Leaking Information Through Cache LRU States in Commercial Processors and Secure Caches

Wenjie Xiong, Stefan Katzenbeisser, Jakub Szefer

Abstract—The Least-Recently Used (LRU) cache replacement policy and its variants are widely deployed in modern processors. This paper shows in detail that the LRU states of caches can be used to leak information: any access to a cache by a sender will modify the LRU state, and the receiver is able to observe this through a timing measurement. This paper presents LRU timing-based channels both when the sender and the receiver have access to shared memory, e.g., shared library, and when they are separate processes without shared memory. In addition, the new LRU timing-based channels are demonstrated on both Intel and AMD processors in scenarios where the sender and the receiver are sharing the cache in both hyper-threaded setting and time-sliced setting. The transmission rates of the LRU channels can be up to 600Kbps per cache set in the hyper-threaded setting. Different from the majority of existing cache channels which require the sender to trigger cache misses, the new LRU channels work with the sender only having cache hits, making the channel faster and stealthier. This paper further discusses the effectiveness of the new LRU channels against a number of secure cache designs. Especially, the LRU channels are demonstrated to work against two representative secure caches, Partition-Locked (PL) cache and Random Fill (RF) cache, in the `gem5` simulator, showing possible vulnerabilities in the secure cache designs in which the security of the replacement state is not protected properly.

Index Terms—Processor Caches, Covert Channels, Cache Replacement Policy, LRU

1 Introduction

Side channels and covert channels in processors have been gaining renewed attention in recent years [1]. Many of these channels leverage timing information. To date, researchers have shown numerous timing-based channels in caches, e.g., [2], [3], as well as other parts of the processor, such as the shared functional units in simultaneous multithreading (SMT) processors, e.g., [4], [5], [6], [7], [8]. The canonical example of timing channels are channels in caches, where timing reveals information about cache states, e.g., a fast access due to a cache hit and a slow access due to a cache miss. These side channels and covert channels can be used to leak information, such as cryptographic keys, e.g., [9]. Further, many of the variants of the recent Spectre and Meltdown attacks also use covert channels, in addition to transient execution, to exfiltrate data, e.g., [10], [11], [12].

In processor caches, the order in which the cache lines are evicted depends on the cache replacement policy. Normally, different variants of the Least-Recently Used (LRU) policy are implemented in modern processors, such as Tree-PLRU [13] or Bit-PLRU [14]. In a cache, the LRU state is maintained for each cache set, and it is used to determine which cache line in the cache set should be evicted when there is a cache miss causing a cache replacement. The LRU state is updated on every cache accesses to indicate which cache line in the set was just accessed. Thus, both cache hits and misses cause updates to the LRU state of cache lines.

The basis of the new LRU timing-based channels presented in this paper is that different LRU states result in timing differences in future memory accesses. Based on the message to be transmitted, the sender first accesses some memory location, which will update the LRU states. Then, the receiver tiggers a cache replacement, which depends on the LRU state. The receiver further measures the timing of a memory access to learn the previous LRU states and infers the sender's access pattern. Note that the LRU states get updated even if the sender only causes cache hits, which is different from most other cache side channels where a cache replacement (i.e., a cache miss) by the sender is required to change the cache state [2], [3]. The attacks in this paper are stealthier compared to the prior work because they do not require a cache miss by the sender. The new attacks can also bypass defenses such as based on performance counters [15], where behavior of cache missies is monitored. Moreover, lack of required misses for the sender benefits the transient execution attacks such as Spectre and Meltdown [10], [11], as only a small speculation window is required for the sender to trigger a cache hit.

In this paper, the new LRU timing-based channels are demonstrated and evaluated in-depth. Two algorithms are designed to build LRU timing channels: both with and without shared memory between the sender and the receiver, making the LRU channels practical in a variety of attack scenarios. We give the details of the cache replacement state during attacks in a set-associative cache to highlight the novelty of the proposed channels. The LRU timing channels are demonstrated in several different commercial processors. The bandwidth and the error rates of the channels are evaluated to show the effectiveness of of the new attacks.

The new LRU timing-based channels are also a threat to many of the existing secure cache proposals, which in turn are often included as part of secure processor architectures. Numerous secure caches [16], [17], [18], [19], [20],

- *W. Xiong and J. Szefer are with the Department of Electrical Engineering, Yale University, New Haven, CT, 06511. E-mail: wenjie.xiong@aya.yale.edu, jakub.szefer@yale.edu*
- *S. Katzenbeisser is with University of Passau, Passau, Bayern, Germany. E-mails: stefan.katzenbeisser@uni-passau.de*

[21], [22], [23], [24], [25] have been presented, and they aim to either partition or randomize the victim's and the attacker's cache accesses to defend against cache timing-based side channels. However, many of the secure caches have not considered the LRU states and are vulnerable to the new LRU channel. This paper demonstrates that the LRU channels are effective against the Partition-Locked (PL) cache [17] and the Random Fill (RF) cache [22] in the `gem5` simulator. This paper further categorizes other secure caches and discusses the effectiveness of different secure caches in defending against the proposed LRU channels. This paper is an extension of our conference paper [26] with the following new contributions:

- We extended the evaluation of the LRU covert-channel attacks using *Hamming Distance*. Compared to the *Edit Distance* that was evaluated in [26], we show Hamming Distance is more practical considering existing error correction algorithms.
- We evaluated the LRU channels in one additional secure cache design, the Random Fill (RF) cache, using the `gem5` simulator, showing that randomized caches may fail to defend against the proposed channels.
- In addition to simulation-based evaluation, we qualitatively analyzed the effectiveness of the channels in a number of secure cache proposals leveraging different strategies. The analysis shows mixed results. Only certain secure cache proposals protect against our channels leveraging the cache replacement state.

We open sourced the code of our covert-channel attacks at caslab.csl.yale.edu/code/cache-lru-attack/.

2 Background

2.1 Timing-Based Cache Channels

In side channel attacks, the attacker (receiver) tries to learn the victim's information by observing the side effects of the victim's execution. In covert channel attacks, the sender (or a trojan in the victim code) intentionally sends messages to the receiver. In both attacks, a communication channel is required to be built across security boundaries.

There are typically two types of timing-based side and covert channels. One type leverages contention, for example, port contention [4], [7], or contention in the cache bank[1] [5], [6]. This type of channels require the sender and the receiver to execute concurrently as two hyper-threads to cause contention. The other type leverages the states, e.g., tag state in the cache [2] or the cache coherence state [3]. Channels using cache states leverage the fact that whether a cache line is available in the cache or not affects the timing of the cache operations. The sender and the receiver do not have to be two concurrent hyper-threads. They can be part of one thread or share the cache in a time-sliced setting. All the existing cache timing-based channels using states, however, require a cache miss by the sender to change the cache state when the sender is sending information. For example, in Flush+Reload attacks [2], the sender will need to access the cache line that was previously flushed to memory by the receiver. Thus, the access will cause a cache miss. Meanwhile, any cache access, both cache hit or miss, can trigger the new LRU attack.

Usually a side or covert channel leveraging cache state includes the following three phases. **1. Initialization Phase:** First, a sequence of memory accesses is performed so that the cache state is (partially) known to the receiver. **2. Encoding Phase:** To send information, the sender accesses one or more memory locations to change the cache state. The pattern of memory accesses depends on the information to be sent. A light-weight encoding phase is desired, especially for side channel attacks, where the attacker does not have control of the victim code. If the encoding phase only needs one memory access, a single secret-dependent access could make a piece of software vulnerable. **3. Decoding Phase:** The receiver then observes the time required to access the memory location to learn the cache state.

1. To increase the cache bandwidth, the Intel L1 cache is composed of multiple banks, where each bank serves part of a cache line. Each bank can only serve one access request at a time.

2.2 Cache Replacement Policy

When a cache line is accessed but it is not in the cache (i.e., a cache miss), the cache line will be fetched into the cache set. In this case, another cache line needs to be evicted from the cache set to make room for the incoming cache line. The replacement policy selects a cache way from the set to evict, known as the *victim way*. The replacement algorithm uses some state to store the history of accesses to cache ways in each set. In the L1 cache, the LRU policy and its variants are most widely used because they give a high cache hit rate. In the last level cache (LLC), due to the reduced data locality, other replacement policies can be used [27], [28].

LRU: The LRU algorithm keeps track of the age of cache lines. If a cache replacement is needed on a cache miss, the least recently used cache way (i.e., oldest way) will be selected to be the victim way and will be evicted. In an N-way cache, $log(N)$ bits are used per cache line per way to store the age of the line, requiring a total of $Nlog(N)$ bits for each cache set.

