Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks

CS6630: Reading Assignment

Prasanna Bartakke

EE19B106

Introduction

Programmers can use prefetch instructions to boost the performance of their programs. Intel processors have multiple prefetch instructions, `PREFETCHW` being one of them. The authors found two security vulnerabilities in `PREFETCHW`.

1. This instruction can execute on data with read-only permissions.
2. This instruction's execution time leaks the target data's coherence state.

This paper aims to exploit the security problems of the prefetch instructions. Based on these issues, two cross-core private cache attacks, Prefetch+Prefetch and Prefetch+Reload, are built. These attacks work on both inclusive and non-inclusive LLCs.

CPU Cache Architecture and Coherence Protocol

Caches on modern x86 processors are divided into three levels, L1, L2 and L3 caches. The L1 and L2 caches are high-speed but relatively small. Each core has separate L1 and L2 caches called private caches. On the other hand, LLC(L3 cache) is a more extensive and slower cache shared among CPU cores.

LLCs on most of the Intel processors are inclusive. The data in private caches is necessarily present in the LLC, and data not in LLC is not in the private cache. Some of the recent Intel Xeon processors have non-inclusive LLCs. A separate directory structure is maintained for non-inclusive LLCs to track the cache lines in the private cache that are not present in the LLC.

Due to data-sharing in multi-core systems, a cache line can be present in multiple private caches. A cache coherence protocol is required to maintain consistency among the copies

of the cache line across the different private caches. The LLC tracks this state to prevent the use of invalid data. The most commonly used cache coherence protocol is the MESI protocol.

MESI Protocol

The MESI protocol changes the coherence state of the target cache line after a memory request from a CPU core.

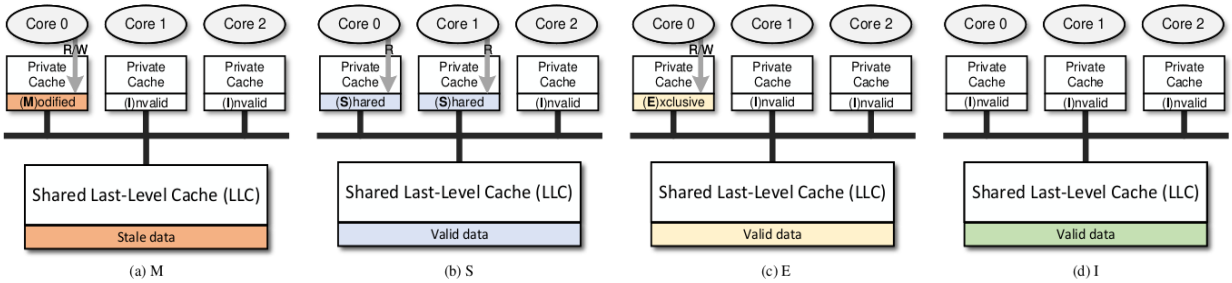
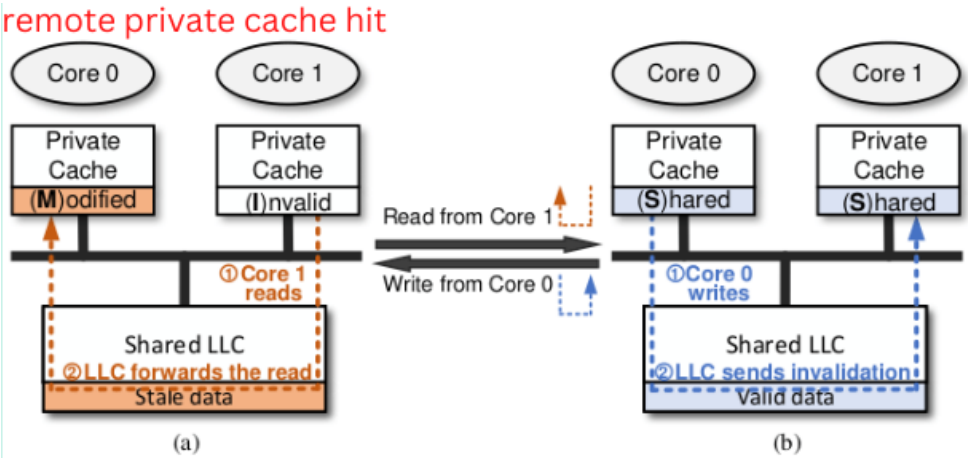


Fig. 1: The four possible states of a private cache line, when using the MESI protocol.

