计算机体系结构安全
缓存侧信道攻击 2

汇报人：李朋阳

导师：尹捷明 教授

2025. 4. 2

Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks

Yanan Guo[1], 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
yag45@pitt.edu, zhangyt@cs.pitt.edu, juy9@pitt.edu

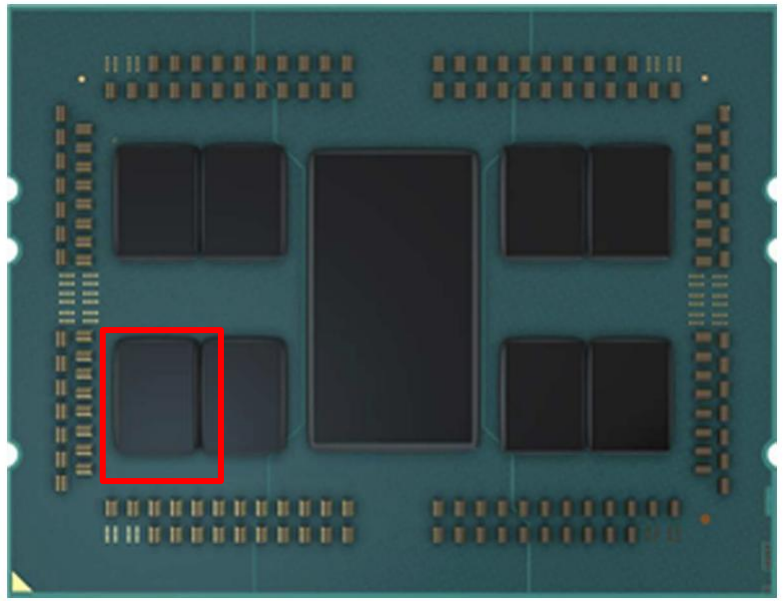
Y. Guo, A. Zigerelli, Y. Zhang and J. Yang, "Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks," *2022 