Pseudo Least-Recently Used (PLRU): The "true" LRU algorithm is expensive in terms of latency (to update LRU states) and area (to store the age of all cache lines). So often a variant called Pseudo Least-Recently Used (PLRU) that uses simpler states is used instead. Tree-PLRU [13] policy and Bit-PLRU [14] policy are two commonly used PLRU policies. Both use $O(N)$ bits to store the relative age of the cache lines. In PLRU policy, it is not guaranteed that the least recently used line will be evicted, but on average a less recently used cache line will be selected.

3 Threat Model and Assumptions

We assume N-way set-associative caches and further assume the cache uses an LRU, Tree-PLRU, or Bit-PLRU replacement algorithm which evicts the least recently used cache line. Like all other side or covert channels, the LRU timing-based channel involves two parties: the sender and the receiver. Following techniques used in [29], [30], we assume the two parties are co-located on the same core and share the L1 cache, as shown in Figure 1, either in an SMT machine as two hyper-threads running in parallel or as two threads time-sharing the core. The LRU states of the shared

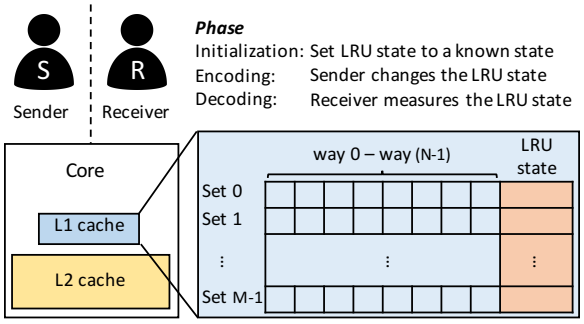

Fig. 1: Cache organization and the phases of the new LRU timing-based side and covert channels.

cache can be influenced by the sender and observed by the receiver. Existing attacks, such as side channels [6], [7], [8] or Spectre attacks using Branch Target Buffer (BTB) or Return Stack Buffer (RSB) [10], [31], show that sharing of the same physical core is practical and poses a real threat in modern computer systems.

In this paper, we focus on the LRU states in the L1 cache. LRU channels in the other levels of caches are also possible. However, depending on the cache architecture, for the sender to update the LRU states of the lower level of caches, a miss in the higher cache level is required, e.g., the sender's hits to L1 or L2 caches will not change the replacement state in the LLC. Especially, L1 is directly accessed by the processor pipeline and L1 LRU state is updated on every memory access. Thus, attacks using the LRU states of L1 are stealthier. Furthermore, timing channels in LRU states in L2 or LLC can be detected or protected by existing cache side channel detection or protection techniques in L1 and prefetching the secure-relevant data to L1.

For all types of attacks, we assume the receiver can extract useful information from the memory access pattern of the sender, which modifies the LRU states.

4 LRU Timing-based Channels

Our new LRU timing-based channels leverage the LRU states of cache sets. In this section, we discuss how the LRU state in *one* cache set can be used to transfer information, which is referred to as the *target set*.

The LRU state for each set contains several bits, thus it is possible to transfer more than one bit per target set. However, limited by the fact that any access to the set will change the LRU state, we focus on letting the receiver only measure the set once. Especially, the receiver can observe the timing of one memory access which can only have two results: a cache hit or a cache miss. Thus, at most one bit can be transferred per cache set at one time.

To transfer information using an LRU channel, we use three phases like in other cache channels (in Section 2.1). The main difference is in the third step, where the receiver first accesses one or more memory locations mapping to the target set. These accesses potentially trigger a cache replacement and cause a cache line to be evicted based on the LRU state. The receiver then observes the timing of accessing the memory location to learn if the cache line is evicted and thus infer what the LRU state was.

Algorithm 1: LRU Channel with Shared Memory

line 0–N: cache lines mapping to the target set
m: a 1-bit message to transfer on the channel
d: a parameter of the receiver

Receiver's Operations:

// Step 0: Initialization Phase
for $i = 0;\ i < d;\ i = i + 1$ **do**
  Access line i;
end
sleep; // To allow the sender to run here for encoding
// Step 2: Decoding Phase
for $i = d;\ i < N + 1;\ i = i + 1$ **do**
  Access line i;
end
Access line 0 and measure the latency of the access;

Sender's Operations:

// Step 1: Encoding Phase
if $m = 1$ **then**
  Access line 0;
else
  Do not access line 0;
end

4.1 LRU Channel with Shared Memory

Algorithm 1 shows a communication protocol using the LRU cache states assuming shared memory. The sender and the receiver first agree on the target cache set they will use to transfer information. We use the term *line* 0–N to denote N+1 different cache lines that map to the target set. This can be achieved by using data in N+1 different *physical addresses* with the same cache index bits but different tag bits. Note that *line* n (where $n \in [0, N]$) refers to a cache line with a certain physical address and not a specific cache entry, and the name does not imply certain literal physical address n. *line* n could be placed in any cache way in the set.

In Algorithm 1, the sender and the receiver both need to use the same physical address (or physical addresses within the cache line) to access cache line 0 in the cache. This can be achieved by a memory location in a shared dynamic linked library, as in [2]. Further, m is a 1-bit message to be sent, and d is a parameter indicating how the receiver's accesses are split between the initialization and decoding phase. Then, a channel can be built following Algorithm 1.

Figure 2 (left) shows an example of the states in the target cache set during an LRU covert channel attack using Algorithm 1, where $N = 8$ and $d = 8$. In the initialization phase, the receiver accesses 8 cache lines to set the LRU state of the cache set to a known state. In the encoding phase, the sender's access changes the LRU state depending on the message to be sent. In this step, the cache line accessed by the sender might already be in the cache, and the sender might always get cache hit in this step. In the decoding phase, the receiver first accesses a cache line that is not previously in the cache, e.g., line 8 here, to trigger a cache replacement. The cache line to be evicted is chosen based on the LRU state. The receiver then measures the time required to access cache line 0, to judge if line 0 got evicted. If the sender accessed line 0 in the encoding phase, an L1 cache hit will be observed by the receiver, otherwise, an L1 cache miss will be observed. Note that in PLRU policy, it is possible that

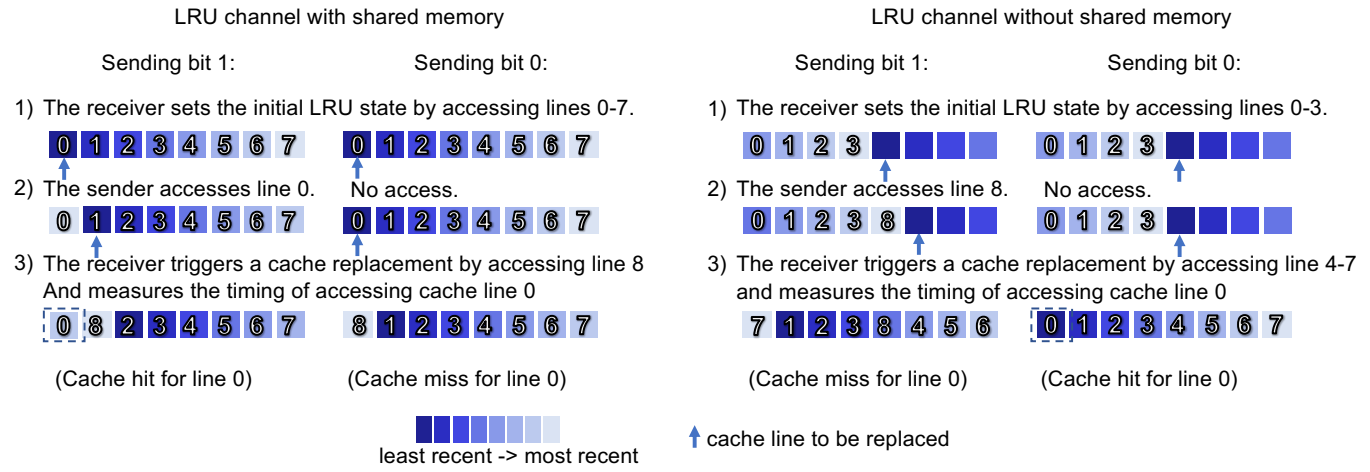

Fig. 2: States of a cache set during the LRU covert channel attacks. The numbers indicate cache lines of different addresses in the target set. Background colors indicate the age of the cache lines, with lighter color indicating more recent used cache lines. The arrow shows the cache line to be evicted under True-LRU policy if a cache replacement happens.