In the MESI protocol, there are four possible states of a private cache line:

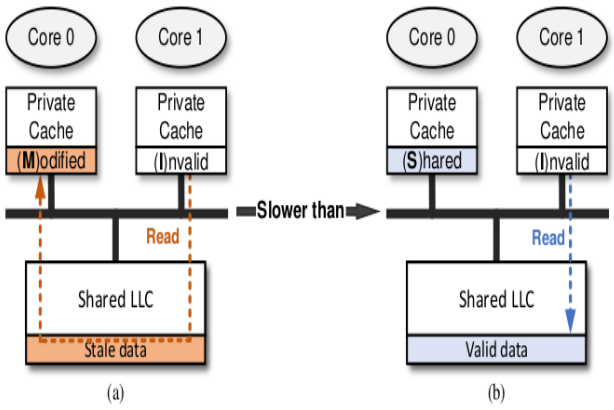
- Modified (M): The cache line is in a single private cache and is dirty(the copy of this cache line in LLC has invalid data). The current owner has read/write permissions on the cache line.
- Shared (S): The cache line is clean and present in one or more private caches. The current core has only read permission on it.
- Exclusive (E): The cache line is present in a single private cache and is clean. The current owner has read/write permissions on this cache line. When a write operation is done on this cache line, the state of this cache line is changed to M.
- Invalid (I): The cache line is invalid, and the current core has neither read nor write permission for this cache line.

State Transitions in MESI Protocol



Consider the case when core 1 reads a cache line that is present in the LLC and a private cache of core 0 in the M state. This read request first misses in core 1's private cache and then goes to LLC. Although the target cache line is present in LLC, its contents may be invalid. Thus, the LLC will get the latest data from the owner's(core 0's) private cache and update the contents of the cache line in LLC. The state of this cache line in core 0's private cache is changed to S. The updated cache line is sent to the requesting core 1, and the private cache of core 1 is filled with a copy of this cache line in the S state. This case is referred to as a remote private cache hit.

In the second case, core 0 tries to write a cache line that is present in the S state in its private cache. Core 0's private cache will request the LLC to acquire write permission on this cache line before executing the write operation. The LLC then sends invalidation signals to other private caches that contain this cache line, core 1 in our case. After this write operation, the cache line is only present in core 0's private cache in the M state.



If a core tries to read a cache line that is not in its private cache but is in LLC, the time taken will be different depending on the coherence state. If this cache line is in another core's private cache in the M state, the request results in a remote private cache hit. If the cache line is in the S state, the LLC can directly serve the read request as the data in LLC is up-to-date. The time taken by an LLC hit is much faster than the remote private cache hit case.

Prefetch

Prefetching is a technique to boost performance by fetching data before needed and placing it closer to the CPU core, e.g. From LLC to the L1 cache.
Prefetch can be performed in two ways:

1. Hardware Prefetch: Implemented in cache hardware, e.g. adjacent cache line prefetcher.
2. Software Prefetch: Explicitly done by the programmer/compiler. They hint to the processor that a memory location will likely be accessed soon. E.g. prefetch instructions in x86 processors - `PREFETCHT0`, `PREFETCHT1`, `PREFETCHT2`, `PREFETCHNTA`, `PREFETCHW`.

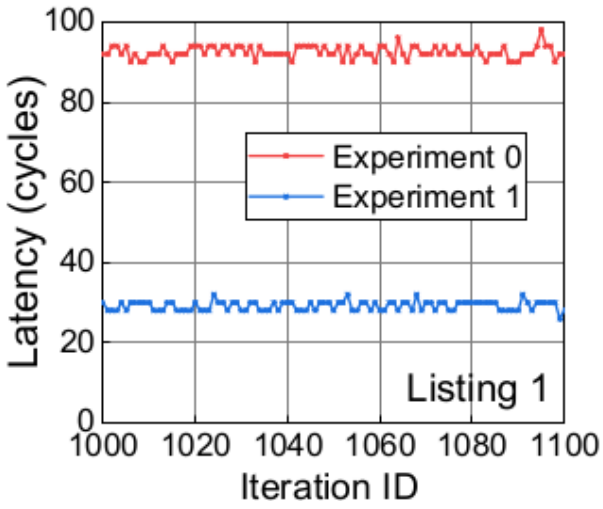
The PREFETCHW instruction aims to accelerate future writes on the target cache line. The `PREFETCHW` instruction presets the coherence state of the target cache line to M so that future writes will likely have an L1 hit.

