Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks

Yanan Guo[1,3], Andrew Zigerelli, Youtao Zhang[2], and Jun Yang[1]

[1]Electrical and Computer Engineering Department, University of Pittsburgh
[2]Department of Computer Science, University of Pittsburgh
[3]yag45@pitt.edu

***Abstract*—Modern x86 processors have many prefetch instructions that can be used by programmers to boost performance. However, these instructions may also cause security problems. In particular, we found that on Intel processors, there are two security flaws in the implementation of `PREFETCHW`, an instruction for accelerating future writes. First, this instruction can execute on data with read-only permission. Second, the execution time of this instruction leaks the current coherence state of the target data.**

Based on these two design issues, we build two cross-core private cache attacks that work with both inclusive and non-inclusive LLCs, named Prefetch+Reload and Prefetch+Prefetch. We demonstrate the significance of our attacks in different scenarios. First, in the covert channel case, Prefetch+Reload and Prefetch+Prefetch achieve 782 KB/s and 822 KB/s channel capacities, when using only one shared cache line between the sender and receiver, the largest-to-date single-line capacities for CPU cache covert channels. Further, in the side channel case, our attacks can monitor the access pattern of the victim on the same processor, with almost zero error rate. We show that they can be used to leak private information of real-world applications such as cryptographic keys. Finally, our attacks can be used in transient execution attacks in order to leak more secrets within the transient window than prior work. From the experimental results, our attacks allow leaking about 2 times as many secret bytes, compared to Flush+Reload, which is widely used in transient execution attacks.

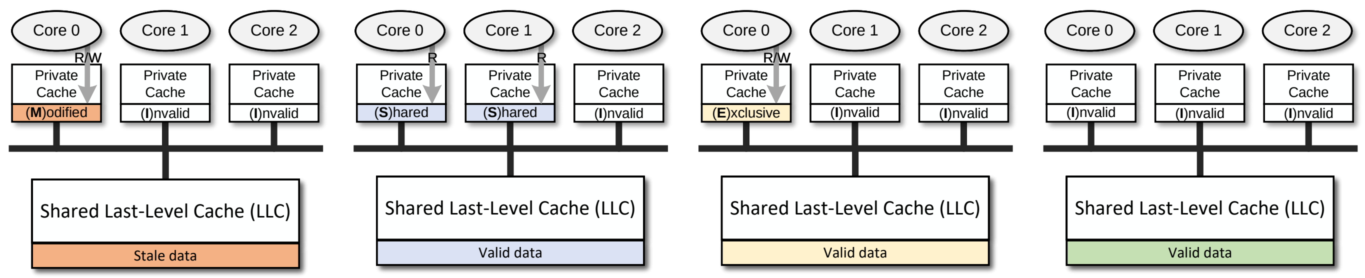
I. Introduction

Modern processors often feature many microarchitectural structures that are shared among applications. Although such resource sharing enables significant performance benefits, it also gives adversaries the potential to build powerful covert channel and side channel attacks. When an application runs on such hardware, its execution may cause various state changes to these shared microarchitectural structures, which can be observed by an attacker on the same platform. Through repeated observations, the attacker can derive the application's private information related to the state changes, bypassing sandboxes and traditional privilege boundaries. Cache timing covert channel and side channel attacks, or cache attacks for short, are extremely potent [1]–[25]. They are especially powerful primitives used in the more recently discovered transient execution attacks [26]–[36]. Different cache behaviors, such as hits and misses create significant timing differences to the execution of an instruction. Attackers can use these timing variances to stealthily transfer data (in the covert channel case) or infer some secrets from a victim (in the side channel case) such as cryptographic keys.

Most cache attacks can be classified into private cache attacks and last level cache (LLC) attacks. Private cache attacks (e.g., [5], [11]) are usually *same-core* attacks; such attacks typically require the victim and attacker to be located on the same physical core so that they share the same private cache. The attacker learns the victim's cache accesses by monitoring the state of the victim's data in the private cache. For example, the attacker evicts the victim's data from one level of private cache to a lower level of private cache or the LLC and waits for the victim: if the victim accesses the data and brings it back to the original cache level, the attacker can observe this through timing information (e.g., if this data is shared with the attacker, the attacker can access it and determine whether it is already present in the original cache level by measuring the access latency). Private cache attacks have the potential to be very fast, since all the data accesses are on-chip cache accesses. However, they are also limited because such attacks usually require simultaneous multithreading (SMT) to be enabled, which is not always the case, especially for security-conscious cloud providers [37]–[39]. In contrast, LLC attacks can be *cross-core* attacks, i.e., the attacker can run on a different physical core than the victim, since the LLC is usually shared among cores. In most LLC attacks, the attacker evicts the victim's data from the LLC to memory to monitor the victim's access on it [1], [2], [4]. Although LLC attacks are more practical, the attack bandwidth is limited, since memory accesses are relatively slow.

These limitations, in both the private cache and LLC attacks, may be overcome by building a *cross-core private cache attack*. In such an attack, the attacker "remotely" evicts the victim's data from the victim's private cache to the LLC, and waits for the victim to access it and bring it back to the victim's private cache. Recently, the directory Prime+Probe attack [40] achieves this by building conflicts in the directory structure, which is used for tracking cache lines in private caches. Unfortunately, this method only works with non-inclusive LLCs. Instead, we present cross-core private cache attacks that work with *both inclusive and non-inclusive LLCs*. Our attacks work by manipulating cache coherence states using a prefetch instruction.

`PREFETCHW` is an x86 prefetch instruction introduced in



(a) M (b) S (c) E (d) I

Fig. 1: The four possible states of a private cache line, when using the MESI protocol.

2000. It is now available on all Intel Xeon Scalable processors and recent Core processors (since Broadwell). According to the technology manual [41], the function of this instruction is to prepare data for *future writes*. It is different from other prefetch instructions (e.g., `PREFETCHT0`) which only move the target cache line closer to the CPU core (i.e., to a higher cache level) to get ready for future accesses. `PREFETCHW` moves the cache line to the requesting core's L1 data cache (L1D cache), as well as sets the coherence state of the cache line to be *Modified*. This can accelerate future write operations from this requesting core, because a cache line in Modified state indicates that 1) the current private cache has exclusive ownership of this cache line, meaning a write operation on this cache line can be directly served by the private cache, and 2) this cache line is already marked as dirty, so the flag (i.e., the dirty bit) does not need to be changed when serving a write operation. For correctness, setting the coherence state of a cache line to Modified causes all copies of this cache line in other cores' private caches to be invalidated [42], [43].

In this work, we make two important observations regarding `PREFETCHW` on Intel processors. First, although its purpose is to accelerate future writes, `PREFETCHW` works on data with *read-only* permission. Second, the execution time of `PREFETCHW` is related to the current coherence state of the target data. With the first observation, an attacker on a different core than the victim can use `PREFETCHW` on the shared data between the attacker and the victim (which is usually read-only [1]), to evict this data from the victim's private cache. This is an essential step for building cross-core private cache attacks. In addition, the second observation means that the attacker can time the execution of `PREFETCHW` on the shared data between the attacker and victim to learn the coherence state changes of this data, which could be related to the victim's cache accesses.

Based on these two observations, we first propose two covert channel attacks: Prefetch+Load and Prefetch+Prefetch. In Prefetch+Load, the sender transmits a bit by prefetching (with `PREFETCHW`) the shared data between the sender and receiver (for "1") or not prefetching (for "0"). The receiver (on a different core) receives the bit by loading this data and timing the load to determine if it is a local private cache hit (for "0") or a remote private cache hit (for "1"). In Prefetch+Prefetch, the sender transmits a bit by loading (or not) the shared data, and the receiver receives the bit by prefetching the data and timing the prefetch instruction to determine whether the sender loaded. We show that our prefetch-based channels have very high capacities: on our Kaby Lake processor, when only using one shared cache line between the sender and receiver, the capacities are 840KB/s for Prefetch+Load, and 822KB/s for Prefetch+Prefetch, which are *the highest single-line capacities among all existing CPU cache covert channels*.

We then modify the covert channel attacks and build the Prefetch+**Re**load and Prefetch+Prefetch side channel attacks, which can be used to leak the victim's access patterns, similar to previous cache attacks (e.g., [1]–[7]). Prefetch+Prefetch can be directly used as a side channel attack by letting the victim be the sender, and the attacker be the receiver, since in this attack the sender transmits signals by accessing (or not) the shared data. However, in Prefetch+Load, the sender is sending signals by prefetching (or not), which is unlikely a side channel. Thus, we modify it and build Prefetch+Reload, where the attacker owns two threads running on different cores. The attacker first uses one thread to prefetch and waits for the victim's possible access, and then reloads using the other thread. When the attacker reloads, he will get a remote private cache hit if the victim accessed this data; otherwise he will get an LLC hit. Then, the attacker can determine the victim's behavior using timing information: a remote private cache hit and an LLC hit take different amount of time to finish. We show that our attacks can be deployed on Intel processors to leak secrets from real-world applications, and that they can be used in transient execution attacks, making those attacks faster (and more potent) than before.

To the best of our knowledge, our prefetch-based attacks are the first cross-core private cache side channel attacks that can work with both inclusive and non-inclusive LLCs.

Responsible disclosure. We have disclosed the security vulnerabilities we found in this paper to Intel, and Intel has allowed the submission.

II. Background

A. CPU Cache Architecture and Coherence Protocol

Cache architecture. Most CPU caches on modern x86 processors are divided into L1, L2, and L3. The L1 and L2 caches are very fast but relatively small. Typically, they are organized separately for each CPU core, and are thus often referred to as private caches. In contrast, the L3 cache, also known as last-level cache (LLC), is a larger but slower cache, shared among CPU cores.