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2022, pp. 1458-1473, doi: 10.1109/SP46214.2022.9833692.

Some Tips

1. Multi-Core Processor
2. Cache
3. Cache Coherence Protocol
4. Prefetch

Some Tips

1. Multi-Core Processor
2. Cache
3. Cache Coherence Protocol
4. Prefetch



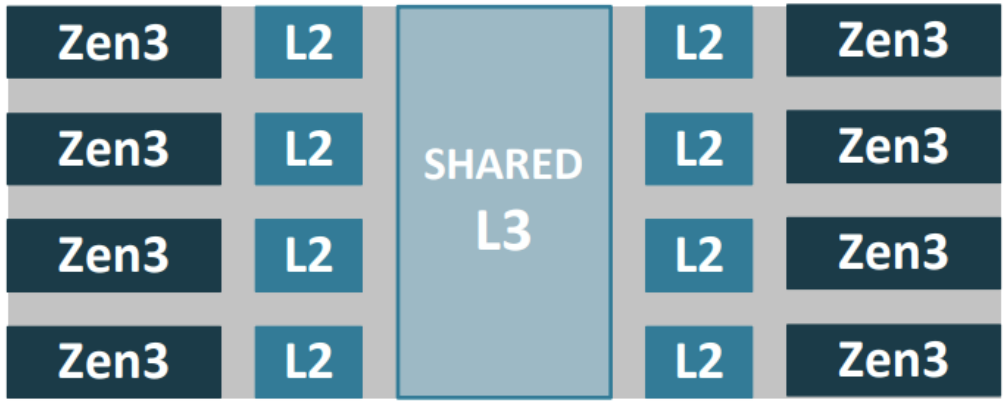
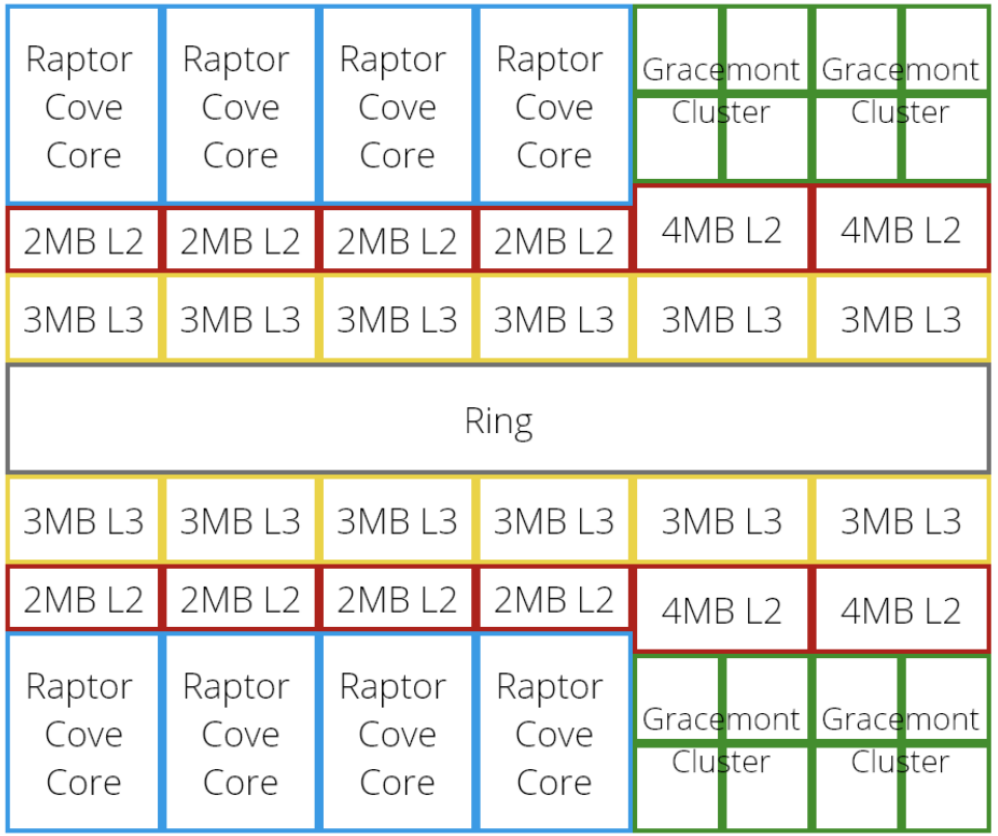