Algorithm 2: LRU Channel without Shared Memory

line 0–N: cache lines mapping to the target set
m: a 1-bit message to transfer on the channel
d: a parameter of the receiver

Receiver's Operations:

// Step 0: Initialization Phase
for $i = 0;\ i < d;\ i = i + 1$ **do**
  Access line i;
end
sleep; // To allow the sender to run here for encoding
// Step 2: Decoding Phase
for $i = d;\ i < N;\ i = i + 1$ **do**
  Access line i;
end
Access line 0 and measure the latency of the access;

Sender's Operations:

// Step 1: Encoding Phase
if $m = 1$ **then**
  Access line N;
else
  Do not access target set;
end

line 0 is not evicted even though it is the least recently used cache line. We will discuss the difference between PLRU and LRU in Section 4.3.

Comparing Algorithm 1 with Flush+Reload attacks [2], both require shared memory, but the LRU channel does not require an explicit flush, and line 0 might always be in the cache, i.e., the sender might only have cache hits.

4.2 LRU Channel without Shared Memory

In Algorithm 2, the sender and the receiver do not need to access any shared memory location. The sender and the receiver can map memory accesses to the target set by using proper virtual memory addresses in their own memory spaces. For performance reasons, L1 cache is usually virtual-indexed and physical-tagged (VIPT). For example, for an L1 cache with 64 sets with a cache line size of 64 bytes, bits 6–11 of the address decide the cache set. The receiver can make sure lines 0–$(N-1)$ map to the same set as line N by using memory locations with bits 6–11 of the virtual address identical to line N. Then, the sender and the receiver can build a channel following Algorithm 2.

Figure 2 (right) shows an example of the states in the target cache set in an LRU covert channel attack using Algorithm 2, where $N = 8$ and $d = 4$. In this example, the sender and the receiver do not share memory. The receiver has access to cache lines 0–7, and the sender has access to cache line 8. In the encoding phase, a potential access to line 8 will change the LRU state, which causes the receiver to observe different cache lines in the cache set in the decoding phase. The receiver will observe different timing when accessing line 0 in the last step.

Compared to Algorithm 1, there will be more noise in the channel by Algorithm 2, as any access to the target set can cause line 0 to be evicted. The noise is due to the absence of shared memory, and other cache side channel attacks (e.g., Prime+Probe channel [9]) also have this source of noise.

Comparing Algorithm 2 with Flush+Reload attacks, no shared memory is required. Comparing Algorithm 2 with Prime+Probe attacks [9], in Prime+Probe, the receiver will access the whole set in both the prime and the probe phases, and the sender will have a miss between the two phases. Meanwhile, in Algorithm 2, the receiver does not access the whole set in either phase. The receiver only needs to measure the time of one memory access in LRU channel rather than the time of N memory accesses in the Prime+Probe attack. Moreover, during a covert channel attack, cache misses on the sender's cache line N are not necessary. If the sender keeps accessing the same cache line N, then the sender's line N might be in the cache throughout the attack. This makes the attack stealthy if only the cache misses of the sender are monitored for defense.

In parallel to this paper, covert channels leveraging the replacement policy in LLC were proposed in [32]. However, [32] requires shared memory between the sender and the receiver, which is not necessary in our Algorithm 2. In addition, [32] requires use of the *clflush* instruction, which is forbidden in some environments, such as JavaScript.

```
rdtscp
movl %eax, %esi
movq (%rbx), %rax //L1 hit
movq (%rax), %rax //L1 hit
movq (%rax), %rax //L1 hit
movq (%rax), %rax //L1 hit
movq (%rax), %rax //L1 hit
movq (%rax), %rax //L1 hit
movq (%rax), %rax //L1 hit
movq (%rax), %rax //target address to measure
rdtscp
subl %esi, %eax
```

Fig. 3: Pointer chasing algorithm used to measure time.

4.3 PLRU vs. LRU Replacement Policy

In true LRU, the least recently used way is always chosen as the victim. However, in PLRU, fewer bits are used to record the access history in the cache, and it is not guaranteed that the least recently used way will be evicted. Thus, when the receiver triggers a replacement and measures the timing, there is uncertainty in the observed timing. This will cause an error when the receiver tries to infer the sender's access pattern. In [26], an in-house simulator is used to simulate the Tree-PLRU and Bit-PLRU replacement policies. The results show that with an initialization phase, the least recently used way will be evicted with high probability, especially after several rounds of accesses. In Section 5, we empirically show that the channel works on commercial processors using Tree-PLRU replacement policy.

4.4 Challenge: Distinguishing an L1 Hit from an L1 Miss

The major challenge for the receiver is to measure the memory access time precisely and to distinguish an L1 cache hit ($<$ 5 CPU cycles) from an L2 cache hit (10–20 CPU cycles). We use a pointer chasing algorithm and a dedicated data structure to measure one memory access precisely. In the pointer chasing algorithm in Figure 3, a linked list, where each element stores the address of the next element, is required. The register rbx points to the head of the linked list. Since the address of the mov instruction depends on the data fetched from the previous mov instruction, all eight accesses are serialized. Here, to avoid building a linked list in the sender's memory, we use a linked list of 7 elements in the receiver's own memory space, and let the 7^{th} element be the memory address to be measured. In this way, when measuring latency with the pointer chasing algorithm in Figure 3, it will first access 7 local elements and the target address at the end. Before running the measurement, the receiver can fetch the first 7 local elements to L1 cache, so the first 7 accesses will always hit in L1, and the total time measurement depends on whether the 8^{th} element is in L1 cache or not. With the method in Figure 3, we can distinguish if the 8^{th} element is in the L1 cache using the time observation on the machines we tested. The size of the linked list does not have to be 7. However, if the size is small, the noise due to *lfence* will affect the measurements. On the other hand, if the size is large, there will be noise in accessing the elements in the linked list.

Algorithm 3: Covert Channel Protocol

m: k-bit message to be sent on the channel
T_s: sender's sending period
T_r: receiver's sampling time
TSC: current time stamp counter, obtained by *rdtscp*

Sender's Code:

for $i = 0;\ i < k;\ i = i + 1$ **do**
  for *an amount time* T_s **do**
    Step 1: Encoding Phase, encoding m[k]
  end
end

Receiver's Code:

while *True* **do**
  Step 0: Initializion Phase
  while $TSC < T_{last} + T_r$ **do**
    nothing;
  end
  T_{last} =TSC
  Step 2: Decoding Phase
end

TABLE 1: Specifications of the tested CPU models.

Model	**Intel Xeon E5-2690**	**Intel Xeon E3-1245 v5**	**AMD EPYC 7571**
Microarchitecture	Sandy Bridge	Skylake	Zen
Number of cores	8	4	N/A[a]
L1D size	32KB	32KB	32KB
L1D associativity	8-way	8-way	8-way
Frequency	3.8GHz	3.9GHz	2.5GHz
OS	16.04.1 Ubuntu		

[a]We use the AMD processor on Amazon AWS EC2 platform. The CPU model is specific for Amazon AWS. One core was leased for our experiments.

5 Evaluation

To evaluate the transmission rate of the LRU channel, we evaluate it as a covert channel using one target set in the L1 data cache. As shown in Algorithm 3, the sender sends each bit of message m for T_s CPU cycles, by running the sender's operations (in Algorithm 1 or 2) for T_s in a loop for each bit in the message that the sender wants to send. We calculate the transmission rate by the total number of bits sent divided by the time (measured by the $time$ command in Linux). Thus, T_s decides the transmission rate. The receiver runs the receiver's operations (in Algorithm 1 or 2) every T_r CPU cycles in a loop and measures the latency using pointer chasing discussed in Section 4.4.

The evaluation is conducted on both Intel and AMD processors. The specifications of the tested CPU models are listed in Table 1. We evaluated both LRU Channels with shared memory and without shared memory presented in Section 4 under both hyper-threaded sharing and time-sliced sharing settings.