Caches operate on fixed-size (e.g., 64 bytes) data blocks called cache lines. Additionally, caches are usually set-associative: a cache is organized into multiple cache sets. Every cache set consists of multiple equivalent cache ways, and each of them can store one cache line. The address bits of a cache line determine which cache set that this line is mapped to. Most LLCs in Intel processors are inclusive, meaning that data present in private caches are necessarily also present in the LLC (and conversely, data not in the LLC are not in the private caches). However, recent Intel server processors (e.g., Intel Xeon Scalable processors [40], [44]) use non-inclusive LLCs, i.e., cache lines in private caches may not be present in the LLC. For non-inclusive LLCs, a separate directory structure is required for tracking the cache lines that are in the private caches but not in the LLC.

When a CPU core performs a memory access request, it first checks whether the target cache line is present in its L1 or L2 cache. If present, the request results in a private cache hit; if not, it is a private cache miss and the core must further check the LLC (and the directory for a non-inclusive LLC). If the cache line is found, the request finishes and the data is sent to the CPU. If not, the cache forwards the request to the memory controller, which can read data from DRAM.

Cache coherence. In multi-core systems, a cache line can be present in multiple private caches, due to data sharing. A cache coherence protocol[1]is required for maintaining data consistency among the copies of a cache line in different private caches: each private cache line is assigned a coherence state, and the LLC needs to track this state to prevent the use of stale data. For inclusive LLCs, the coherence states of private cache lines are stored together with the tag array in the LLC since all the private cache lines are also in the LLC. For non-inclusive LLCs, the directory structure mentioned earlier is used for storing the coherence states of cache lines that are in the private caches but not the LLC.

Most modern x86 processors use variants of the MESI coherence protocol [42], [43]. In the rest of this section, we use inclusive cache as an example to introduce MESI. For non-inclusive caches, the protocol is essentially the same, except that a cache line in a private cache might not be present in the LLC. With MESI, there are four possible states of a private cache line:

- ***Modified (M)***, in which the cache line is only present in one private cache and is dirty, i.e., the copy of this cache line in the LLC contains stale data (Figure 1(a)). Additionally, when a private cache line is in M state, the current owner core has read/write permission for it.
- ***Shared (S)***, in which the cache line is present in one or more private caches and is clean, i.e., the data of this cache line matches all other copies (both in other private caches and the LLC). The current core can only read this cache line (Figure 1(b)).
- ***Exclusive (E)***, in which the cache line is only present in one private cache, and is clean (Figure 1(c)). The current core can read/write this cache line; however, a write operation will change the state of this cache line to M.
- ***Invalid (I)***, in which the cache line is invalid, and thus the current core has neither read nor write permission for it (Figure 1(d)).

[1]In this paper, we only focus on the cache coherence inside a processor; this should not be confused with the coherence among sockets (processors) [45].

With MESI, a memory request from a CPU core will sometimes 1) change the coherence state of the target cache line, and 2) take different amount of time to finish, depending on the coherence state of the target cache line.

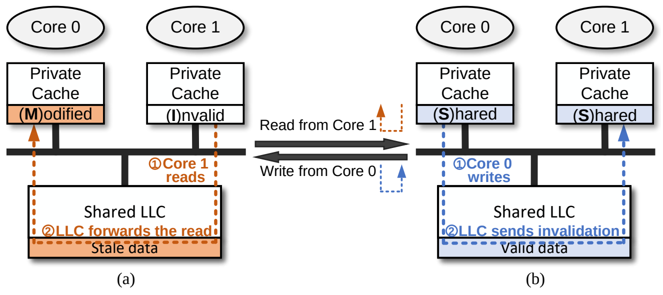


Fig. 2: The illustration of cache coherence state changes. The state of a line changes from M (shown in (a)) to S (shown in (b)) when a CPU core is loading it; conversely, the state changes from S to M when a CPU core is writing it. Dashed lines show the request path of the read/write operation.

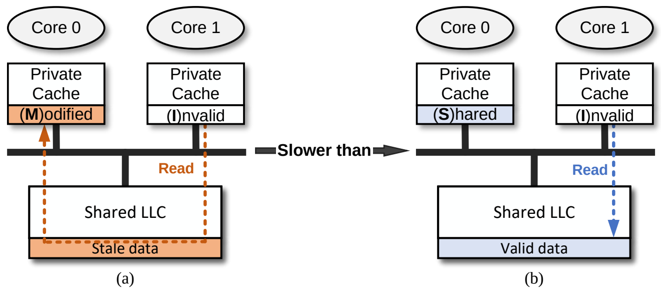


Fig. 3: The illustration of an LLC access with the target cache line in M state (a), and S state (b).

State transitions. There are many different coherence state transitions, we only discuss the two scenarios related to our attacks. First, as shown in Figure 2(a), when a CPU core (core 1) is reading a cache line that is present in the LLC and the private cache of another core (core 0) in M state, this read request will first miss in its private cache and then search the LLC. Although this target cache line can be found in the LLC, its content is potentially stale. Thus, the LLC will fetch the data from the owner private cache (in core 0) that contains the up-to-date data of this cache line, change the coherence state of this cache line in that private cache (in core 0) to S, update the content of this cache line in the LLC, and then return the updated cache line to the requesting core (core 1) as well as fill its private cache with a copy of this cache line in S state. Thus, after serving this read request, the target cache line is present in two private caches, and is in S state in both caches, as shown in Figure 2(b). This case is usually referred to as *remote private cache hit*.

Second, as shown in Figure 2(b), when a CPU core (core 0) is trying to write a cache line that is in S state in its own private cache, this private cache (in core 0) needs to send request to the LLC to acquire write permission before it can serve this write operation. As a result, the LLC will send invalidation signal(s) to the other private cache(s) that the cache line is present in (in core 1), and then change the state of the cache line in the private cache of the requesting core (core 0) to M so that the requesting core can write this cache line in its private cache. Thus, after this write operation, the target cache line is only present in the requesting core's private cache, and is in M state, as shown in Figure 2(a).

Timing difference. As one can observe in Figure 3, if a CPU core is reading a cache line that is not present in its private cache but is present in the LLC, the total latency it takes to finish this read request can be different when this cache line has different coherence states: a remote private cache hit is much slower than an LLC hit. When another core has a copy of this cache line in M state in its private cache, this request results in a remote private cache hit. As explained earlier, serving this request will require fetching data from the owner private cache. In contrast, when all the private cache copies of this cache line are in S state, the data of this cache line in the LLC is up-to-date. This means the LLC can serve this read request directly, resulting in an LLC hit. Due to these different execution paths, an LLC hit is much faster than a remote private cache hit. This has been observed by previous work [7] and has been verified in our experiments. On an Intel Core i7-6700 processor, an LLC hit takes less than 60 cycles to finish and a remote private cache hit takes about 90 cycles.

B. Prefetch

Prefetch is a technique to boost performance by fetching data and placing them closer to the CPU core (e.g., from the LLC to L1 cache) before they are needed. Prefetch can be performed in two ways: 1) hardware prefetch, which is implemented in cache hardware and is transparent to users (e.g., the adjacent cache line prefetcher); 2) software prefetch, which needs to be explicitly done by the programmer/compiler. Recent x86 CPUs offer many different instructions for software prefetch, such as `PREFETCHT0`, `PREFETCHT1`, `PREFETCHT2`, `PREFETCHNTA`, and `PREFETCHW`[2]. These instructions are used to hint the processor that a memory location is very likely to be accessed soon [41], then the processor will prefetch the corresponding data into certain level of cache, thereby accelerating future accesses to this data. Software prefetch is an important way to improve performance. For example, compilers sometimes inject prefetch instructions to accelerate for loops.

[2]Some CPU models (e.g., Intel Xeon Phi Processor 7200) use `PREFETCHWT1` instead of `PREFETCHW`.

C. Cache Side Channel Attacks

There are typically two types of cache attacks. The first type utilizes the contention on certain cache hardware (e.g., the ring interconnect [46] or L1 cache ports [47], [48]): the attacker passively monitors the latency of accessing this hardware resource to infer the victim's usage of it. Such attacks are usually referred to as contention based attacks or stateless attacks. The other type utilizes cache states: the attacker actively brings the cache line/cache set to a certain state, then lets the victim execute (which potentially changes the state), and later checks the state again to infer the victim's behavior. Such attacks are often referred to as eviction based attacks or stateful attacks. In this overview, we focus on stateful attacks because they are more numerous, and our proposed attacks are stateful. We further divide stateful attacks into private cache attacks and LLC attacks, based on whether the attacker evicts the victim's data from the LLC during the attack.

Private cache attacks. In private cache attacks, the attacker learns the victim's cache behavior by monitoring the state of the victim's data in the private cache. For example, in L1 Evict+Reload, the attacker evicts the victim's data (which is shared with the attacker) from the L1 cache to the L2 cache by building set conflicts, and waits for the victim's execution. Later the attacker accesses this data and times the access to determine it is in the L1 or L2 cache: if it is in the L1 cache, it means the victim accessed the data and brought it back to the L1 cache, otherwise the victim did not access. Private cache attacks could have high-bandwidth since they do not create slow DRAM accesses. However, most private cache attacks require the attacker to be on the same physical core with the victim (e.g., [5], [11]), and many of them further require SMT. This significantly limits the attacks, as cloud providers may allocate users to different cores and may disable SMT for security [37]–[39].

The directory Prime+Probe attack [40] and its optimization, the directory Prime+Scope attack [49], are an exception: they are cross-core private cache attacks. On a processor with a non-inclusive LLC, the attacker can "remotely" evict the victim's data from the victim's private cache to the LLC (but not to DRAM) by building conflicts in the directory.