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

```
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.

This is observed using coherence state changes of the data using timing information.
Two threads are running on different cores. mmap is used to map part of the system file as a read-only data block; both threads can only read d0. In each iteration, first, thread0 executes and waits for thread1.
Two experiments are conducted, where expt_idx is set as 0 and 1.
In experiment 0, thread0 performs PREFETCHW on d0, then thread1 loads d0 and notes down the time required to load d0. In experiment 1, thread0 stays idle and thread1 loads d0.



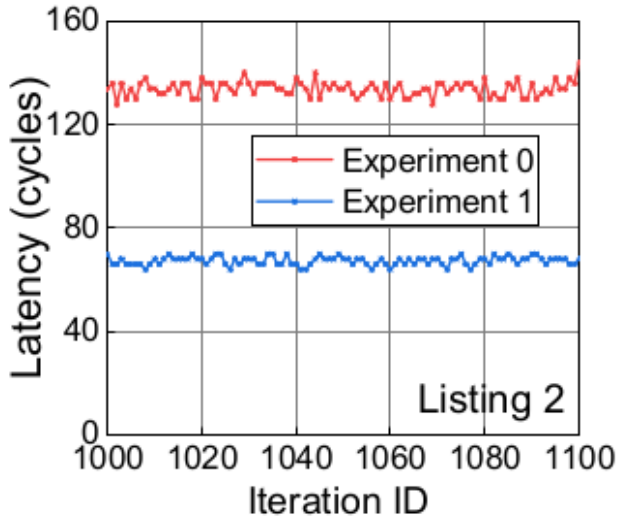
We can predict that the load time will be higher in experiment 0 as every time thread0 prefetches d0, d0 is loaded into its private cache, and its coherence state is set to M. When thread1 tries to access d0, it will have a remote secret cache hit. In experiment 1, when thread1 accesses d0, it is already present in the private cache of thread1, and it will take a lower time to load d0 as compared to experiment 0. As seen in the graph, the experimental results match our expectations.

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

```
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.

Similar to the last experiment, we have two threads running on two different cores.
Two experiments are conducted, where expt_idx is set as 0 and 1. In experiment 0, thread0 loads d0 and then thread1 performs PREFETCHW on d0 and times the prefetch. In experiment 1, thread0 remains idle and thread1 times prefetch.
We can expect that the time taken by prefetch would be more in experiment 0 than in experiment 1.



This is because in experiment 0, after thread0 loads d0, the cache line's state containing d0 becomes S. A copy of this cache line is stored in thread0's private cache. Subsequently, when thread1 prefetches it, the state needs to be changed from S to M. The LLC needs to invalidate the copy of d0 in the private cache of thread0. In experiment 1, this will not happen as d0 is already in the M state because thread0 did not load d0. As seen in the graph, the experimental results match our expectations.

Prefetch-based covert channel attacks

The sender and receiver are running on different physical cores. The sender and receiver can share read-only data and agree on predefined channel protocols like synchronisation, core allocation, data encoding and error correction protocols.

Prefetch + Load Attack

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";
```

This attack is based on observation 1.

To send bit 1, the sender executes PREFETCHW on the shared cache line. To send bit 0, the sender remains idle. The receiver loads the shared cache line and interprets the shared bit by looking at the time required.

The receiver receives a bit 1 when having a remote private cache hit and otherwise receives a bit 0.

Prefetch + Prefetch Attack

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";
```

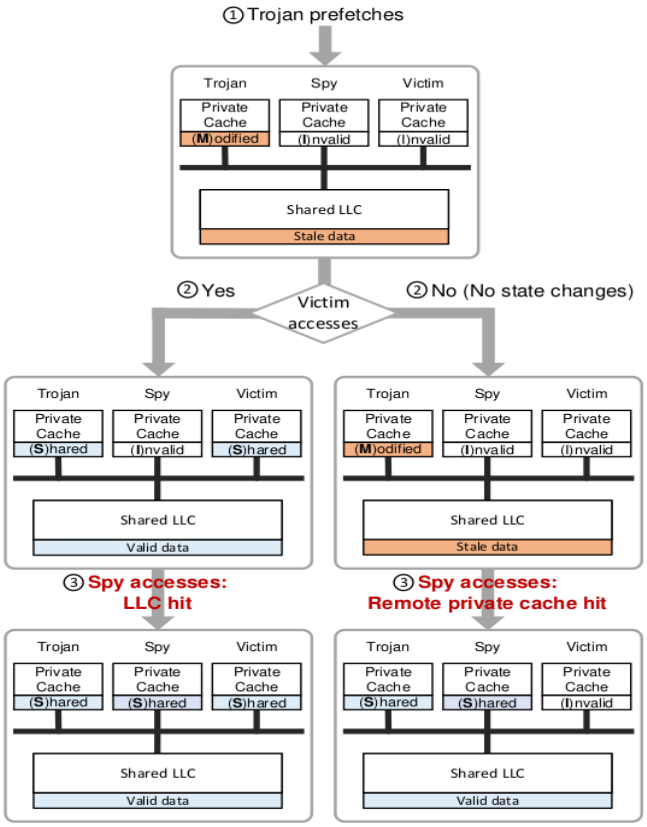
This attack is based on observation 2.