To evaluate the errors in the transmission channel, different metrics can be used. In a previous evaluation of LRU covert channels [26] and other covert channels [1], the *Edit Distance* (ED) (a.k.a., Levenshtein distance) is used, which considers bit flips, bit insertions, and bit deletions. However, in practice, most error correction codes (e.g., BCH

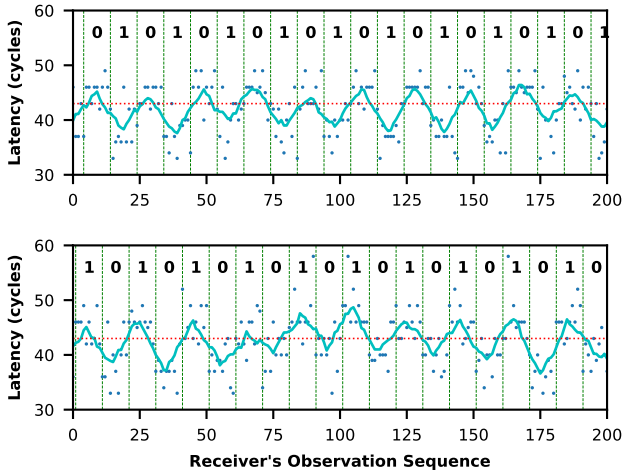

Fig. 4: Example sequences of the receiver's observation when the sender is sending 0 and 1 alternatively on Intel Xeon E5-2690 with a transmission rate of 480Kbps using (top) Algorithm 1 with T_r=600, T_s=6000, and d=8 and (bottom) Algorithm 2 with T_r=600, T_s=6000 and d=4. The blue dots show the latencies observed by the receiver, the red dot line shows the threshold of the L1 cache hit, and the light blue dot line shows the moving average of 10 measurements.

code, Goppa code, etc.) can only correct bit flips, but not bit insertions or bit deletions. Thus, the *Hamming Distance* (HD), which only considers bit flips in the received sequence, is the appropriate metric for practical error correction. In this paper, we evaluate the error rates in the channels using HD.

5.1 LRU Covert Channels in Intel Processors

5.1.1 LRU Channels in Hyper-Threaded Sharing

For the hyper-threading case, we tested the covert channel when the sender and the receiver are sharing the same physical core as two hyper-threads. Each of the sender and the receiver is a process (i.e., a separate program) in Linux.

LRU Channel with Shared Memory: In Algorithm 1, shared memory is needed between the sender and the receiver processes, e.g., achieved by a shared library. Figure 4 (top) shows the traces observed by the receiver when the sender is sending 0 and 1 alternatively. When the sender is sending bit 1, the access time of line 0 by the receiver is shorter, as is discussed in Section 4.1. The results on Intel Xeon E5-2690 are shown in Figure 4. The evaluation on E3-1245 v5 shows similar results, except that the two processors have different thresholds for L1 hit and miss latencies. This is due to different latencies for L1 and L2 cache access in the two processors. Also, the two processors are running at different frequencies, and thus, even with the same $T_s = 6000$, the transmission rate is 480Kbps for E5-2690 and 580Kbps for E3-1245 v5.

In the evaluation, the sender process sends a 128-bit random binary string repeatedly. We evaluate $T_r = \{600, 1000, 3000\}$ cycles, and $T_s = \{4500, 6000, 12000, 30000\}$ cycles. The receiver's operations of Algorithm 1 in total takes about 560 cycles, including logging of the results, and thus, T_r>560. Because the CPUs have 8-way set-associative caches and the maximum possible d is 8, we test parameters $d = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

TABLE 2: Bit flip direction.

	1 to 0	0 to 1
Algorithm 1	46.8%	53.2%
Algorithm 2	63.0%	37.0%

Also, the 128-bit string is sent at least 30 times to obtain average errors.

Figure 5 (top) shows the error rates of the channel versus the different transmission rates (i.e., different values of T_s). Error rates are evaluated by both Edit Distance (ED) and Hamming Distance (HD). When calculating the ED, the received signal is cropped into 128-bit strings, and compared with the original string with an offset that minimizes the distance. When calculating the HD, the received sequence is cropped into 1024-bit strings. We use a 32-bit sliding window to identify the start of chunks. The first 32 bits in the string to be transmitted is a fixed synchronization string to help the receiver align the start of the sequence. Then, the HD between the 1024-bit received string and the original string is calculated. We also evaluate different chunk sizes and synchronization string lengths (i.e., sliding window sizes). Smaller chunk sizes give better error rate results, but the overhead due to synchronization string becomes larger. Smaller synchronization string sizes might result in larger errors due to possible failures to align the strings. As shown in Figure 5 (top), the error rates evaluated by HD of 1024-bit chunk and 32-bit synchronization string only give slightly larger error rates than that evaluated with ED[2]. This indicates that there are not many bit insertions or bit deletions in the received bit string, especially for the optimal $T_r = 1000$. If the sender uses a 32-bit synchronization string in every 1024 bits (3% overhead), the receiver is able to decode the signal practically. With an error rate of less than 5% for $T_r = 1000$, error correction codes can be applied. In addition, Table 2 shows the type of error in Algorithm 1. There are both 1-to-0 flips and 0-to-1 flips, and the two types of errors occur with similar frequency. In Algorithm 1 both types of errors might occur because PLRU does not always choose to evict the least recently used cache line in the set.

As shown in Figure 5 (top), for Algorithm 1, d does not affect the error rates much on the E5-2690. This is because, in hyper-threaded sharing, the sender process and the receiver process execute in parallel. The sender operation can happen when the receiver is executing any part of his or her operation, and d only makes the sender operation more likely to happen in the sleep part of the receiver's operation. $T_r = 1000$ gives slightly better error rates than $T_r = 600$. This might be because more interleaving between the two threads due to greater T_r and the receiver can observe more sender's activity in one measurement. As T_r increases to 3000 cycles, the error rates increase. In general, the error rates increase as the transmission rate increases (i.e., T_s decreases). This is because a greater T_s or a smaller T_r will result in more measurements for each of the bit transmitted, and the noise can be canceled out by taking the average of the measurements.

2. For the same string pair, ED should be always smaller than HD. However, when we process HD and ED, the received sequences are chunked slightly differently, and thus, for some cases HD becomes smaller than ED.

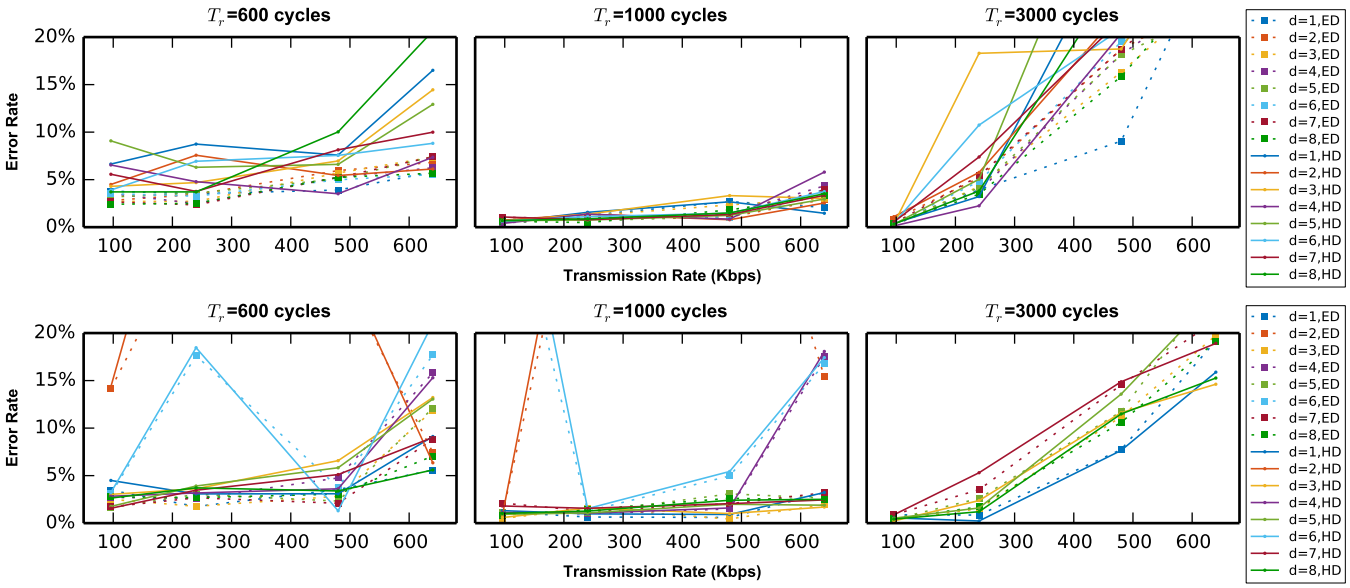

Fig. 5: Transmission error rate evaluated using *Edit Distance* (in dotted line) and *Hamming Distance* (in solid line) as a function of the transmission rate (different T_s) for different T_r on Intel Xeon E5-2690 using (top) Algorithm 1 and (bottom) Algorithm 2. d is a parameter of the receiver in Algorithm 1 and Algorithm 2.

LRU Channel without Shared Memory: In Algorithm 2, shared memory between the sender and the receiver is not required. Figure 4 (bottom) shows the traces observed by the receiver. When the sender is sending bit 1, the access time of line 0 by the receiver is longer, due to the sender's access to the same set.