Figure 2-2: Eight Compute Cores sharing an L3 cache within a single Core Complex Die (CCD)

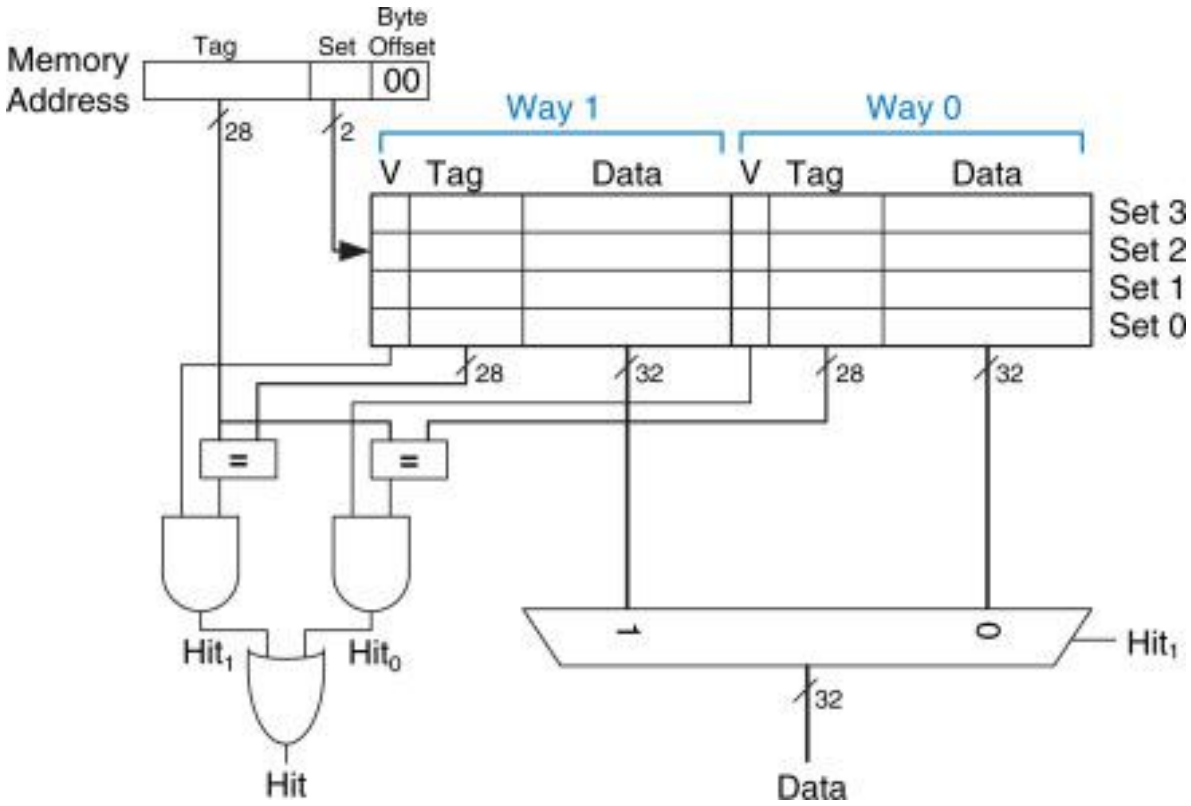
Some Tips

1. Multi-Core Processor
2. Cache
3. Cache Coherence Protocol
4. Prefetch



Some Tips

1. Multi-Core Processor
2. Cache
3. Cache Coherence Protocol
4. Prefetch



Some Tips

1. Multi-Core Processor
2. Cache
3. Cache Coherence Protocol

- **Modified（M）：** 缓存行中的数据被修改，与主内存不一致。此时数据只存在于当前缓存中，其他缓存中没有该数据。
- **Shared（S）：** 缓存行中的数据未被修改，与主内存一致，并且可能存在于其他缓存中。

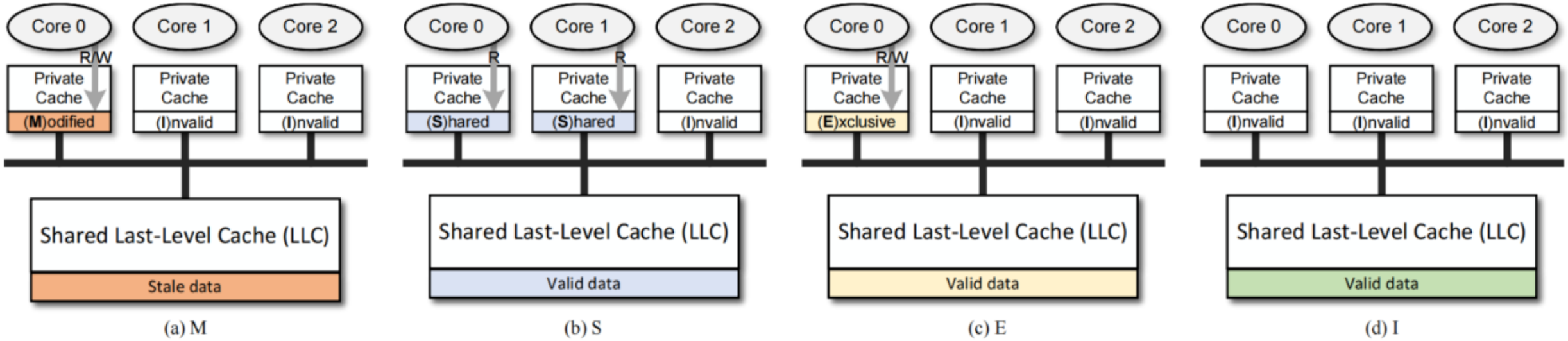


Fig. 1: The four possible states of a private cache line, when using the MESI protocol.

Some Tips

1. Multi-Core Processor
2. Cache
3. Cache Coherence Protocol

- **Exclusive（E）：** 缓存行中的数据未被修改，与主内存一致，且数据只存在于当前缓存中。
- **Invalid（I）：** 缓存行中的数据无效，可能是由于其他缓存修改了该数据并广播了失效消息。