LLC attacks. In LLC attacks (e.g., [1], [4], [50]), the attacker monitors the state of the victim's data in the LLC. The LLC is usually shared among CPU cores. Thus, different than private cache attacks, LLC attacks do not require the attacker to be on the same core as the victim. These attacks are considered more practical. However, DRAM accesses are usually involved in LLC attacks. To monitor the victim's access on the LLC data, the attacker needs to first evict the victim's data from the LLC to memory. For example, in Flush+Reload [1], the attacker uses `CLFLUSH` instruction to flush the victim's data from the LLC (and also the private caches), and later reloads this data and times this operation to determine whether the victim has brought this data back to the LLC. Therefore, the bandwidths of LLC attacks are bottlenecked by DRAM latencies.

```
1  void* thread0 (void* addr_d0, int expt_idx){
2      for(int i = 0; i < 1000000; i++){
3          /* check the experiment index*/
4          if(expt_idx == 0){
5              /* execute prefetchw on d0*/
6              prefetchw(addr_d0);}
7          /*let thread1 execute 1 iteration*/
8          wait_for_thread1();
9      }}
10
11 void* thread1 (void* addr_d0){
12     for(int i = 0; i < 1000000; i++){
13         /*let thread0 execute 1 iteration*/
14         wait_for_thread0();
15         int result = read_and_time(addr_d0);
16     }}
17
18
19 int main() {
20     /* open and map a file as read-only*/
21     int fd = open(FILE_NAME, O_RDONLY);
22     int* addr_d0 = mmap(fd, PROT_READ, ...);
23
24     /*pin thread0 on core0 and start thread0*/
25     /*pin thread1 on core1 and start thread1*/
26     ...
```

Listing 1: The code snippet for verifying Observation 1.

```
1  void* thread0 (void* addr_d0, int expt_idx){
2      for(int i = 0; i < 1000000; i++){
3          /* check the experiment index*/
4          if(expt_idx == 0){
5              read(addr_d0);}
6          /*let thread1 execute 1 iteration*/
7          wait_for_thread1()
8      }}
9
10 void* thread1 (void* addr_d0){
11     for(int i = 0; i < 1000000; i++){
12         /*let thread0 execute 1 iteration*/
13         wait_for_thread0();
14         int t1 = rdtscp(); /* read time stamp*/
15         prefetchw(addr_d0);
16         int result = rdtscp()-t1;
17     }}
18
19 int main() {
20     /* open and map a file as read-only*/
21     int fd = open(FILE_NAME, O_RDONLY);
22     int* addr_d0 = mmap(fd, PROT_READ, ...);
23
24     /*pin thread0 on core0 and start thread0*/
25     /*pin thread1 on core1 and start thread1*/
26     ...
```

Listing 2: The code snippet for verifying Observation 2.

III. Characterizing Data Prefetching

Among the prefetch instructions discussed in Section II-B, PREFETCHW (or PREFETCHWT1 on some CPU models) works slightly differently than the others. It not only brings the data close to the CPU core, but also changes the coherence state of the data: PREFETCHW places the target data cache line into the L1D cache[3] and sets the coherence state of this cache line to M. According to the technology manual [41], [51], the role of PREFETCHW is to *accelerate future writes on the target cache line*. As explained in Section II-A, the CPU core can directly write a cache line in its local L1 cache iff the state of this cache line is E/M. Thus, PREFETCHW pre-sets the coherence state of the target cache line to M so that future writes on this cache line will likely have an L1 hit. PREFETCHW sets the cache line state to M instead of E because writing a cache line in E state results in changing the state to M, and thus has higher latency than writing a cache line that is already in M state.

Most of the recent Intel desktop and server processors (since Broadwell) support PREFETCHW. When used appropriately, it can significantly improve performance. However, we make two observations about PREFETCHW on Intel processors, which can be leveraged to create security vulnerabilities.

Observation 1 *PREFETCHW successfully executes on data with read-only permission.*

We observe this by monitoring the coherence state changes of the data, using timing information. Specifically, as shown in Listing 1, we run a program with two threads (*thread*$_0$ and *thread*$_1$, both in one process), and pin them on different physical cores. We use mmap [52] to map part of a system file (e.g., glibc) as a read-only data block (in cache line size) in this program and name it d_0: both threads can only read d_0. If any thread tries to write d_0, it will trigger a segmentation fault. *thread*$_0$ and *thread*$_1$ both consist of a for loop with the same amount of iterations. In each iteration, *thread*$_0$ first executes, then waits for *thread*$_1$ to execute. After *thread*$_1$ finishes this iteration, they both move to the next iteration and repeat this procedure again. We use pthread mutex locking [53] to ensure that in each iteration *thread*$_0$ and *thread*$_1$ execute sequentially (the implementation details of locking is omitted in Listing 1).

We run the code in Listing 1 twice: in the first experiment (i.e., expt_idx = 0 in line 3), in each iteration of the for loop, *thread*$_0$ performs PREFETCHW on d_0, and then *thread*$_1$ loads d_0 as well as times the load. In the second experiment (i.e., expt_idx = 1 in line 3), in each iteration *thread*$_0$ stays idle and then *thread*$_1$ still loads d_0 and times the load.

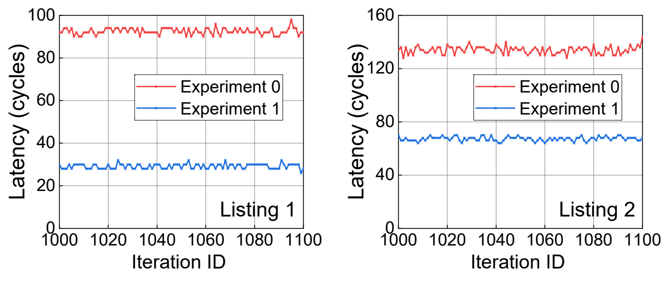


Fig. 4: The timing measurement results in *thread*$_1$ of Listing 1 and Listing 2.

Figure 4 shows a segment of the timing results from *thread*$_1$ (in line 15) in both of the above experiments on an Intel Core i7-6700 processor. Note that we observe similar results on other Intel processors that support PREFETCHW. In experiment 0, *thread*$_0$ prefetches d_0 in each iteration, which causes *thread*$_1$ to

[3]PREFETCHW can only be used on data but not instructions [41], [51]. Thus, the cache line will be brought into L1D cache.

take around 90 cycles to load d_0 after the prefetch. In contrast, in experiment 1, $thread_0$ stays idle, which causes $thread_1$ to take only around 30 cycles to load d_0. This timing difference infers that d_0 is in different states in the above two experiments. In experiment 0, every time when $thread_0$ prefetches, it will load d_0 to its private cache and set the coherence state of it to be M. According to MESI, explained in Section II-A, this will invalidate the copy of d_0 in the private cache of $thread_1$ (if it exists). Therefore, when $thread_1$ later loads d_0, it will have a *remote private cache hit* (see Figure 2). This load also changes the state of d_0 from M to S and fills a copy of it in the private cache of $thread_1$. Thus, the same cache behavior (i.e., invalidating the copy of d_0 in the private cache of $thread_1$) will happen when $thread_0$ prefetches in the next iteration. However, in experiment 1, since $thread_0$ is not prefetching, when $thread_1$ loads d_0, it will very likely have a *local private cache hit*, which is much faster than a remote private cache hit (30 cycles[4] vs. 90 cycles on the tested processor).

Rationale. We observe reliable cache state changes on read-only data when executing `PREFETCHW`, with a F-score of 1.0 ($n = 1000000$). This indicates that Intel processors very unlikely perform a *write permission check* when executing `PREFETCHW`. This does not cause any error in the architecture level, because `PREFETCHW` only has microarchitectural effects: although it can get a cache line ready for future writes, if later the program without write permission for this cache line actually tries to write it, it will still trigger a fault and likely terminate the process. However, later we will show that allowing coherence-based cache invalidation (which should only happen upon writes) on read-only data cause significant security problems. This is because in cache attacks based on shared memory, the attacker can manipulate the coherence state of the shared data (which is usually read-only) between the victim and attacker to monitor the victim's access on it.

Observation 2 *The execution time of* `PREFETCHW` *is related to the coherence state of the target cache line.*

We observe this with the program shown in Listing 2. We still use two threads pinned on different physical cores, and let them execute sequentially in each iteration of the for loop. Again, we run the program twice: in experiment 0 (expt_idx = 0 in line 3), in each iteration, $thread_0$ loads d_0, and then $thread_1$ performs `PREFETCHW` on d_0 and times the prefetch. In experiment 1, $thread_0$ stays idle and $thread_1$ still prefetches and times the prefetch in each iteration.

Figure 4 shows the execution time of `PREFETCHW` observed by $thread_1$ (in line 16) on our Intel Core i7-6700 processor in both experiments. In the first experiment, it always takes around 130 cycles for `PREFETCHW` to finish; however, in the second experiment it only takes around 70 cycles. This is because in the first experiment, after $thread_0$ loads d_0, the state of d_0 becomes S, and a copy of d_0 is filled into the private cache of $thread_0$ (see Figure 2). Then when $thread_1$ prefetches, it needs to change the state from S to M, which means it has to inform the LLC to invalidate the copy of d_0 in the private cache of $thread_0$. However, in the second experiment, since $thread_0$ stays idle, when $thread_1$ prefetches, d_0 is already in M state. Thus, in this case `PREFETCHW` does not cause any state change and finishes much faster.

[4]Due to the granularity of time stamp counters, this measured latency is in fact longer than the real private cache access latency.

Affected processors. We have tested these two observations on many Intel processors including the available 1st/2nd/3rd Generation Intel Xeon Salable Processors on AWS EC2, and five Intel desktop/server processors we own. As shown in Table I, Observation 1 is valid on all the tested processors, and Observation 2 is valid on most, excluding the Intel Xeon Platinum 8375C processor. On this processor, there is no difference on the execution time of `PREFETCHW` when the target data is different coherence states: `PREFETCHW` always takes 70 to 80 cycles to finish, even when the target data is not already in M state.

In general, we believe that Observation 1 should be valid on all Intel processors that support `PREFETCHW`, and Observation 2 should be valid on most of them. Note that all 1st/2nd/3rd Generation Intel Xeon Scalable processors and most Intel Core i7/i9 processors (other than the early generations before Broadwell) support `PREFETCHW`.

TABLE I: The evaluated processors for the two observations.