For Algorithm 2, we also evaluate the same set of values of T_r, T_s, and d. Figure 5 (bottom) shows the error rates versus the different transmission rates (different values of T_s) on E5-2690. The error rates evaluated by HD are similar to that of ED, showing there are not many bit insertions or bit deletions. Compared to the LRU channel with shared memory, the LRU channel without shared memory has more noise. In Tree-PLRU, when the sender accesses the set, the receiver may not observe a miss in the end, resulting in a false 0. Also, any access to the same set (by the other part of the program or other processes on the core) may result in a false 1. However, these errors usually occur consecutively in time. So the receiver can detect the noise if observing a long sequence of all 1 or all 0. We exclude those traces to obtain Figure 5.

When $d = \{2, 4, 6\}$, the error rates are large on E5-2690, especially for large T_r. This is because even d makes the Tree-PLRU point to another side of the sub-tree, and the receiver will not evict line 0 during decoding. Table 2 shows the frequency of the two types of errors. 1-to-0 flips are more frequent in Algorithm 2. This reaffirms the observation that the receiver may not evict line 0 in some cases.

5.1.2 LRU Channels in Time-Sliced Sharing

When the sender and receiver are sharing the same core in a time-sliced sharing setting, the two processes still share the same L1 cache. To evaluate the covert channel in a time-sliced sharing setting, we programmed the sender process to always send 1 or 0, and the receiver to measure the time of accessing line 0 every T_r. Figure 6 shows the percentage of

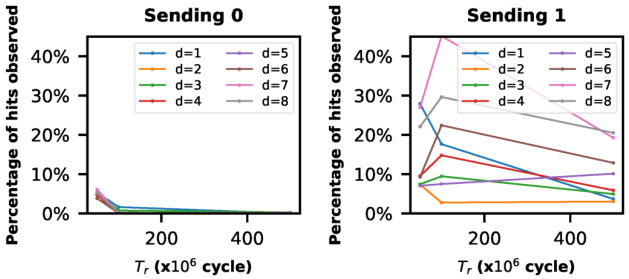

Fig. 6: Percentage of cache hits observed by the receiver on Intel Xeon E5-2690, when the sender is sending (left) 0 and (right) 1 using Algorithm 1 under time-sliced sharing.

cache hits received for different d and T_r when the sender is sending 0 or 1 using Algorithm 1 on both CPUs tested. Each data point comes from 1000 measurements.

As shown in Figure 6, with proper parameters, the receiver can distinguish between the sender sending 0 and 1. For example, if $d = 8$ and $T_r = 10^8$ cycles, the receiver will observe almost 100% of L1 cache misses when the sender is sending 0, and the receiver will observe about 30% of L1 cache hits when the sender is sending 1 on both Intel processors. The receiver does not observe hits with a higher probability, because in time-sliced sharing, each process uses the core for a certain period of time. When the receiver monitors the sender in a loop, multiple loop iterations will run within a time-slice period, and only the first iteration will reflect the sender's behavior, the other iterations in the time period run without interleaving with the sender. Nevertheless, the receiver can still recognize the message the sender is sending by the percentage of cache hits received. Assuming that 10 measurements are needed when $T_r = 10^8$ to differentiate 30% from <5%, the transmission rate is about 2.4bps.

Compared to hyper-threaded sharing, much larger T_r is needed here to have interaction between the two threads

(about 10^8 cycles for both processors tested). However, if T_r is too large, the distinguishability decreases, as other processes might be scheduled during T_r. As shown in Figure 6, $d = 8$ and $d = 7$ gives the best distinguishability between the sender sending 0 and 1. This is because T_r is large, and the time for the receiver's operations becomes small compared to the sleep time. Thus, a context switch is more likely to happen during the sleep time. In Algorithm 1, a greater d leads to fewer accesses to the target set after the sleep, and thus, line 0 is less likely to be evicted during decoding. Such evicted line 0 may result in a false 0.

We also tried Algorithm 2 but failed to observe any signal from the measurement. We think the reason is that T_r should be large to allow interference between the sender and the receiver, however, any other processes running during T_r could pollute the target set and introduce a lot of noise.

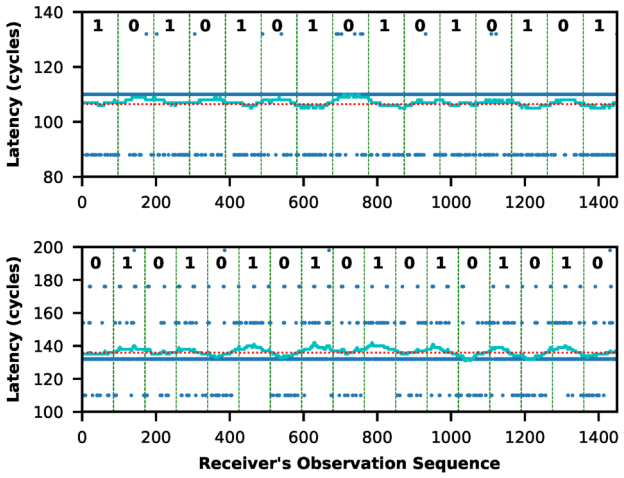

Fig. 7: Example sequences of receiver's observation when the sender is sending 0 and 1 alternatively using (top) Algorithm 1 and (bottom) Algorithm 2 on AMD EPYC 7571. For Algorithm 1, $T_r = 1000$, $T_s = 10^5$, $d = 8$, and the transmission rate is 22Kbps. For Algorithm 2, $T_r = 1000$, $T_s = 10^5$, $d = 4$, and the transmission rate is 25Kbps.

5.2 LRU Covert Channels in AMD Processors

In this section, we evaluate the channel on AMD processors, which have a different L1 cache design and time stamp counter from Intel processors.

5.2.1 *utag in AMD L1 Cache*

To save power, AMD L1 cache has a special linear address *utag* and *way-predictor* (see 2.6.2.2 in [33]). The *utag* is a hash of the linear address. For a load, while the physical address is looked up in TLB, the L1 cache uses the hash of the linear address (i.e., virtual address) to match the *utag* and determines which cache way to use in the cache set. When the physical address is available after address translation, only that cache way will be looked up instead of all 8 ways. So, when the physical address of a load matches a cache line in the cache, if the *utag* of that way is of a different linear address unless the hash of two linear addresses conflicts, a latency of an L1 miss will be observed, even though the physical address matches and data is in L1.

This limits the use of our Algorithm 1 across processes using different address spaces. If the sender process accesses line 0, the *utag* of line 0 will be updated with the linear address of line 0 in the sender's address space. When the receiver accesses line 0 and measures the time, unless the hash of the linear address of line 0 in the sender's process and in the receiver's process conflicts, the receiver will always observe an L1 cache miss latency no matter if the line 0 is in L1 or not. However, the hash of *utag* is not designed for security and can be reverse-engineered [34]. Furthermore, as long as the sender and the receiver are in the same address space, the LRU channel using Algorithm 1 still exists. For example, it can be used to transfer information in the case of escaping the sandbox in JavaScript [10].

5.2.2 *Evaluation of LRU Channels in AMD Processors*

We evaluate the characteristics of the LRU covert channel on an AMD EPYC 7571 processor on Amazon AWS EC2 platform. Figure 7 (top) shows the trace observed by the receiver, when the receiver and the sender are two threads in the same address space (using *pthread*s in C) running in a hyper-threaded sharing using Algorithm 1. Due to the coarse granularity of the readout value of the time stamp counter in AMD, it is hard to identify the signal from the raw measurements (blue dots). Thus, we evaluate the moving average. The light blue dot line in Figure 7 shows the moving average of the latency of 97 measurements, where the 97 is the best fit period of sending one bit for this trace[3]. When the sender is sending 0 and 1 alternatively, the moving average is a wave-like pattern. The receiver can decode the message using a threshold for the moving average, red dotted line in Figure 7. By measuring the total time taken by the receiver to gather the trace and the period of each bit received, the effective transmission rate is 22Kbps. Due to the coarser-granularity of the AMD time stamp counter and lower frequency, the transmission rate of the channel is about one order of magnitude lower than that in Intel processors.

We also tested Algorithm 2 under hyper-threaded sharing on AMD EPYC 7571. Figure 7 (bottom) shows a trace observed by the receiver. The receiver and the sender are two programs (in different memory space). Similarly, the light blue dot line shows the moving average of the latency of 85 measurements, where the 85 is the best fit, resulting in an effective transmission rate of 25Kbps. When the sender is sending 0 and 1 alternatively, the moving average is a wave-like pattern. The measured latency in Figure 7 (top) and (bottom) are quite different. This might due to the processor running at a different frequency for power saving at the time of measurement.