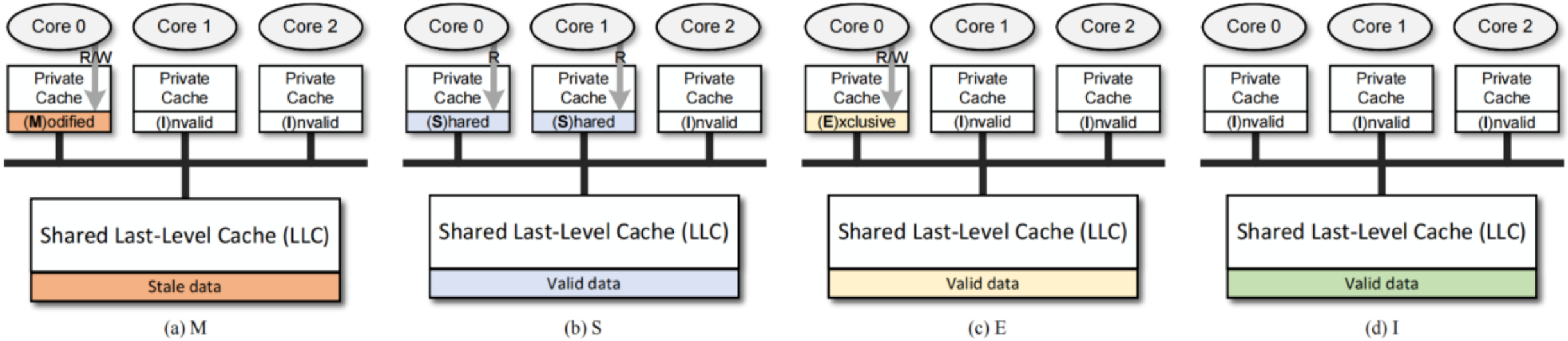


Fig. 1: The four possible states of a private cache line, when using the MESI protocol.

Some Tips

1. Multi-Core Processor
2. Cache
3. Cache Coherence Protocol
4. Prefetch

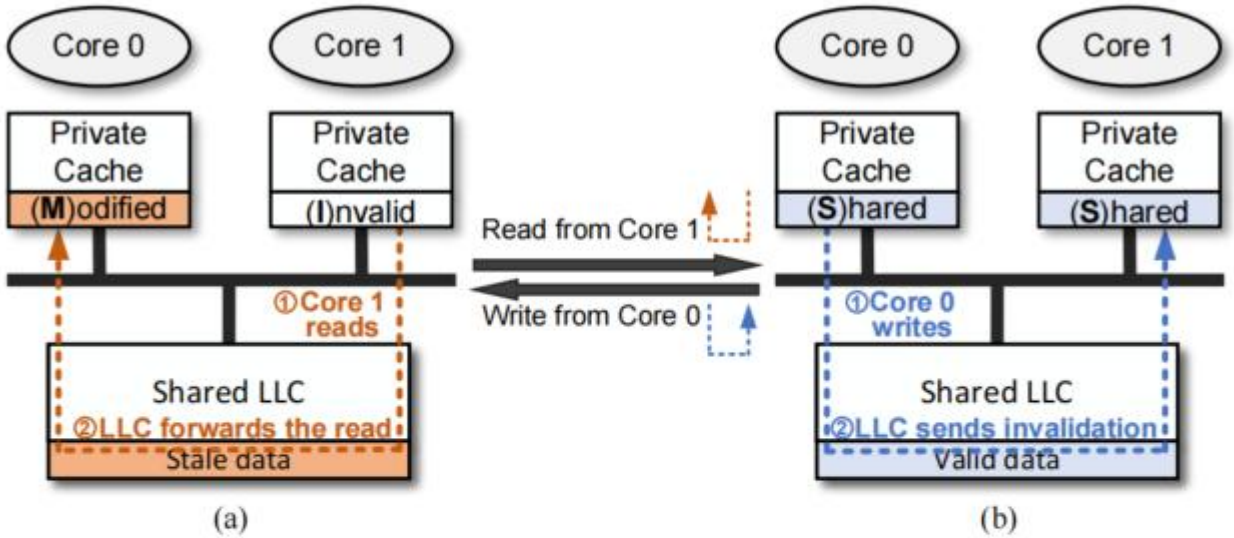


Fig. 2: The illustration of cache coherence state changes. The state of a line changes from M (shown in (a)) to S (shown in (b)) when a CPU core is loading it; conversely, the state changes from S to M when a CPU core is writing it. Dashed lines shows the request path of the read/write operation.