Processor	Microarch.	LLC Type	Observ.1	Observ.2
Intel Core i7-6700	Broadwell	Inclusive	✓	✓
Intel Core i7-6800K	Broadwell	Inclusive	✓	✓
Intel Core i7-7700K	Kaby Lake	Inclusive	✓	✓
Intel Core i9-10900X	Cascade Lake	Non-incl.	✓	✓
Intel Xeon Silver 4114	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8151	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8124M	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8175M	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8259CL	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8275CL	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8375C	Ice Lake	Non-incl.	✓	✗

IV. Prefetch-Based Covert Channel Attacks

Based on the observations in Section III, we build two cache covert channel attacks: Prefetch+Load and Prefetch+Prefetch. In this section, we first introduce the threat model, then discuss the details of each attack.

A. Threat Model

We assume that the two essential parties in the attack, the sender and receiver, are two unprivileged processes that are running on the same processor with multiple CPU cores. The sender and receiver can launch themselves on different physical cores (e.g., using `taskset` [54]). We also assume that the sender and receiver can share data; however, the shared data can be read-only (e.g., via shared library or page deduplication[5]), similar to previous attacks [1], [2], [5]–[7]. In addition, the sender and receiver should agree on pre-defined channel protocols, including the synchronization, core allocation, data encoding, and error correction protocols. We

Algorithm 1: Prefetch+Load Covert Channel

```
line0: the shared cache line between the sender and receiver
message[n]: the n-bit long message to transfer on the channel
Th0: the timing threshold for distinguishing local and remote private cache hit

Sender Algorithm
// Send 1 bit in each iteration.
for $i = 0$; $i < n$; $i{+}{+}$ do
    sync_with_receiver();
    if $message[i] == 1$ then
        Prefetch line0;
    else
        Do not prefetch;

Receiver Algorithm
// Detect 1 bit in each iteration.
for $i = 0$; $i < n$; $i{+}{+}$ do
    sync_with_sender();
    Access line0 and time the access;
    if $access\_time > Th0$ then
        Received a bit "1";
    else
        Received a bit "0";
```

Algorithm 2: Prefetch+Prefetch Covert Channel

```
line0: the shared cache line between the sender and receiver
message[n]: the n-bit long message to transfer on the channel
Th0: the timing threshold on PREFETCHW to distinguish M and S states

Sender Algorithm
// Send 1 bit in each iteration.
for $i = 0$; $i < n$; $i{+}{+}$ do
    sync_with_receiver();
    if $message[i] == 1$ then
        Load line0;
    else
        Do not load;

Receiver Algorithm
// Detect 1 bit in each iteration.
for $i = 0$; $i < n$; $i{+}{+}$ do
    sync_with_sender();
    Prefetch line0 and time the prefetch;
    if $prefetch\_time > Th0$ then
        Received a bit "1";
    else
        Received a bit "0";
```

do not have requirement on the LLC inclusivity; our attacks work with both inclusive and non-inclusive LLCs. We also do not require SMT; SMT can be turned off for security.

B. Prefetch+Load Attack

We build the first covert channel attack, Prefetch+Load, following Observation 1. Algorithm 1 shows the details of it. In this attack, the sender and receiver first agree on the shared cache line used to transmit information. Then in each iteration of the attack, the sender transmits a bit "1" by performing `PREFETCHW` on the shared cache line, or a bit "0" by idling. The receiver loads the same cache line and times the load to determine if it is a remote private cache hit or local private cache hit: the receiver receives a bit "1" when having a remote private cache hit, and otherwise receives a bit "0".

Note that different than the experiments in Section III, the sender and receiver cannot synchronize using pthread mutex locking, since they do not belong to the same process. Thus, we let the sender and receiver synchronize the transmission using time stamp counters (TSCs), as done in prior covert channel attacks (e.g., [1], [2], [5], [7], [49]).

C. Prefetch+Prefetch Attack

Our second attack, Prefetch+Prefetch, is based on Observation 2. As shown in Algorithm 2, in each iteration of the attack, the sender transmits "1" by loading the shared cache line, or transmits "0" by idling. After this, the receiver performs `PREFETCHW` on the shared cache line and times the prefetch to decode the bit: when the sender sends "1", it takes longer for the receiver to prefetch than when the sender sends "0". Prefetch+Prefetch follows the same synchronization method with Prefetch+Load.

[5]Page deduplication (a.k.a kernel same-page merging [55]) was originally created for virtual environments but is now included in OSs. Although many cloud providers no longer use it, it is usually still available in OSs [6].

V. Prefetch-Based Side Channel Attacks

A. Basic Idea and Assumptions

In the Prefetch+Prefetch covert channel attack, the sender is sending the signal by "accessing (or not) the shared data". Thus, this attack can be directly applied as a side channel attack to leak a victim's *access pattern* on the shared data: the victim is the sender, and the attacker is the receiver. This leakage (the victim's access pattern) is same as the one in previous cache attacks (e.g., [1], [2], [7], [8]).

However, Prefetch+Load cannot be directly used as a side channel, because the sender is transmitting the signal by "prefetching (or not) the shared data". In other words, the attacker (receiver) can only detect the victim's (sender's) prefetch patterns on the shared data. Since software prefetch is not as common as memory accesses in real-world applications, the attack opportunities are limited. However, we can modify the attack slightly to make it work more generally.

We term the new attack *Prefetch+**Reload***. The attacker prefetches the shared data to pre-set the coherence state, and then waits for the victim to possibly access this data. Later the attacker reloads the data (using a different thread on a different core, explained later) and uses the timing information to learn the current coherence state of the data, which leaks whether the victim has loaded this data (thus changing the coherence state). Different than Prefetch+Load, in Prefetch+Reload, the attacker needs to have two threads running on different cores.

Threat model. We assume a similar threat model as the one for the covert channels. First, the attacker is an unprivileged process that can 1) run on the same processor with the victim and 2) share data with the victim (e.g., through a shared library). The attacker aims at leaking the victim's access pattern on a shared data block, as in [1], [2]. In addition, the attacker can launch his thread(s) on different core(s) than the victim's.

For Prefetch+Reload, the attacker needs to have two threads running on different physical cores; but for Prefetch+Prefetch, there is still only one thread required in the attacker's process, same as the setup for covert channel attacks.

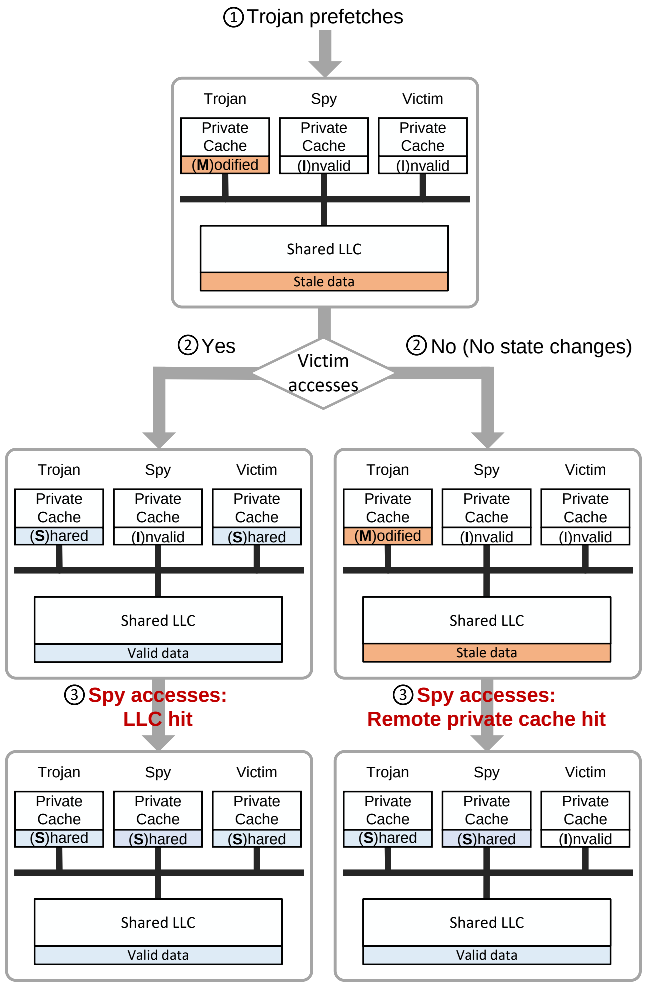


Fig. 5: The details of the three steps in Prefetch+Reload.

B. *Prefetch+**Re**load Attack*

In this attack, we assume that the attacker controls two threads named *Trojan* and *Spy*. Trojan and Spy should be located on different cores, which are also both different than the victim's core, i.e., Trojan, Spy, and the victim all run on different cores. As mentioned in Section II-A, the execution times of a remote private cache hit and an LLC hit are different. The Prefetch+Reload attacker uses this timing difference to observe cache state changes caused by the victim's accesses. Specifically, before the victim accesses the target shared cache line, Trojan executes `PREFETCHW` on this cache line, which invalidates the copies of this cache line in the victim's and Spy's private caches (if they exist), and places a copy of this cache line (in M state) in Trojan's private cache, as shown in Step 1 of Figure 5. Then, if the victim accesses this cache line, according to MESI, the coherence state changes from M to S, and the copy of this cache line in the LLC is updated (although the content did not change, see Section II-A) and is now valid (Step 2 in the left path of Figure 5).

Unfortunately, Trojan cannot observe this state change caused by the victim's access: if Trojan accesses (reloads) this cache line, he will get a private cache hit, no matter if the victim accessed this line or not. This is because the victim's read does not invalidate the copy in Trojan's private cache (Step 2 in the left path of Figure 5). However, Spy is able to distinguish whether the victim accessed this cache line. Trojan's original `PREFETCHW` invalidated the copy in Spy's private cache. Thus, if Spy now accesses this cache line, he will get an LLC hit if the victim has accessed this cache line after Trojan's prefetch (Step 3 in the left path of Figure 5); otherwise, he will get a remote private cache hit (Step 3 in the right path of Figure 5). We recall that Spy can distinguish these two situations by timing the access (the difference is over 30 cycles on our desktop processor). Based on this, we build Prefetch+Reload. Similar to previous cache attacks, each iteration in this attack contains three steps, as shown in Figure 5:

Step 1: Trojan performs `PREFETCHW` on the target cache line and becomes the exclusive owner of this cache line.