5.3 Comparing the Evaluated LRU Channels

Table 3 compares the transmission rate per cache set of the channels tested under different configurations. Hyper-threading gives a much higher transmission rate than

3. The fact that the period does not equal to T_s/T_r indicates that threads do not get scheduled evenly. This might be due to the Amazon EC2 platform, as we observe similar phenomenon on Intel processors on EC2.

TABLE 3: Transmission rate of the evaluated LRU channels.

		Intel	AMD
Hyper-Threaded	Algorithm 1	∼500Kbps	∼20Kbps
	Algorithm 2	∼500Kbps	∼20Kbps
Time-Sliced	Algorithm 1	∼2bps	∼0.2bps
	Algorithm 2	–	–

TABLE 4: Cache miss rate of Spectre V1 attack.

		F+R (mem)	F+R (L1)	L1 LRU Alg.1	L1 LRU Alg.2
Intel Xeon E5-2690	L1D	2.75%	4.73%	4.19%	4.75%
	L2	7.58%	0.07%	0.11%	0.09%
	LLC	98.15%	0.87%	0.72%	0.87%
Intel Xeon E3-1245 v5	L1D	2.86%	4.84%	4.13%	4.86%
	L2	7.39%	0.49%	0.71%	0.45%
	LLC	91.17%	1.83%	0.74%	0.96%

time-sliced sharing because of more interference between the sender and the receiver. Under hyper-threading, Algorithm 1 and Algorithm 2 have a similar transmission rate. The transmission rate is comparable to other timing channels in caches [1], [3]. However, recall that Algorithm 2 is easily affected by noise due to activities of other programs, but the noise is easy to filter, because the noise activity is usually of a different frequency. The LRU channel on AMD processors is about one order of magnitude slower than on Intel processors, due to the coarser-granularity of readout value of timestamp counter and lower clock frequency.

6 Evaluation of LRU Channels in Transient Execution Attacks

Transient execution attacks, e.g., Spectre, leverage transient execution to access a secret and a covert channel to pass the secret to the attacker [10], [11], [12]. Currently, most proof-of-concept codes of transient execution attacks use the cache Flush+Reload covert channel. Here we demonstrate that our LRU covert channel also works in a Spectre attack to retrieve the secret.

Note that here the secret contains more than 1 bit, and multiple cache sets are used to encode the secret. In practice, 63 cache sets are used (both Intel and AMD processors tested have 64 sets, the remaining one set is for the 7 elements in the pointer chasing algorithm as discussed in Section 4.4).

The Flush+Reload covert channel needs one memory access depending on the secret as the sender's operation. Meanwhile, as shown in both algorithms in Section 4, the sender's operation in the LRU channels also only needs one memory access whose target set depends on the secret. Thus, the victim code using the LRU channel can be identical to the disclosure gadget in the Flush+Reload channel. When demonstrating transient execution attack using the LRU channels, we take the Spectre variant 1 attack sample code [10] and keep the victim (sender) code the same, and change the attacker (receiver) code to use the L1 LRU channels as the disclosure primitive instead. We are able to launch the Spectre attack using the LRU channels (both Algorithm 1 and 2) to observe the secret. Table 4 shows the cache miss rate (including both the victim and the attacker) during a Spectre attack.

Comparing to the Flush+Reload channel, the advantage of the LRU disclosure primitive is the short encoding time (i.e., the sender's operations), and thus, a smaller speculative window is required, which may make the attack more dangerous and harder to defend.

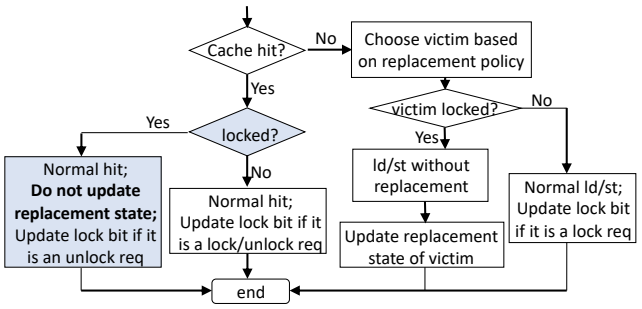

Fig. 8: PL cache replacement logic flow-chart. White boxes show the original PL cache design in [17]. Blue boxes show the new PL logic added in our simulation to defend the LRU attack.

7 LRU Covert Channels in Secure Caches

Many secure cache designs have been proposed to defend against conventional and transient execution attacks. Partitioning or randomization are the two main leveraged techniques. In this section, we first show how these two types of secure caches might fail to defend side and covert channel in cache replacement states using simulation, and provide a security analysis of different secure caches.

7.1 Partitioning

Some secure cache designs prevent certain cache line sharing between the victim and the attacker using partitioning, e.g., [16], [17], [18], [19], [35]. The goal of cache partitioning is to prevent the cache state that is changed by one party to be observable to another party. Partitioned caches require information from software about which cache line need to be isolated from others. The partitioning can be either static or dynamic. Current proposals focus mainly on the data and tag of a cache line. However, the cache replacement state might still be shared between different users in such secure caches, leaving the possibility of a covert channel as we show in this work.

Partition-Locked (PL) cache [17] is an example of a secure cache leveraging partitioning with small performance and area overhead. Each cache line is extended with one lock bit. When a cache line is locked, the line will not be evicted by any cache replacement until unlocking to protect the line, as shown in Figure 8. If a locked line is chosen as victim to be replaced, the replacement will not happen, and the incoming line will be sent to the pipeline without being cached. In this way, when a line is locked, the cache line state will not be changed until unlocking. PL cache is shown to be effective against Flush+Reload, Prime+Probe, etc.

But the LRU state will still be updated on accesses to the locked cache line, and the update will affect the LRU states of other lines. Figure 9 shows an example of the LRU covert channel attack in PL cache using Algorithm 2, where $N = 8$ and $d = 4$. First, the sender locks line 8, as required by PL cache to enforce protection. Then, the sender and the receiver execute Algorithm 2. If the sender accesses line 8,

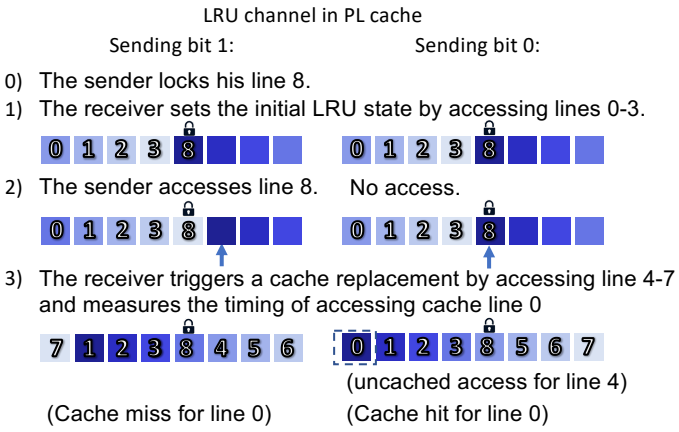

Fig. 9: Cache states in PL cache in LRU covert channel attacks.

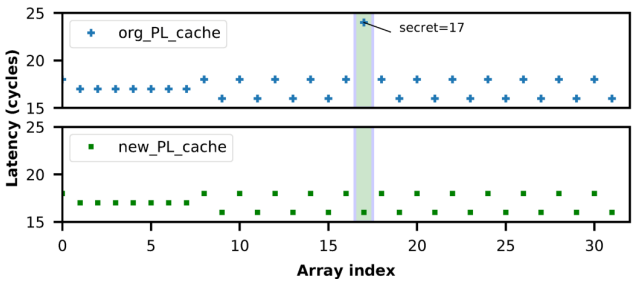

Fig. 10: Simulation result of the LRU attack with Algorithm 2 in gem5 with (top) original PL cache design and (bottom) new PL cache design which locks the LRU state to defend the LRU attack.

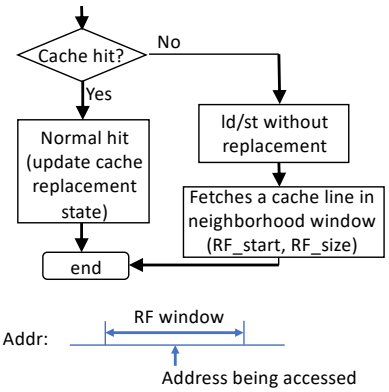

Fig. 11: RF cache replacement logic flow-chart and RF window.