Some Tips

1. Multi-Core Processor
2. Cache
3. Cache Coherence Protocol
4. Prefetch

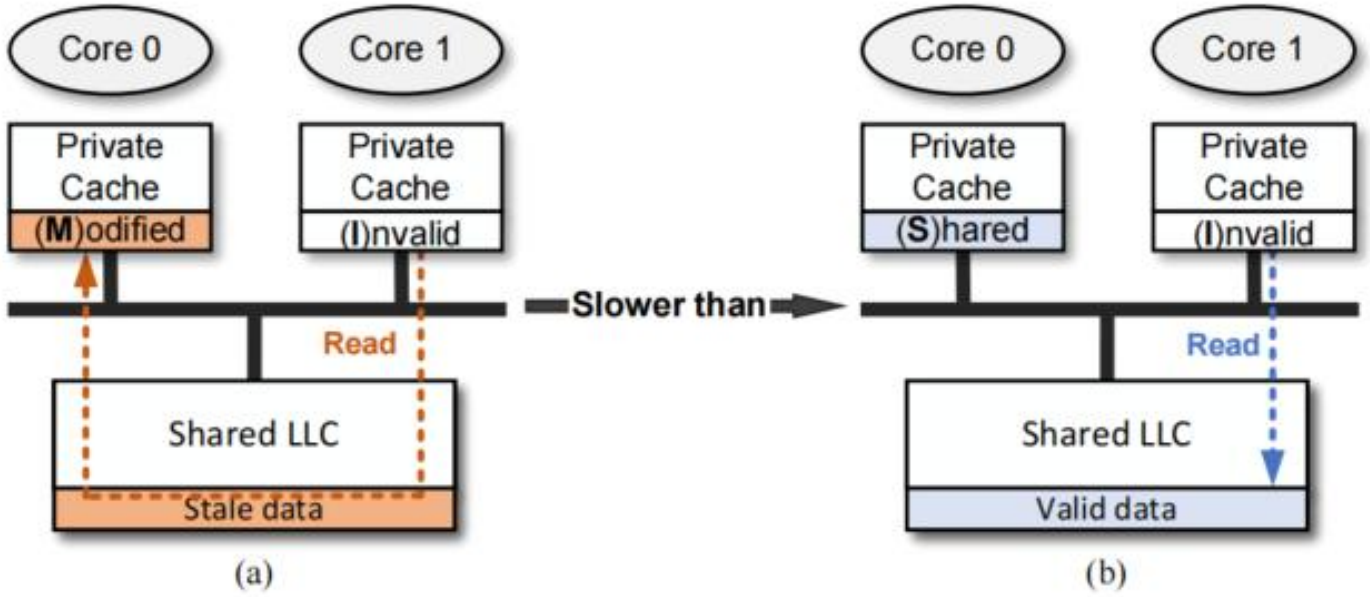
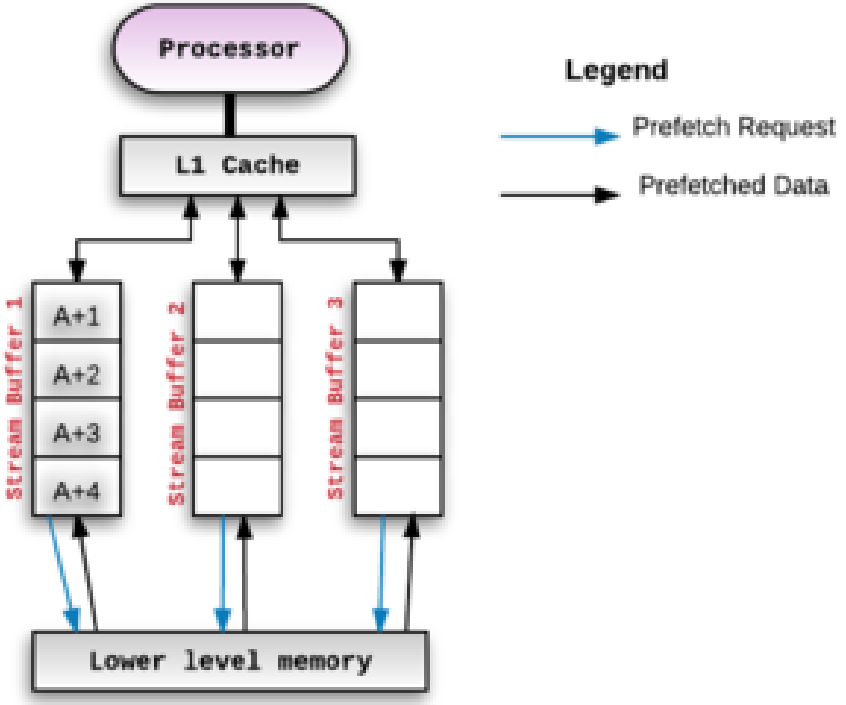


Fig. 3: The illustration of an LLC access with the target cache line in M state (a), and S state (b).