Step 2: The attacker waits for the victim's behavior: if the victim accesses this cache line, its coherence state will become S, meaning the copy in the LLC is now valid.

Step 3: Spy accesses this cache line and times the access to determine it was a remote private cache hit or an LLC hit. If it was a remote private cache hit, then the victim did not access this cache line; otherwise the victim did access.

LLC presence. Prefetch+Reload requires that the target shared cache line is present in the LLC, so that Spy can get an LLC hit in Step 3, if the victim has accessed this cache line. This is naturally true for inclusive LLCs, since all the cache lines in the private cache are also present in the LLC. However, it is not guaranteed for non-inclusive LLCs. In those caches, a cache line is directly brought into the private cache when loaded from DRAM, bypassing the LLC; it usually goes to the LLC when evicted from the private cache due to cache replacement [40], [44]. Thus, strictly speaking, it is the attacker's responsibility to ensure the presence of this cache line in the LLC, if it is non-inclusive. For example, before the attacker starts the attack loop, he can first build set conflicts in his private cache to evict this cache line to the LLC.

In fact, empirically we found that in Step 1, when `PREFETCHW` invalidates the copies of the target cache line in Spy and the victim's private caches, this cache line will be placed in the LLC if it does not already exist. Therefore, in practice the attacker does not need to explicitly place this cache line in the LLC.

C. *Prefetch+Prefetch Attack*

Following the Prefetch+Prefetch covert channel attack, we also build the Prefetch+Prefetch side channel attack. The attacker learns if the victim accessed the shared cache line by timing `PREFETCHW`. In contrast to the Prefetch+Reload side channel attack, each iteration in this attack only has two steps:

Step 1: The attacker prefetches the target shared cache line using `PREFETCHW`, and times this operation to learn

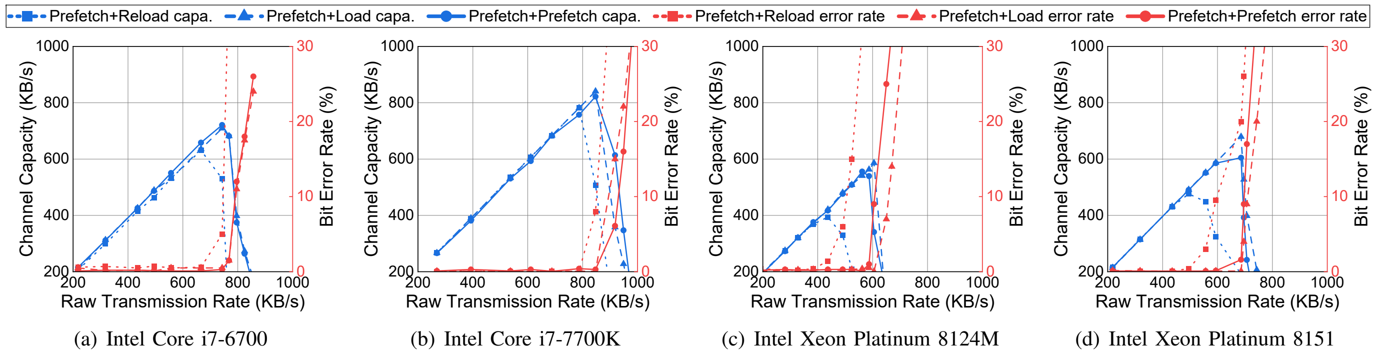


Fig. 6: The capacities and bit-error-rates of the prefetch-based channels on various Intel processors.

whether the victim accessed this cache line in the last iteration.

Step 2: The attacker waits for the victim's behavior.

As explained earlier in Section III, in Step 1 above, if the victim accessed this cache line, `PREFETCHW` executes slower; if the victim did not access, `PREFETCHW` executes faster.

In contrast to most previous cross-core cache attacks, which can only work on the LLC and require repeatedly evicting the target cache line to DRAM (e.g., Flush+Reload), the proposed prefetch-based attacks work on the private cache. Thus, the target cache line is always kept in the on-chip cache hierarchy. Compared to cross-core LLC attacks, cross-core private cache attacks have two benefits. First, higher bandwidth, since cache accesses are fast and are usually much faster than DRAM accesses. This is especially important when the attacks are used as covert channels. Second, stealthier, since there are less cache misses, especially LLC misses involved in the attacks [56]. *To the best of our knowledge, the proposed prefetch-based attacks are the first cross-core private cache side channel attacks that can work regardless of the LLC inclusivity.*

TABLE II: The specifications of the tested processors.

Platform	Desktop processors		Server processors	
	Core i7-6700	Core i7-7700K	Xeon Platinum 8124M	Xeon Platinum 8151
Microarchitecture	Broadwell	Kaby Lake	Skylake-SP	Skylake-SP
Num of cores	6	4	N/A[6]	N/A
Frequency	3.4 GHz	4.2 GHz	3.0 GHz	3.4 GHz
LLC type	Inclusive	Inclusive	Non-inclusive	Non-inclusive

VI. Evaluation

We evaluate the proposed covert channel and side channel attacks on modern Intel processors. For covert channel attacks, we evaluate the channel capacities, comparing them to previous cache covert channels on CPU. For side channel attacks, we demonstrate how they can be used to leak information from common applications. We also collect the cache miss rates of our attacks and compare them to SPEC 2017 [57] workloads to show the attack stealthiness. In addition, we also show how our attacks strengthen transient execution attacks.

A. Evaluation of Prefetch-Based Covert Channel Attacks

We implement Prefetch+Load, Prefetch+Prefetch, and Prefetch+Reload on four Intel processors, including two desktop processors and two server processors. Note that although Prefetch+Reload is introduced as a side channel attack in Section V, it can be a covert channel attack as well. Table II lists the specifications of the four tested processors. The two desktop processors have inclusive LLCs, and the server processors have non-inclusive LLCs.

We use one shared cache line between the sender and receiver to transmit secrets. Although using more shared cache lines or using channel coding techniques (e.g., [2]) may further improve the channel capacity [2], [7]; here we do not include them since we aim at showing the conservative results (i.e., the lower bounds).

TABLE III: The maximum capacities of the prefetch-based channels.

Platform	Desktop processors		Server processors	
	Core i7-6700 (3.4 GHz)	Core i7-7700K (4.2 GHz)	Xeon Platinum 8124M (3.0 GHz)	Xeon Platinum 8151 (3.4 GHz)
Prefetch+Reload	631 KB/s	**782 KB/s**	394 KB/s	476 KB/s
Prefetch+Load	709 KB/s	**840 KB/s**	586 KB/s	680 KB/s
Prefetch+Prefetch	721 KB/s	**822 KB/s**	556 KB/s	605 KB/s

We measure the channel capacity and bit error rate of each attack, under different transmission intervals. Although the raw transmission rate increases when decreasing the transmission interval, the bit error rate may also increase, especially when the interval is too short. To find the best transmission rate, we use the channel capacity metric (as in [46], [58]). This metric is computed by multiplying the raw transmission rate with $1 - H(e)$, where e is the bit error rate and H is the binary entropy function. The results are shown in Figure 6. The bit error rates of all three attacks stay low (lower than 0.6%) and are almost constant, when the raw transmission rate is under a threshold (e.g., 660 KB/s for Prefetch+Reload in

[6]We use Intel Xeon Scalable processors on Amazon AWS EC2 platform, and we leased four physical cores on the tested processors for our experiments.

Figure 6(a)). Thus, the channel capacity increases proportionally to the raw transmission rate. It reaches the peak when the raw transmission rate is around this threshold. Beyond this threshold, the increasing error rate causes a decrease in the channel capacity. The peak channel capacities of the three attacks are summarized in Table III. Prefetch+Reload always has lower capacity than the other two attacks because more cache operations are involved in each iteration of Prefetch+Reload.

Our prefetch-based attacks are faster than almost all existing cache attacks on x86 CPUs. First, for attacks tested on desktop processors, the ring interconnect contention based attack [46] is reported with a very high capacity which is 518 KB/s on a 4.0 GHz desktop processor. Flush+Reload and Flush+Flush have capacities of 298 KB/s and 496 KB/s on a 3.6 GHz desktop processor [2], respectively. Prime+Scope [49], the optimized attack for Prime+Probe, achieves 438 KB/s on a 3.5 GHz desktop processor. Second, most of the attacks that were tested on server processors, including the L1 LRU attack [5], the directory Prime+Probe attack [40], and the Flush+Coherence attack [7] have capacities of less than 200 KB/s. The directory version of Prime+Scope achieves 387 KB/s.

To the best of our knowledge, our attacks are only slower than Streamline [59]. This attack claims to achieve a capacity of 1801 KB/s. However, it has such a high channel capacity because the sender and receiver use 64 MB shared data to transmit secrets; our results are based on one shared cache line (64 B).

B. Evaluation of Prefetch-Based Side Channel Attacks

*1) **Side Channel Attack on Cryptographic Code**:* Our first attack targets cryptographic libraries, where the access patterns to some instructions are related to the value of the cryptographic key. More specifically, we target the square-and-multiply algorithm [60] which is used in GnuPG 1.4.13 for ciphers such as RSA [61] and ElGamal [62]: leaking the exponent e of this algorithm leaks the private key. As shown in Algorithm 3, in each loop iteration, it first executes a `sqr` and a `mod` instruction. Then, if the exponent bit is "1", a `mul` and another `mod` instruction are executed; otherwise they are skipped. Thus, by monitoring the access pattern to the cache lines that contain `sqr` and `mul`, the attacker is able to leak each bit of the exponent e and therefore the decryption key.