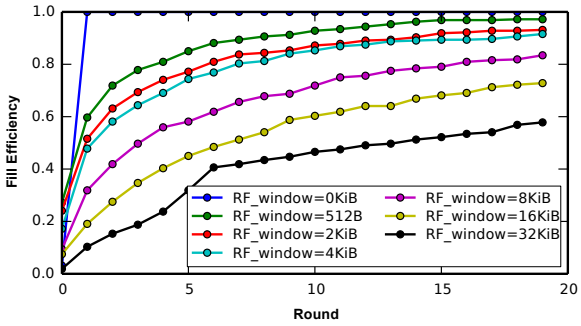

Fig. 12: Probability of a cache line being filled in RF cache in gem5 simulator.

older cache lines will be evicted and a cache miss will be observed by the receiver. Otherwise, line 8 will be chosen as the line to be evicted. However, line 8 is locked, so the new cache line will be accessed uncached. In this example, even though the sender's cache line is locked, sender's access will still change the LRU state, which gives the receiver the ability to infer the sender's access pattern by observing memory access time.

We implement the PL cache using PLRU replacement algorithm in the gem5 simulator, and test the LRU attack illustrated in Figure 9. As shown in Figure 10 (top), with the original design, the receiver can still receive the secret by observing the time of accessing line 0. In this demonstration, the receiver is monitoring 32 cache sets and observes a cache miss in the cache set the sender accesses. Thus, the receiver learns the access pattern of the sender, e.g., set 17 here.

To mitigate the LRU channel, the LRU state should be locked when the cache line is locked. The changes to logic of PL cache are shown in the blue boxes in Figure 8. In this way, the receiver will always observe a cache hit, and thus not learn any information, as confirmed in Figure 10 (bottom).

7.2 Randomization

Randomization is another method to mitigate cache covert and side channels. In a random fill (RF) cache [22], the cache line to be fetched into cache is chosen randomly to decouple the cache state and the access pattern. As shown in Figure 11, in RF cache, a cache hit is identical to that in a normal cache. A cache miss will cause the data to be sent to the pipeline directly without being cached, and then, a cache line in the neighborhood window will be fetched into cache instead of the cache line being accessed. The neighborhood RF window is defined by $[addr-\texttt{RF\_start}, addr-\texttt{RF\_start}+\texttt{RF\_size}]$. The RF window can be reconfigured by changing the value of registers RF_start and RF_size. The idea behind RF window is that a data in the neighborhood window might be accessed in the future due to locality, so the performance benefits from future cache hits.

However, in the RF cache, the randomization is in the data fetch. Once a cache line is in the cache, the RF cache works as a normal set-associative cache. If there is a cache hit by the sender, the cache replacement state will be updated. If there is a cache replacement, the cache replacement state will be used to chose a cache line to evict. Hence, if the receiver can initialize the cache set in a controlled manner by prefilling the cache with desired cache lines, the LRU side or covert channel is still possible in the RF cache.

We implement the RF cache logic in the L1 cache in the gem5 simulator. To test the probability of a cache line being fetched into cache, we access 32 consecutive cache lines in each round, and measure the time to see if the cache line is in the cache. We test different RF sizes, and always center the RF window to the cache line being accessed, i.e., $\texttt{RF\_start} = \frac{1}{2}\texttt{RF\_size}$. Figure 12 shows the probability of a cache line being filled into the RF cache, dubbed "fill efficiency", after n rounds of accesses. The results show that more rounds of accesses give higher fill efficiency, because

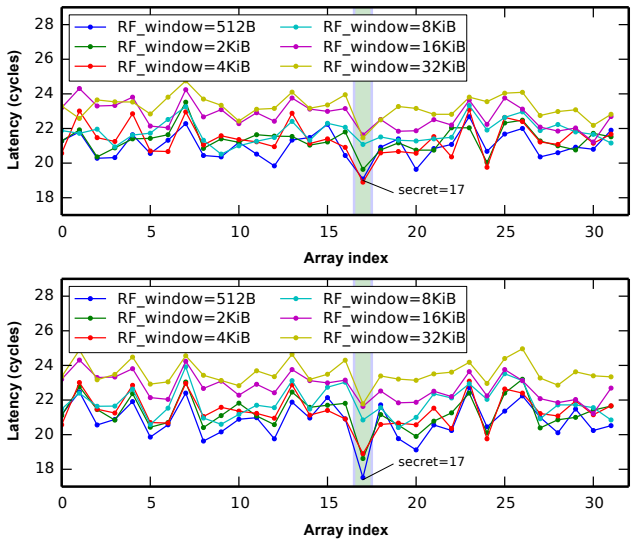

Fig. 13: Simulation result of the LRU attack in RF cache with Algorithm 1 in `gem5` simulator with (top) 5 rounds of cache fill and (bottom) 10 rounds.

more rounds of accesses cause more cache misses and trigger more RF cache fetching. For a normal set-associative cache (`RF_size`=0), the fill efficiency will reach 100% in round 1. A small RF window result in high fill efficiency, because the desired cache lines will be fetched with higher probability in each random fill. High fill efficiency indicates better performance when the workload's memory access has locality. However, higher fill efficiency also means that the attacker can fill the cache in a controlled manner.

Because the LRU channel does not require the sender to cause cache replacement to trigger cache state updates, the RF cache is still vulnerable. We demonstrate the LRU covert channel in the RF cache. The operation is the same as introduced in Section 4, but the initialization step now requires several rounds to prefill the desired cache lines and the last step requires to access a cache line several times to trigger cache replacement in the target cache set. Figure 13 shows the result of the attack using Algorithm 1. The results are the average of observed latency generated from 100 different random seeds. The receiver can infer the sender's access pattern of the sender, e.g., accessing set 17 here, especially when the `RF_size` is less than 4KiB. The top figure shows the result when the receiver fills the cache with 5 rounds of accesses, and the bottom shows 10 rounds of accesses. More rounds of cache fill and smaller RF window size give more difference in the time observation between the sender sending 0 and 1, which is consistent with the fill efficiency result in Figure 12. Also, a small RF window in general gives smaller access timing, as there could be more cache hits of line 0.

The noise in RF cache LRU channel comes from two sources. Firstly, the receiver's multiple rounds of access does not guarantee that the desired cache line to be fetched into cache. For example, if line 0 is not fetched in the first step, it is likely to always get a cache miss regardless of the sender's access. Secondly, random fills of other accesses will cause unexpected cache lines to be fetched or cache replacement states to be updated. For example, in the decoding phases, the receiver accesses a new cache line multiple times, which might fetch the line 0 or update the line 0 as the most recently used line, causing cache hit in the observation. Our experimental results in Figure 13 show that `RF_size` of at least 8KiB (128 cache lines) is required to inject enough noise to the channel. However, when `RF_size` is 32 cache lines, performance in terms of instructions per cycle (IPC) can be decreased as much as 30% for certain workloads, as reported by [22]. This means the RF cache will introduce significant performance overhead to mitigate the LRU channel.

On the other hand, compared to a normal set-associative cache, RF fill cache requires the sender to have multiple rounds of accesses to set the initial cache replacement state. This makes the LRU channel without shared memory (e.g., Algorithm 2) less practical, because either the receiver needs to have access to the cache line that is accessed by the sender (i.e., shared memory space) for initialization or the sender needs to fetch the cache line in the initialization, resulting in more effort on the sender's side.

7.3 Secure Cache Designs and Cache Replacement Covert Channels

In addition to the quantitative analysis in Section 7.1 and Section 7.2, we qualitatively analyze all remaining known secure caches. We summarize a number of secure caches and whether LRU side and covert channels exist in Table 5.

We consider external attacks where the sender and the receiver reside in different security domains. There are also internal attacks where all the operations are done by the sender and the receiver only observes the timing of the sender program. However, internal attacks are not mitigated by many of the secure caches even for existing cache channels, and thus, it is hard to compare the internal attacks leveraging cache replacement state on the secure caches.

Moreover, the secure cache designs focus on different levels of caches. While in this paper, we demonstrate the LRU channel only in L1 cache, there is evidence that side and covert channel can be built in other levels of caches [32]. In Table 5, we analyze whether a channel leveraging cache replacement state could be built in each secure cache design.

There are two techniques used in the secure cache designs: partitioning and randomization. Partitioning could be between both data look up and eviction, or only eviction. Also, partitioning can be static or dynamic, as shown in Table 5. The lesson we learn from PL cache is that when we partition a cache, the cache replacement state (and other cache states) should also be partitioned. Otherwise, not all covert channels or side channels are closed in the partition. However, it is not trivial to partition the cache replacement state, especially when the partitioning is dynamic. Thus, DAWG [21] proposed a method partition the PLRU states dynamically. Meanwhile, in some of the partitioned cache designs, only the eviction is isolated between security domains, i.e., program in different secure domain can access each other's cache partition, but cannot trigger eviction of cache line of another partition. However, eviction is not the root cause of the attacks leveraging cache replacement states, and thus, the attacks can not be mitigated.

