefore, we just set two bits for enabling PMC2 and PMC3 (Line 9). Apple provides another PMU control register called PMCR1 mapped to system register s\_1\_c15\_c1\_0. This register controls which execution modes count events. We utilize the Linux driver

instead of user process, which runs at EL1 privilege (kernel). Therefore, we set the PMCR1 register to allow our kernel module to access the PMC2 and PMC3. Because the 8 bits from 16 to 23 is dedicated to controlling PMC0 to PMC7 on the EL1 privilege, we set the two bits as shown in the Line 10. Lastly, the PMESR0 register mapped to S3\_1\_C15\_C5\_0 provides a way to select PMU events for PMC2 to PMC5. Each PMC can track PMU events 0 to 254, and 8 bits are assigned per PMC. The least significant bit 0 to 7 are used to select PMU event of PMC2, and the next 8 bits are dedicated for PMC3 event selection.

**Measuring the events using the PMU.** After the PMU registers are properly initialized, one can measure the event by reading the PMC register assigned for tracking specific events. As shown in Line 39-40 and Line 42-43, to measure the events caused by one memory load operation (Line 41), PMC registers should be read before and after the memory operation using mrs instruction. Also, to prevent the instruction reordering between mrs and ldr instruction, we carefully insert Instruction Synchronization Barriers (ISB). To reverse engineer the eviction policy of the L1 cache on the M1 platform, we follow the reversing logic described in Algorithm 1. First, we fill a specific cache set by issuing multiple memory load operations to the addresses mapped to the same set (Line 29-33). And then, we enumerate any combinations of read operations (Line 34-35) to change the underlying state of the eviction policy. By accessing one more entry mapped to the same set (Line 36-37), we can evict one entry from the set. After the eviction, we measure the latency of accessing a particular memory address we used for filling the L1 cache set (Line 29-33). Based on the latency difference measured in Figure 2, we can determine which cache line has been evicted and recover the eviction policy from the trace.