Algorithm 3: Square-and-multiply Exponentiation

```
Input: base $b$, modulo $m$, exponent $e = (e_{n1}...e_0)_2$
Output: $b^e$ mod $m$

$r \leftarrow 1$
for $i = n-1$; $i >= 0$; $i--$ do
    $r \leftarrow r^2$  mod $n$
    if $e_i == 1$ then
        $r \leftarrow r * b$  mod $n$
```

Implementation. As done in the Flush+Reload attack on GnuPG [1], we use `mmap` to map the pages that contain `sqr` and `mul` into the attacker's address space. Note that during the execution of the victim (GnuPG), the cache lines containing those instructions are brought into the victim's L1 instruction cache (L1I cache). However, since we map the instruction pages as data blocks in the attacker's address space, the same cache lines containing those instructions are brought to the attacker's L1D cache. Thus, although `PREFETCHW` can only prefetch cache lines into L1D cache, it can still leak the victim's access patterns to instructions.

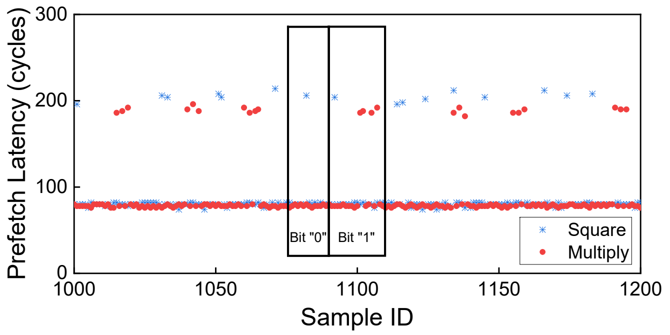


Fig. 7: A segment of the prefetch latencies measured in Prefetch+Prefetch while attacking GnuPG; part of the the exponent e shown here is "111001011001".

Results. For simplicity, we only show the attack results of Prefetch+Prefetch on the Intel Xeon Platinum 8151 processor. However, we have performed this attack on other processors listed in Table II too, using both Prefetch+Prefetch and Prefetch+Reload. Here we use a waiting latency of 500 cycles in each iteration of Prefetch+Prefetch. Figure 7 shows the timing measurement results from the attacker for 200 samples: a lower prefetch latency (less than 100 cycles) indicates that the victim did not access the target cache line during the last iteration; a higher prefetch latency (around 200 cycles) means the victim did access. As explained above, an access to `sqr` followed by an access to `mul` indicates a bit "1", and two consecutive accesses to `sqr` (one from the current iteration, one from the next iteration) indicate a bit "0" (in the current iteration). Thus, part of the exponent e shown in Figure 7 is "111001011001". The average attack accuracy is 96.2%.

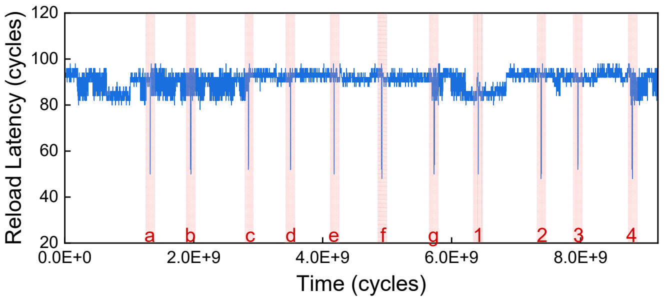


Fig. 8: The access latencies measured in Step 3 of Prefetch+Reload when a user types "abcdefg1234" in gedit; we monitor address 0x7b980 of libgdk.so.

*2) **Side Channel Attack on Keystroke Timing**:* Our second attack focuses on leaking the precise timing information of keystrokes, i.e., detecting when a keyboard input occurs. This

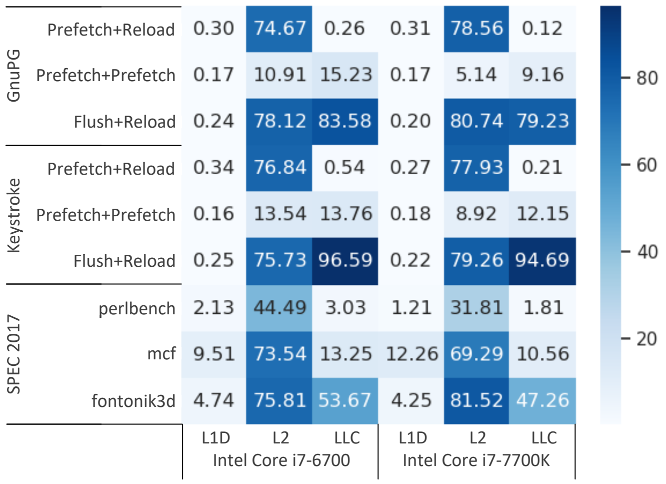
leakage is important since it can assist reconstructing typed words from users [63]–[65]. Previous work shows that certain functions in graphics libraries are called when a keystroke happens (e.g., [2], [66]). Thus, we can monitor the accesses to the cache lines containing these functions to detect keystrokes.
Implementation. We attack an address in the shared GDK library which is invoked when processing keystrokes. Specifically, we launch gedit as the victim, and input keystrokes in it. At the same time, we run the prefetch-based attacks to monitor accesses to the address selected in the GDK library, and record the timing measurement results. The attacker process raises an alarm when a keystroke is detected.
Results. Figure 8 shows the timing trace collected by Prefetch+Reload when the user is typing "abcdefg1234" in gedit, on our Intel Core i7-6700 processor. Again, the attack has been done the other desktop processor too (but not on the server processors since EC2 instances do not come with GUI). As one can observe, when a keystroke occurs, the reload operation (in Step 3 of Prefetch+Reload) takes around 50 cycles to finish; it takes over 80 cycles to reload when there is no keystroke. This significant timing difference makes keystrokes very detectable. During the attack, we observe zero false positives and zero false negatives.

*3) **Attack Stealthiness**:* Since most previous cross-core attacks have a large amount of cache misses, especially LLC misses, prior work (e.g., [56]) has suggested detecting the attacker process by monitoring the cache miss rates of each process. In this section, we show that our prefetch-based attacks cannot be detected in this way. We use performance counters to collect the attacker's miss rates to L1D cache, L2 cache and LLC, while attacking GnuPG and user inputs (keystrokes), respectively. The results are in Figure 9. As one can observe, the LLC miss rates of Prefetch+Reload are less than 1% in both attack scenarios. For Prefetch+Prefetch, the LLC miss rates are a little higher (about 10%) because there are fewer LLC accesses.

To compare, in Figure 9 we also list the attacker's cache miss rates of Flush+Reload as well as the cache miss rates of three typical SPEC 2017 workloads that have high, medium, and low memory access locality, respectively. From this figure, the LLC miss rates of our prefetch-based attacks are lower than or similar to the ones of SPEC workloads. In contrast, in Flush+Reload, the attacker has very high LLC miss rates: they are about 70% to 80% when attacking GnuPG, and are over 90% when attacking user inputs. These are much higher than the LLC miss rates of all the three SPEC workloads. In fact, from our experiments, there are only two workloads (out of 23) in SPEC that have an LLC miss rate higher than 30%.

Although in the prefetch-based attacks, the target cache line is always evicted from L1D cache, the miss rates to L1D cache are still less than 1% because other data accesses (to the cache lines which are not used for leaking secrets) cause a lot of L1 hits. This means we cannot detect the attacks by monitoring the L1 miss rate. In addition, Prefetch+Reload has high miss rates to L2 cache (similar to Flush+Reload). However, as shown in Figure 9, SPEC 2017 workloads can also have high miss rates to L2 cache, making it hard to detect those attacks through L2 miss rates.

Our attacks may be detected using performance counters related to software prefetch. For example, `ls_pref_instr_disp.store_prefetch_w` counts the amount of dispatched `PREFETCHW` instructions; `ls_inef_sw_pref.data_pipe_sw_pf_dc_hit` counts the amount of prefetch instructions that did not fetch data outside of the private cache. These two performance counters together may identify a process that repeatedly uses `PREFETCHW` on data that is not in the local private cache. However, such a process is not necessarily an attacker, because a program using `PREFETCHW` to accelerate a loop could also have this behavior: in each iteration, the data required for the next iteration is prefetched from the LLC. Thus, using these two performance counters may result in false positives. Other finer-grained performance counters such as the ones related to read for ownership (RFO) requests may give a better attacker signature. We leave finding effective combinations of performance counters as future work.



		Intel Core i7-6700 L1D	Intel Core i7-6700 L2	Intel Core i7-6700 LLC	Intel Core i7-7700K L1D	Intel Core i7-7700K L2	Intel Core i7-7700K LLC
GnuPG	Prefetch+Reload	0.30	74.67	0.26	0.31	78.56	0.12
	Prefetch+Prefetch	0.17	10.91	15.23	0.17	5.14	9.16
	Flush+Reload	0.24	78.12	83.58	0.20	80.74	79.23
Keystroke	Prefetch+Reload	0.34	76.84	0.54	0.27	77.93	0.21
	Prefetch+Prefetch	0.16	13.54	13.76	0.18	8.92	12.15
	Flush+Reload	0.25	75.73	96.59	0.22	79.26	94.69
SPEC 2017	perlbench	2.13	44.49	3.03	1.21	31.81	1.81
	mcf	9.51	73.54	13.25	12.26	69.29	10.56
	fontonik3d	4.74	75.81	53.67	4.25	81.52	47.26

Fig. 9: The cache miss rates of 1) the attacker processes in various cache attacks and 2) three workloads in SPEC 2017[7].