Some Tips

1. Multi-Core Processor
2. Cache
3. Cache Coherence Protocol
4. Prefetch



- **硬件预取**：处理器内部有一个专门的硬件机制，它会监视正在执行的程序所请求的指令或数据流。基于这个流，它能够识别出程序接下来可能需要的几个元素，并将它们预取到处理器的缓存中。

- **软件预取**：程序员手动/编译器在编译程序的过程中在代码中插入的“预取”指令，如：
  $PREFETCHT0, PREFETCHT1, PREFETCHT2, PREFETCHNTA$

攻击基础

- **PREFETCHW** 指令将指定的内存数据预取到处理器的缓存中，并使其他缓存中的副本失效。这种预取操作通常用于优化写操作的性能。

- 当数据需要被写入时，PREFETCHW 可以提前将数据加载到缓存中，并使其处于"M"状态，从而减少后续写操作的延迟。
- 对于竞争缓存行的情况，PREFETCHW 可以避免在写入时需要额外的缓存状态转换。

- 如果后续有写操作，PREFETCHW 可以加速写入，因为缓存行已经处于“M”状态。

PREFETCHW是2000年引入的x86预取指令。它支持所有英特尔至强处理器和最近的核心处理器(since Broadwell）。指令的功能是为将来的写入准备数据。

攻击基础

- **PREFETCHW successfully executes on data with read-only permission.**
- 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              /* 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     ...
```

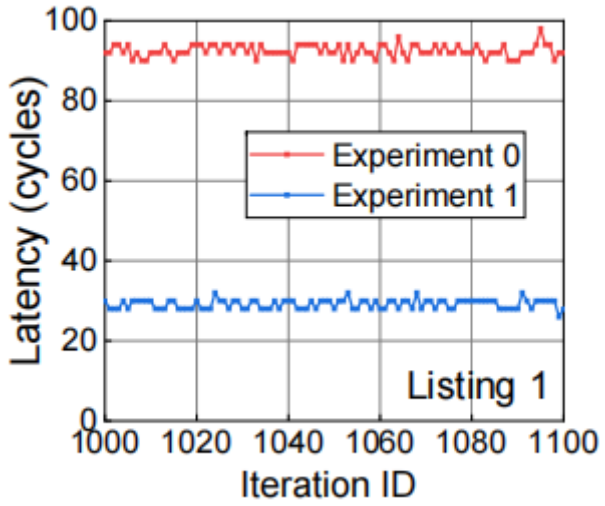
Listing 1: The code snippet for verifying Observation 1.

1.实验0：在for循环的每次迭代中，thread0执行PREFETCHW指令预取数据d0，然后thread1加载d0并对这次加载。

2.实验1：在for循环的每次迭代中，thread0保持空闲状态，而thread1加载d0。

攻击基础

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

2. CPU缓存侧信道攻击

攻击基础

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

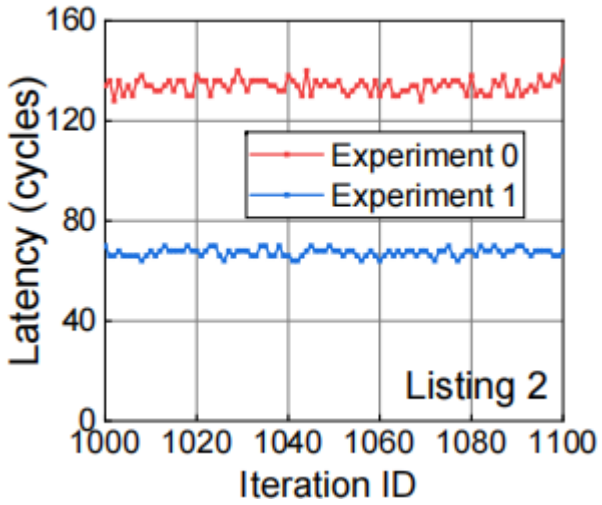
Listing 2: The code snippet for verifying Observation 2.

1. **实验0**：在for循环的每次迭代中，thread0首先加载数据d0，然后thread1对d0执行PREFETCHW指令。
2. **实验1**：在for循环的每次迭代中，thread0保持空闲状态，而thread1在每次迭代中仍然执行PREFETCHW指令。

2. CPU缓存侧信道攻击

攻击基础

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

2. CPU缓存侧信道攻击

Convert Channel攻击

Prefetch + Load Attack：

1.共享缓存行：发送者和接收者首先协商确定一个共享的缓存行（line0），用于传输信息。

2.发送者传输比特：

1. 如果要传输的比特是 "1"，发送者执行 PREFETCHW 指令预取共享缓存行，这将导致接收者的缓存行被驱逐出缓存。
2. 如果要传输的比特是 "0"，发送者不执行任何操作。

3.接收者检测比特：

1. 接收者加载缓存行，并测量加载时间。
2. 如果加载时间较长，表示缓存行被驱逐，接收者接收到 "1"。
3. 如果加载时间较短，表示缓存行仍在本地缓存中，接收者接收到 "0"。

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";
```