To send bit 1, the sender loads the shared cache line. To send bit 0, the sender remains idle. The receiver executes PREFETCHW on the shared cache line and interprets the shared bit by looking at the time required.

When the sender sends 1, it takes longer for the receiver to prefetch than when the sender sends 0.

Prefetch-based side channel attacks

Prefetch + Reload Attack



The attacker controls two threads, Trojan and Spy. Trojan, Spy and Victim all run on different cores.
The Trojan executes PREFETCHW on the target cache line, invalidating the copies of this cache line in the victim's and Spy's private caches (if they exist) and placing a copy of this cache line (in M state) in the Trojan's private cache. Then, the attacker waits for the victim's behaviour. 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 and is now valid. The Trojan cannot observe this state change caused by the victim's access. If the Trojan accesses this cache line, he will get a private cache hit, irrespective of whether the victim accessed this line or not, because the victim's read does not invalidate the copy in the Trojan's private cache.

However, the Spy can identify whether the victim accessed this cache line. The Trojan's original PREFETCHW had invalidated the copy of the cache line 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; otherwise, he will get a remote private cache hit.
The Spy distinguishes between these two situations by timing the access.
We can see that this attack requires that the target shared cache line is present in LLC for the Spy to get an LLC hit. This is true in the case of inclusive LLCs. In the case of non-inclusive LLCs, when PREFETCHW invalidates the copies of the target cache line in Spy and the victim's private caches, this cache line is placed in LLC. So the attack works on both inclusive and non-inclusive LLCs.

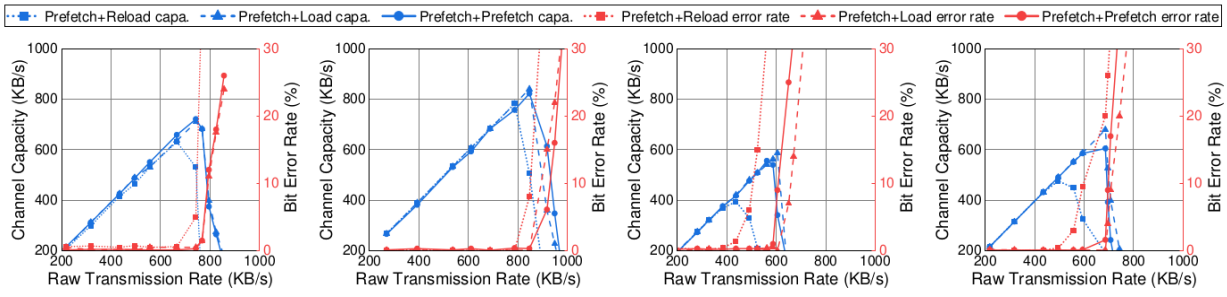
Prefetch + Prefetch Attack

Step 1: The attacker prefetches the target shared cache line using PREFETCHW and times this operation to learn whether the victim accessed this cache line in the last iteration.

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

If the victim accessed the target cache line, PREFETCHW executes slower; if the victim did not access it, PREFETCHW executes faster.

Evaluation of Prefetch-Based Covert Channel Attacks



(a) Intel Core i7-6700 (b) Intel Core i7-7700K (c) Intel Xeon Platinum 8124M (d) Intel Xeon Platinum 8151

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

Flush + Reload - 298 KB/s
Flush + Flush - 496 KB/s
Prime + Probe - 438 KB/s

One shared cache line is used between the sender and the receiver. The channel capacity metric is used to find the best transmission rate. Channel capacity is computed by multiplying the raw transmission rate with 1-H(e), where e is the bit error rate(BER) and H is the binary entropy function. The BER stays low and is almost constant when the raw transmission rate is under a threshold, beyond which it increases, causing a decrease in the channel capacity. The prefetch-based attacks are faster than almost all existing cache attacks on x86 CPUs, as shown in the above table.