C. *Prefetch-Based Channels in Transient Execution Attacks*

Transient execution attacks such as Spectre [26] and Meltdown [27] usually require a microarchitectural covert channel to transfer the secrets to the attacker. Currently, most transient execution attacks (e.g., [26]–[29], [67]) use the Flush+Reload channel because it is simple, reliable, and ubiquitous. Here we demonstrate that prefetch-based channels can also work with transient execution attacks to leak secrets, and may even work better than Flush+Reload. We use Spectre v1 as an example to show the details and benefits of using prefetch-based channels in transient execution attacks.
Higher bandwidth. When using Flush+Reload, the sender operation in Spectre is a memory access where the ad-

[7]We only show the results from our own Intel processors, because we do not have access to the hardware performance counters on AWS EC2.

dress depends on the secret value. Since Prefetch+Reload and Prefetch+Prefetch use the same sender function as Flush+Reload, a victim program vulnerable to Spectre with Flush+Reload is also vulnerable to Spectre with Prefetch+Reload and Prefetch+Prefetch. We have verified this using the Spectre v1 PoC code [68]. We modify it accordingly such that Prefetch+Reload or Prefetch+Prefetch is used in the attacker code; the victim remains the same. In addition, as observed in prior work [27], the leakage rate of a transient execution attack is significantly affected by the capacity of the covert channel used in the attack. Since Prefetch+Reload and Prefetch+Prefetch have much higher capacities than Flush+Reload, Spectre works faster with these two channels. For example, on our Intel Core i7-6700 processor, when leaking an 8-bit secret in each transmission, the leakage rate of Spectre is 3.02 times and 1.61 times as fast when using Prefetch+Prefetch and Prefetch+Reload, respectively, as compared to Flush+Reload. The results on other processors are shown in Appendix A.

```
if (x+n < array1_size)
{
    y0 = array2[0][array1[x] * 4096];
    y2 = array2[1][array1[x+1] * 4096];
    ...
    yn = array2[2][array1[x+n] * 4096];
}
```

Listing 3: The Spectre v1 code example when a bounds check is followed by multiple secret accessing and encoding operations. This code is essentially a for loop in a conditional branch; we show the unrolled version for clarity.

More leakage in the transient window. When using Spectre with Flush+Reload, the data access for sending (encoding) the secret in the transient window is a *slow DRAM access*, since this data was flushed by the attacker. In contrast, the data access for secret encoding is a *remote private cache hit* when using Prefetch+Reload or Prefetch+Prefetch, which is usually faster than a DRAM access. This indicates that within the same transient window, more encoding operations can be performed using the two prefetch-based channels than Flush+Reload, and thus more secrets may be leaked. An example Spectre v1 gadget that can benefit from this is shown in Listing 3. There are n operations in the branch, where each operation accesses a secret and encodes it to a cache index. The secrets are array1[x] to array1[x+n] (when x is out of bounds); each of the secrets is encoded to an index of a sub-array in array2. The more of these n operations we can perform in the transient window, the more secrets we can leak out at once.

This gadget might be found in a victim; it is essentially the original Spectre v1 gadget with multiple secrets accessed and encoded in the branch (instead of one). Additionally, in the scenario where the attacker has control over the gadget (e.g., spectre-type-meltdown[8]), the attacker can build such a gadget to leak multiple secrets in one transient window and thus accelerate the attack. We still prove this with the Spectre v1 PoC code and modify the attacker code to use Prefetch+Reload or Prefetch+Prefetch. We also modify the victim code to simulate the gadget in Listing 3 where n secrets are accessed and encoded in the branch. We run this code and collect the amount of these n secrets the victim can transmit within one transient window, and draw the histograms in Figure 10. We omit the results when leaking by Prefetch+Reload since its encoding stage is same as the one of Prefetch+Prefetch.

[8]Spectre can be used for exception suppression in Meltdown.

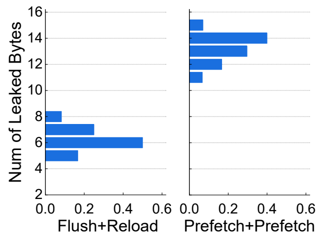
On the desktop processors, the victim can transmit up to 17 8-bit secrets speculatively when using Prefetch+Reload or Prefetch+Prefetch, while the victim can transmit at most 8 secrets when using Flush+Reload. However, on server processors, the amount of transmitted secrets when using prefetch-based channels is only slightly larger than the one when using Flush+Reload. This is because on these processors, the latency of a remote private cache hit is much longer, compared to desktop processors (160 cycles vs. 90 cycles). Note that although same-core private cache attacks, such as the L1 LRU attack [5], can also achieve more secret encodings in a transient window than Flush+Reload, these attacks are less practical, because they are limited by the number of private cache sets. In these attacks, secret values are encoded into the cache set index instead of cache line index.

Other transient execution attacks. All of the three prefetch-based channels can be used in transient execution attacks when the attacker has full control of the gadget (e.g., Meltdown). As shown above, Prefetch+Reload and Prefetch+Prefetch has faster encoding operations than Flush+Reload, enabling more leakage in a transient window. The same is true for Prefetch+Load, since a remote private cache hit for `PREFETCHW` is usually faster than a DRAM access. In a Meltdown PoC with the three prefetch-based channels, we can reliably leak 8 bytes in the transient window on Our Intel Core i7-6700 processor; we can only leak 6.1 bytes on average when using Flush+Reload. An example gadget to achieve is shown in Appendix B.

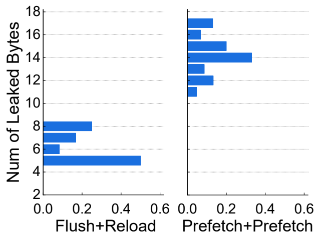
VII. Discussion

A. Attack Reliability

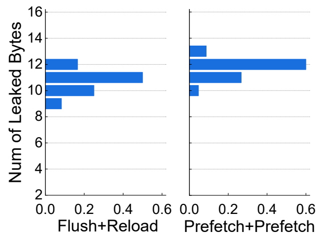
According to Intel [41], a prefetch instruction will not fetch any data when the request buffer between the L1 and L2 cache is full. This may reduce the performance of the prefetch-based attacks, when SMT is available and a memory-intensive thread is located on the same core as the attacker thread. We verified this by running `stress -m 1` in a co-located thread (i.e., the hyperthread sibling) of the attacker thread: this causes many prefetch instructions from the attacker to be ignored, which significantly reduces the attack performance. For example, on our Intel Core i7-6700 processor, the capacity of Prefetch+Prefetch is reduced to 56 KB/s. However, SMT enables many security vulnerabilities (e.g., [69]) and thus is often suggested to be disabled. In fact, if SMT is available, the attacker can always launch same-core private cache attack instead. Our cross-core private cache attacks target the scenarios where same-core attacks are impractical or impossible.



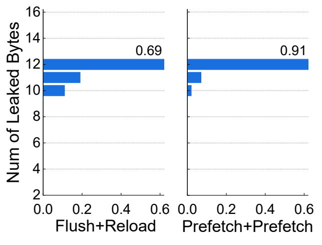
(a) Intel Core i7-6700



(b) Intel Core i7-7700K



(c) Intel Xeon Platinum 8124M



(d) Intel Xeon Platinum 8151

Fig. 10: The distributions of the amount of secret bytes that can be accessed and encoded in a transient window, when leaking by Flush+Reload and Prefetch+Prefetch, respectively.

B. *Prefetch-Based Attacks on AMD*

Modern AMD processors also support `PREFETCHW`; this instruction was originally invented by AMD [51], and was later adopted by Intel. We performed the same experiments as the ones in Section III on AMD desktop and server processors. However, from our experiments, `PREFETCHW` does not cause any coherence state changes on data with read-only permission; it only works on data with write permission. Thus, we believe that AMD processors actually have permission checks for `PREFETCHW`.

C. *Related Work*

Prefetch-based attacks. Gruss et al. [70] made two observations about prefetch instructions on Intel processors. They found that the execution time of a prefetch instruction, such as `PREFETCHT0`, leaks the translation levels of inaccessible kernel addresses. Using this, they built an attack to break Kernel Address Space Layout Randomization (KASLR). They also observed that prefetch instructions change the cache state of inaccessible kernel memory, but recent work [71] proved this incorrect. In fact, their observation is the result of transient execution caused by a Spectre gadget in the kernel, not the prefetch instruction.

Very recently, Lipp et al. [72] observed that on AMD processors, the timing (and power consumption) of a prefetch instruction on an inaccessible kernel address leaks the translation level and TLB state of this address. They used this to break KASLR and leak kernel memory (with Spectre) on AMD processors. These two prefetch attacks are orthogonal to our attacks. They focus on specifically attacking the kernel; we instead focus on building general cache timing attacks.

Regarding hardware prefetch attacks, Shin et al. [73] attacked OpenSSL, leaking the private key by leveraging the Intel stride prefetcher. Rohan et al. [74] reverse-engineered the stream prefetcher on Intel processors, using it to build a covert channel.

Cache coherence vulnerabilities. Although we are the first to propose cross-core private cache side channel attacks leveraging cache coherence protocol invalidations, cache coherence protocols have been exploited in many different attacks. Trippel et al. [75] discovered that a transient write may change the coherence state of the target data, which can be used as a covert channel in transient execution attacks. In addition, previous studies [26], [76], [77] mention that “bouncing” cache lines between private caches may be used as a replacement for `CLFLUSH` or set conflicts in Spectre and Rowhammer attacks. However, in this method, coherence states are manipulated by write operations. This means it requires that at least part of the target cache line happens to contain writable data (unless Meltdown-RW [78], [79] can be exploited). Unfortunately, as discussed in [77], this requirement is impractical for general side channel attacks.

Prior work [45], [80] built cross-core attacks on AMD and ARM processors, respectively, based on cache coherence. An Evict+Reload attack on Intel processors with non-inclusive LLCs was proposed in [40]. In these three attacks, the attacker learns the victim’s behavior by distinguishing between remote private cache hits and DRAM accesses. A variant of Flush+Reload attack on x86 processors was proposed in [7]. It works by distinguishing between remote private cache hits and LLC hits. These attacks are more general than the ones discussed earlier, but they all suffer from low bandwidth as DRAM accesses are involved in the attacks.

D. *Mitigations*

Our attacks can be prevented through modifications on the microarchitecture behavior of `PREFETCHW`. The complete protection is two-fold. First, `PREFETCHW` should perform write permission checks, just as a regular memory write instruction, and trigger a fault or ignore this instruction if the target data is not writable. Second, `PREFETCHW` should execute in constant time. These modifications may introduce some performance overhead. We do not suggest eliminating `PREFETCHW` since it is important for improving write performance.