2. CPU缓存侧信道攻击

Convert Channel攻击

Prefetch + Prefetch Attack：

1.发送者传输比特：

1. 如果要传输的比特是 "1"，发送者Load共享缓存行，这将导致接收者的缓存行状态改变。
2. 如果要传输的比特是 "0"，发送者不执行任何操作。

2.接收者检测比特：

1. 接收者PREFETCHW缓存行，并测量时间。
2. 如果加载时间较长，表示缓存行状态改变，接收者接收到 "1"。
3. 如果加载时间较短，表示缓存行状态未改变，接收者接收到 "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";
```

2. CPU缓存侧信道攻击

Convert Channel攻击

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

2. CPU缓存侧信道攻击

Side Channel攻击

Prefetch+Reload Attack：

- **步骤1**：在victim访问目标共享缓存行之前，Trojan执行PREFETCHW指令，这会使得victim和Spy的私有缓存中该缓存行的副本失效（如果存在的话），并将该缓存行的副本（以M状态）放入Trojan的私有缓存中。
- **步骤2**：如果victim访问这个缓存行，根据MESI协议，一致性状态会从M变为S，LLC中的该缓存行副本会被更新。
- **步骤3**：Spy通过访问这个缓存行并计时来确定是远程私有缓存命中还是LLC命中。如果是远程私有缓存命中，说明victim没有访问这个缓存行；否则，victim访问了。

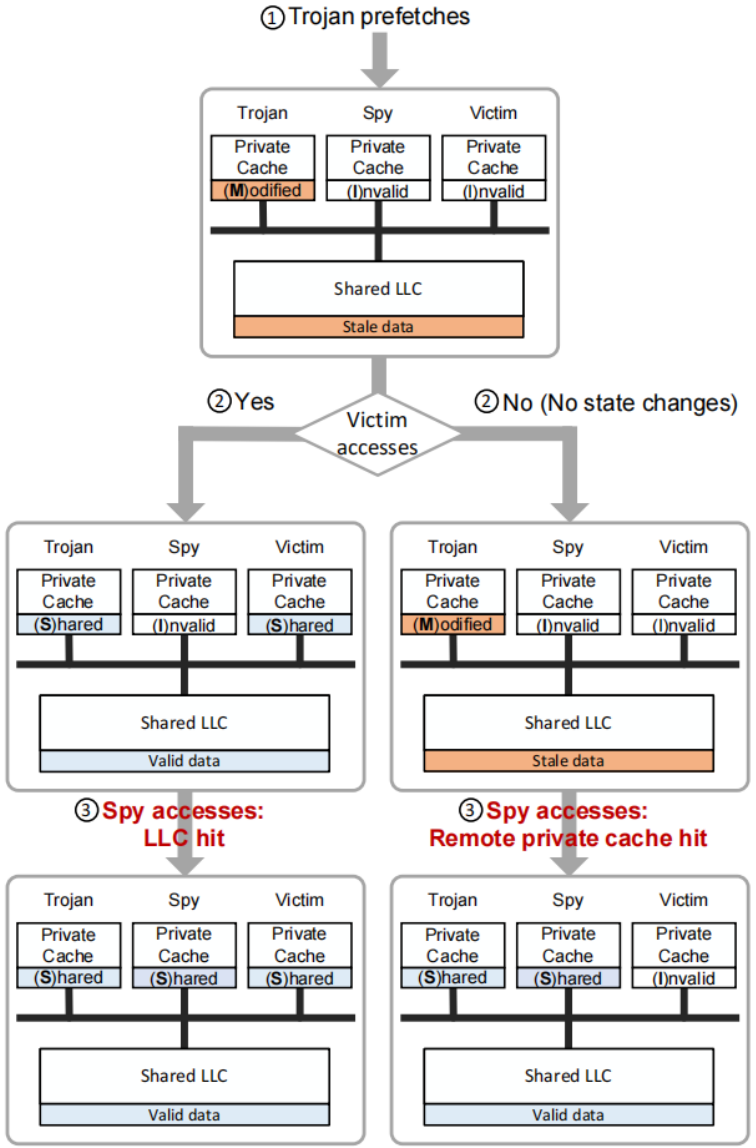


Fig. 5: The details of the three steps in Prefetch+Reload.

2. CPU缓存侧信道攻击

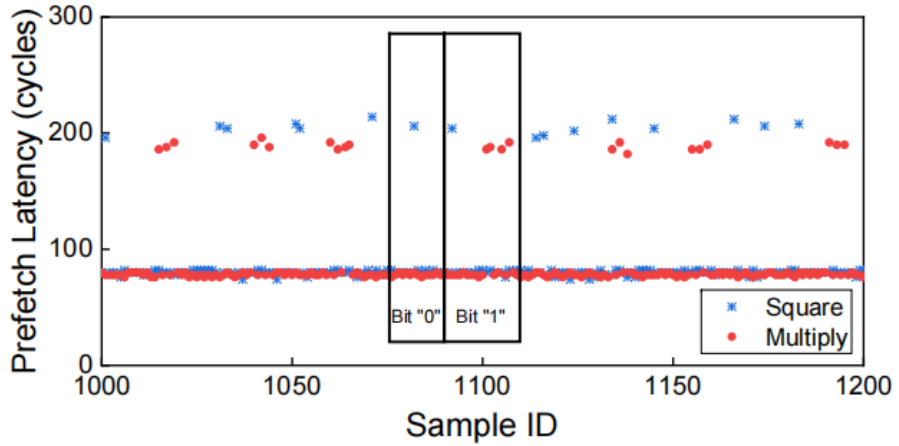


Fig. 7: A segment of the prefetch latencies measured in Prefetch+Prefetch while attacking GnuPG; part of the the exponent e shown here is "111001011001".

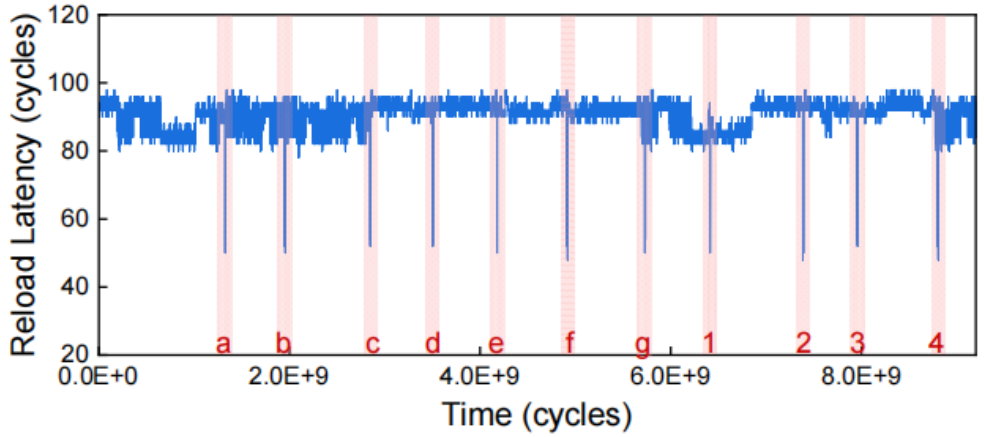


Fig. 8: The access latencies measured in Step 3 of Prefetch+Reload when a user types "abcdefg1234" in gedit; we monitor address 0x7b980 of libgdk.so.[7]

Thanks