Side channel attack on cryptographic code

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**
        $r \leftarrow r * b \mod n$

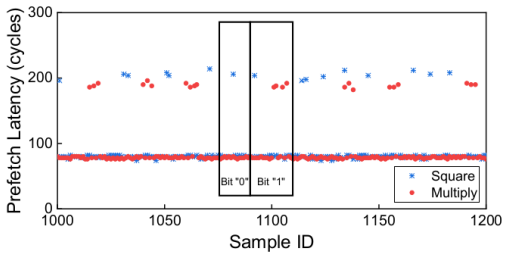


Fig. 7: A segment of the prefetch latencies measured in Prefetch+Prefetch while attacking GnuPG; part of the the exponent e shown here is "111001011001".

We can target the cryptographic libraries using this attack where the access pattern to some instructions is related to the value of the cryptographic key. The square and multiply algorithm used in RSA is targeted. mmap instruction is used to map the pages that contain sqr and mul into the attacker's address space. During the victim's execution, the cache lines corresponding to those instructions are brought into the victim's L1 instruction cache. As the instruction pages are mapped as data blocks in the attacker's address space, the same cache lines containing those instructions are brought into the attacker's L1D cache.
The graph shows the timing measurement results from the attacker for 200 samples. Lower prefetch latency indicates that the victim did not access the target cache line during the last iteration, whereas a higher prefetch latency means the victim did access. Access to sqr followed by access to mul indicates a bit 1, and two consecutive accesses to sqr indicate a bit 0.

Side channel attack on keystroke timing

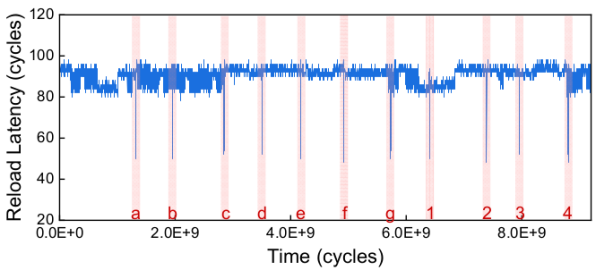


Fig. 8: The access latencies measured in Step 3 of Prefetch+Reload when a user types "abcdefg1234" in gedit; we monitor address 0x7b980 of libgdk.so.[7]

This attack leaks the precise timing information of keystrokes. Prefetch-based attacks are executed to monitor accesses to the address selected in the GDK library. When a keystroke occurs, the reload operation in Prefetch+Reload takes around 50 cycles to finish, otherwise over 80 cycles.

Prefetch-Based Channels in Transient Execution Attacks

Transient execution attacks like Meltdown and Spectre require a microarchitectural covert channel attack to transfer the secrets to the attacker. Prefetch-based attacks can work with transient execution attacks. Prefetch-based attacks can achieve higher bandwidth due to the higher channel capacity of the covert channel attack. Also, prefetch-based attacks can have more leakage in the transient window. This is because while using Spectre with Flush+Reload, the data access is slow DRAM access compared to remote private cache hits using prefetch-based attacks. This indicates that more encoding operations can be performed within the same transient window.

Paper Critique

The authors show that the prefetch-based channels have very high capacities when using a single shared cache line between the sender and the receiver among all existing CPU cache covert channels.

In private cache channel attacks, the attacker learns the victims' cache accesses by monitoring the state of the victim's data in the private cache. As private cache channel attacks do not require DRAM access, they have high bandwidth. However, they have one limitation, the attacker and the victim should run on the same core, and they require SMT. So, these attacks can be prevented by simply disabling SMT for security. LLC attacks can solve this problem as the LLC is shared among different CPU cores. The attacker can monitor the state of the victim's data in the LLC. However, DRAM accesses are involved in LLC attacks which can bottleneck the bandwidth.

The proposed cross-core prefetch-based attacks in this paper use the private cache, and the target cache line is always kept in the on-chip hierarchy, preventing DRAM accesses.

Thus these attacks have higher bandwidth since the cache accesses are much faster than the DRAM accesses. Also, the proposed attacks work irrespective of the LLC inclusivity.

Most of the recent Intel processors have the PREFETCHW instruction, which, if used correctly, can significantly improve performance. The following modifications in implementing PREFETCHW can avoid the attacks proposed in the paper:

1. PREFETCHW should perform write permission checks like a regular memory write instruction before changing the state of the cache line. The instruction can be ignored if the requesting core does not have written permission on the target cache line.
2. PREFETCHW should execute in constant time irrespective of the state of the target cache line.

Also, as this attack works by manipulating and monitoring the cache state, defences to the prior cache attacks work against prefetch-based attacks. The prefetch instruction does not fetch any data if the buffer between the L1 and L2 cache is full. Thus, if a memory-intensive process runs on the same core as the attacker, the performance of this attack can reduce drastically.