Randomization can be applied to the three basic operations in cache: lookup (e.g., which cache entries a cache line

TABLE 5: LRU channels in existing secure cache designs.

	Features	Example Designs	Secure against LRU channels?
Partition between security domains	Statically partition all access	SecVerilog cache [19], CATalyst [36], Sanctum cache (Yes) [35], [37], InvisiSpec (Yes) [25]	Only if the replacement state is partitioned.[1]
	Dynamically partition all access	PL cache [17], NoMo cache [18], DAWG (Yes) [21]	
	Partition only evictions (Dynamically)	SHARP cache [20], Relaxed Inclusion Caches [38]	No
Randomization	Randomize address to cache entry mapping	New cache [23], RP cache [17], Time-Secure Caches [39], CEASER cache [40], Skewed-CEASER [41], ScatterCache [42]	Yes
	Randomize cache fetch	RF cache [22],	No
	Randomize cache replacement/invalidation	Non Deterministic cache [24], CleanupSpec [43]	Yes

[1] "Yes" next to the specific cache design means the replacement state is partitioned properly.

can map to), fetch (e.g., when and which cache line to be fetched into cache), and invalidation/eviction (e.g., when and which cache line to be invalidated/evicted from the cache). The mapping between the line addresses and the cache entries or sets can be randomized (Row 4 in Table 5). Some of the designs (e.g., scattercache [42]) even completely remove the concept of a cache set and a random replacement policy is used. The LRU side and covert channel assume the sender and the receiver to share the same cache set. Because the receiver (and the sender) cannot map the addresses to the target cache set to build a channel, these caches are safe against cache replacement side and covert channels. As for the idea of randomizing the cache fetch, RF cache is the only proposal so far. We show in Section 7.2 that RF is not effective in defending the LRU channel. Randomization in cache replacement and eviction is also proposed. In this case there is not replacement state, and thus, no side channel.

In addition to the conventional covert channels, security improvements and designs have been recently proposed to defend transient execution attacks, such as Spectre and Meltdown [10], [11], [12]. Some defenses, e.g., [44], [45], prevent memory accesses in case of unsafe transient execution, and thus, prevent our channels from being leveraged as part of transient execution attacks. Some other defenses mitigate the covert channel in cache replacement states using partitioning (e.g., DAWG [21], InvisiSpec [25]) and use a random replacement policy (e.g., CleanupSpec [43]). However, many other designs do not consider covert channels in the replacement state and our covert channels remain unprotected.

8 Conclusion

We presented novel timing-based channels leveraging the cache LRU replacement states. We designed two protocols to transfer information between processes using the LRU states for both cases when there is shared memory between the sender and the receiver and when there is no shared memory, and demonstrated the LRU channels on real-world commercial processors. The LRU channels require access (cache hit or miss) from the sender, while all existing state-based timing-based cache side and covert channels always need the sender to trigger a cache replacement (a cache miss). Thus, the LRU channel has shorter encoding time, lower cache miss rate for the sender, and requires a smaller speculation window in transient attack scenarios. We conducted a detailed evaluation on the proposed covert channels using both ED and HD. We also showed the new LRU channels also affect the current secure cache designs, such as PL cache and RF cache.

Acknowledgements

We would like to acknowledge Amazon for providing AWS Cloud Credits for Research. This work was supported by NSF grants 1651945 and 1813797, and through SRC award number 2844.001.

References

[1] J. Szefer, "Survey of microarchitectural side and covert channels, attacks, and defenses," *Journal of Hardware and Systems Security*, vol. 3, no. 3, pp. 219–234, 2019.

[2] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack," in *USENIX Security Symposium (USENIX)*, 2014, pp. 719–732.

[3] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are Coherence Protocol States Vulnerable to Information Leakage?" in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 168–179.

[4] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Annual Computer Security Applications Conference (ACSAC)*, 2006, pp. 473–482.

[5] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: a timing attack on OpenSSL constant-time RSA," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[6] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 538–570, 2019.

[7] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *Symposium on Security and Privacy (S&P)*, 2019, pp. 870–887.

[8] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[9] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers' Track at the RSA Conference*, 2006.

[10] 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 *Symposium on Security and Privacy (S&P)*, 2019, pp. 1–19.

[11] 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 *USENIX Security Symposium (USENIX)*, 2018, pp. 973–990.

[12] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security Symposium (USENIX)*, 2019, pp. 249–266.

[13] K. So and R. N. Rechtschaffen, "Cache operations by MRU change," *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 700–709, 1988.

[14] A. Malamy, R. N. Patel, and N. M. Hayes, "Methods and apparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature," 1994, US Patent 5,353,425.

[15] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2016, pp. 118–140.

[16] R. B. Lee, P. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, 2005, pp. 2–13.

[17] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, 2007, pp. 494–505.

[18] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *Transactions on Architecture and Code Optimization*, vol. 8, no. 4, 2012.

[19] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 99–110, 2012.

[20] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks," in *Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 347–360.

[21] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *International Symposium on Microarchitecture (MICRO)*, 2018, pp. 974–987.

[22] F. Liu and R. B. Lee, "Random fill cache architecture," in *International Symposium on Microarchitecture (MICRO)*, 2014, pp. 203–215.

[23] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[24] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras, "Non deterministic caches: A simple and effective defense against side channel attacks," *Design Automation for Embedded Systems*, vol. 12, no. 3, pp. 221–230, 2008.

[25] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *International Symposium on Microarchitecture (MICRO)*, 2018, pp. 428–441.

[26] W. Xiong and J. Szefer, "Leaking information through cache LRU states," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 139–152.

[27] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 60–71.

[28] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 381–391, 2007.

[29] 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 *Conference on Computer and Communications Security (CCS)*, 2009, pp. 199–212.

[30] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *Conference on Computer and Communications Security (CCS)*, 2014, pp. 990–1003.

[31] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Conference on Computer and Communications Security (CCS)*, 2018, pp. 2109–2122.

[32] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks," in *USENIX Security Symposium*, 2020, pp. 1967–1984.

[33] *Software Optimization Guide for AMD Family 17h Processors*, https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf, accessed Feb. 2019.

[34] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of amd's cache way predictors," in *15th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2020.

[35] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *USENIX Security Symposium (USENIX)*, 2016, pp. 857–874.

[36] 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 *International symposium on high performance computer architecture (HPCA)*, 2016, pp. 406–418.

[37] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *International Symposium on Microarchitecture (MICRO)*, 2019, pp. 42–56.

[38] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "RIC: Relaxed inclusion caches for mitigating LLC side-channel attacks," in *54th Design Automation Conference (DAC)*, 2017, pp. 1–6.

[39] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems," in *55th Annual Design Automation Conference (DAC)*, 2018, pp. 1–6.

[40] M. K. Qureshi, "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping," in *International Symposium on Microarchitecture (MICRO)*, 2018, pp. 775–787.

[41] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 360–371.

[42] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium*, 2019, pp. 675–692.

[43] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An undo approach to safe speculation," in *International Symposium on Microarchitecture (MICRO)*, 2019, pp. 73–86.

[44] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *International Symposium on Microarchitecture (MICRO)*, 2019, pp. 572–586.

[45] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data," in *International Symposium on Microarchitecture (MICRO)*, 2019, pp. 954–968.

Wenjie Xiong (S'17) received her Ph.D. degree from the department of Electrical Engineering at Yale University in 2020 and her B.Sc. in Microelectronics and Psychology from Peking University in 2014. She is currently a postdoctoral researcher at Facebook. Her research interests comprise Physically Unclonable Functions and side-channel attacks and defenses.

Stefan Katzenbeisser (S'98–A'01–M'07–SM'12) received the Ph.D. degree from the Vienna University of Technology, Austria. After working as a Research Scientist with the Technical University of Munich, Germany, he joined Philips Research as a Senior Scientist in 2006. After holding a professorship for Security Engineering at the Technical University of Darmstadt, he joined University of Passau in 2019, heading the Chair of Computer Engineering. His current research interests include embedded security, data privacy and cryptographic protocol design.

Jakub Szefer (S'08–M'13–SM'19) received B.S. with highest honors in Electrical and Computer Engineering from University of Illinois at Urbana-Champaign, and M.A. and Ph.D. degrees in Electrical Engineering from Princeton University where he researched secure hardware architectures. He is currently an Associate Professor in the Electrical Engineering department at Yale University, where he leads the Computer Architecture and Security Laboratory (CASLAB). His research interests are at the intersection of computer architecture, hardware security, and FPGA security.