Similar to prior cache attacks, our attacks also work by manipulating and monitoring cache states. Thus, defenses against prior cache attacks (e.g., [81]–[85]) may also work on our attacks. For example, DAWG [82] allows replicated cache copies of the data shared across security domains: each domain gets their own copy of this data in cache. Thus, the attacker cannot monitor the coherence state changes from a victim who is in another domain, which would stop our attacks.

VIII. Conclusion

In this paper, we proposed the first two cross-core private cache side channel attacks that work with both inclusive and non-inclusive LLCs. One of the prefetch instructions on x86 processors, `PREFETCHW`, prepares the data for future writes by modifying the coherence state of the data. In this work, we made two important microarchitectural observations on

PREFETCHW. First, it works on data with read-only permission. Second, its execution time is related to the coherence state of the target data. Given these observations, the coherence state modifications by PREFETCHW enable significant security vulnerabilities. Using PREFETCHW, we first built two covert channel attacks which have very high capacities. We also demonstrated that these high-capacity covert channels enable more powerful transient execution attacks. We then slightly modified the covert channel attacks to build two side channel attacks and showed that these attacks leaked information from real-world applications.

IX. Acknowledgement

We thank Daniel Weber for the help on the Meltdown implementation, and Daniel Gruss for his valuable comments. This work is supported in part by US National Science Foundation #1422331, #1535755, #1617071, #1718080, #1725657, #1910413, and #2011146.

References

[1] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security Symposium*, 2014.

[2] 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*, 2016.

[3] Y. A. Younis, K. Kifayat, Q. Shi, and B. Askwith, "A new prime and probe cache side-channel attack for cloud computing," in *IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, 2015.

[4] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE symposium on security and privacy (S&P)*, 2015.

[5] W. Xiong and J. Szefer, "Leaking information through cache lru states," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[6] 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.

[7] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[8] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security Symposium*, 2015.

[9] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[10] C. Percival, "Cache missing for fun and profit," 2005.

[11] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*, 2006.

[12] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[13] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[14] 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 *ACM SIGSAC conference on Computer and communications security (CCS)*, 2009.

[15] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *USENIX Security Symposium*, 2012.

[16] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *ACM SIGSAC conference on Computer and communications security (CCS)*, 2012.

[17] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[18] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[19] O. Aciiçmez, "Yet another microarchitectural attack: exploiting i-cache," in *ACM workshop on Computer security architecture*, 2007.

[20] O. Acıiçmez 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*, 2008.

[21] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+ probe 1, javascript 0: Overcoming browser-based side-channel defenses," in *USENIX Security Symposium*, 2021.

[22] L. G. Bruinderink, A. Hülsing, T. Lange, and Y. Yarom, "Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme," in *International Conference on Cryptographic Hardware and Embedded Systems*, 2016.

[23] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *USENIX Security Symposium*, 2019.

[24] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+ abort: A timer-free high-precision l3 cache attack using intel tsx," in *USENIX Security Symposium*, 2017.

[25] T. Hornby, "Side-channel attacks on everyday applications: Distinguishing inputs with flush+ reload," *BlackHat USA*, 2016.

[26] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[27] 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*, 2018.

[28] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[29] 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 *IEEE Symposium on Security and Privacy (S&P)*, May 2019.

[30] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[31] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[32] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *European Symposium on Research in Computer Security*, 2019.

[33] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: Automated synthesis of hardware exploits and security litmus tests," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[34] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[35] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[36] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, *et al.*, "Speculative interference attacks: Breaking invisible speculation schemes," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[37] L. Armasu, "Openbsd will disable intel hyper-threading to avoid spectre-like exploits (updated)," 2018. Available at https://www.tomshardware.com/news/openbsd-disables-intel-hyper-threading-spectre,37332.html.

[38] T. Claburn, "Rip hyper-threading? chromeos axes key intel cpu feature over data-leak flaws – microsoft, apple suggest snub," 2019. Avail-

able at https://www.theregister.co.uk/2019/05/14/intel_hyper_threading_mitigations/.

[39] A. Marshall, M. Howard, G. Bugher, B. Harden, C. Kaufman, M. Rues, and V. Bertocci, "Security best practices for developing windows azure applications," *Microsoft Corp*, 2010.

[40] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[41] "Intel® 64 and ia-32 architectures optimization reference manual." Available at https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html.

[42] P. Conway and B. Hughes, "The amd opteron northbridge architecture," *IEEE Micro*, 2007.

[43] "Intel quickpath architecture," 2012. Available at http://www.intel.com/pressroom/archive/reference/whitepaperQuickPath.pdf.

[44] "Intel® xeon® scalable processor: The foundation of data centre innovation," 2017. Available at https://simplecoreger.intel.com/swdevcon-uk/wp-content/uploads/sites/5/2017/10/UK-Dev-Con_Toby-Smith-Track-A_1000.pdf.

[45] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *ACM on Asia conference on computer and communications security (Asia CCS)*, 2016.

[46] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring (s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *USENIX Security Symposium*, 2021.

[47] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *International Journal of Parallel Programming*, 2019.

[48] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, 2017.

[49] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+ scope: Overcoming the observer effect for high-precision cache contention attacks," 2021.

[50] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Annual Design Automation Conference (DAC)*, 2016.

[51] "3dnow! technology manual," 2000. Available at https://www.amd.com/system/files/TechDocs/21928.pdf.

[52] "mmap(2) — linux manual page." Available at https://man7.org/linux/man-pages/man2/mmap.2.html.

[53] "pthread_mutex_lock(3p) — linux manual page." Available at https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html.

[54] "taskset(1) — linux manual page." Available at https://man7.org/linux/man-pages/man1/taskset.1.html.

[55] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *Proceedings of the linux symposium*, 2009.

[56] M.-M. Bazm, T. Sautereau, M. Lacoste, M. Sudholt, and J.-M. Menaud, "Cache-based side-channel attacks detection through intel cache monitoring technology and hardware performance counters," in *International Conference on Fog and Mobile Edge Computing (FMEC)*, 2018.

[57] "SPEC CPU 2017."

[58] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *USENIX Security Symposium*, 2016.

[59] G. Saileshwar, C. W. Fletcher, and M. Qureshi, "Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[60] D. M. Gordon, "A survey of fast exponentiation methods," *J. Algorithms*, 1998.

[61] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, 1978.

[62] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Inf. Theor.*, 2006.

[63] K. Zhang and X. Wang, "Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems.," in *USENIX Security Symposium*, 2009.

[64] D. X. Song, D. A. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on ssh.," in *USENIX Security Symposium*, 2001.

[65] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical cache attacks from the network," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[66] D. Wang, A. Neupane, Z. Qian, N. B. Abu-Ghazaleh, S. V. Krishnamurthy, E. J. Colbert, and P. Yu, "Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries.," in *Network and Distributed System Security Symposium (NDSS)*, 2019.

[67] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[68] "Spectrepoc."

[69] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *USENIX Security Symposium*, 2018.

[70] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[71] M. Schwarzl, T. Schuster, M. Schwarz, and D. Gruss, "Speculative dereferencing: Reviving foreshadow," in *International Conference on Financial Cryptography and Data Security*, 2021.

[72] M. Lipp, D. Gruss, and M. Schwarz, "Amd prefetch attacks through power and time," in *USENIX Security Symposium*, 2022.

[73] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[74] A. Rohan, B. Panda, and P. Agarwal, "Reverse engineering the stream prefetcher for profit," in *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020.

[75] C. Trippel, D. Lustig, and M. Martonosi, "Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," *arXiv preprint arXiv:1802.03802*, 2018.

[76] J. Horn, "CPU security bug: information leak using speculative execution," 2017. Available at https://bugs.chromium.org/p/project-zero/issues/attachmentText?aid=287305.

[77] A. Fogh, "Row hammer, java script and mesi," 2016. Available at https://dreamsofastone.blogspot.com/2016/02/row-hammer-java-script-and-mesi.html.

[78] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[79] M. Dworkin, "Recommendation for block cipher modes of operation. methods and techniques," tech. rep., National Inst of Standards and Technology Gaithersburg MD Computer security Div, 2001.

[80] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *USENIX Security Symposium*, 2016.

[81] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," in *International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021.

[82] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[83] "Security best practices for side channel resistance." Available at https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/security-best-practices-side-channel-resistance.html.

[84] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in xen," in *ACM workshop on Cloud computing security workshop*, 2011.

[85] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *Annual International Symposium on Computer Architecture (ISCA)*, 2012.

Appendix

A. The Leakage Rate of Spectre v1

TABLE I: The leakage rate of Spectre v1 when using Prefetch+Reload and Prefetch+Prefetch, respectively, normalized to Flush+Reload.

	Desktop processors		Server processors	
Model	Core i7-6700 (3.4 GHz)	Core i7-7700K (4.2 GHz)	Xeon Platinum 8124 (3.0 GHz)	Xeon Platinum 8151 (3.4 GHz)
Prefetch+Reload	1.61	2.40	1.57	1.64
Prefetch+Prefetch	3.02	3.94	2.01	2.08

B. The Meltdown Gadget

```
#define encode(x, b) ((((x) >> (b * 8)) & 0xff))
#define SPACING 4096
char mem[8][SPACING * 256];

uint64_t secret = *(uint64_t*)secret_addr;
memaccess(mem[0] + encode(secret, 0) * SPACING);
memaccess(mem[1] + encode(secret, 1) * SPACING);
memaccess(mem[2] + encode(secret, 2) * SPACING);
memaccess(mem[3] + encode(secret, 3) * SPACING);
memaccess(mem[4] + encode(secret, 4) * SPACING);
memaccess(mem[5] + encode(secret, 5) * SPACING);
memaccess(mem[6] + encode(secret, 6) * SPACING);
memaccess(mem[7] + encode(secret, 7) * SPACING);
```

Listing 1: The example Meltdown gadget where an access to the 64-secret is followed by eight secret encoding operations.