2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

PREFETCHX: Cross-Core Cache-Agnostic Prefetcher-Based Side-Channel Attacks

Yun Chen, Ali Hajiabadi, Lingfeng Pei, and Trevor E. Carlson
School of Computing, National University of Singapore
{yun.chen, ali.hajiabadi}@u.nus.edu, lfpei@nus.edu.sg, tcarlson@comp.nus.edu.sg

***Abstract*—In this paper, we reveal the existence of a new class of prefetcher, the XPT prefetcher, in modern Intel processors which has never been officially detailed. It speculatively issues a load, bypassing last-level cache (LLC) lookups, when it predicts that a load request will result in an LLC miss. We demonstrate that XPT prefetcher is shared among different cores, which enables an attacker to build cross-core side-channel and covert-channel attacks. We propose PREFETCHX, a cross-core attack mechanism, to leak users' sensitive data and activities.**

We empirically demonstrate that PREFETCHX can be used to extract private keys of real-world RSA applications. Furthermore, we show that PREFETCHX can enable side-channel attacks that can monitor keystrokes and network traffic patterns of users. Our two cross-core covert-channel attacks also see a low error rate and a 122 KiB/s maximum channel capacity. Due to the cache-independent feature of PREFETCHX, current cache-based mitigations are not effective against our attacks. Overall, our work uncovers a significant vulnerability in the XPT prefetcher, which can be exploited to compromise the confidentiality of sensitive information in both cryptography and non-cryptography-related applications among processor cores.

I. INTRODUCTION

Decades of research in designing efficient and modern processors has resulted in various performance enhancements at the microarchitectural level, like out-of-order execution, speculative execution, multi-core processing, caching, and prefetching. However, in recent years there has been rapid discovery of security vulnerabilities arising from these performance enhancement techniques [7], [9], [10], [13], [16], [27], [30], [35], [44], [46]. These vulnerabilities are exploited to either infer a user's private data and secret keys (in case of side-channel attacks) or to stealthily transfer data in the system (in the case of covert-channel attacks).

One of the prominent sources of information leakage in modern processors are prefetchers since they can leave a footprint of the data accessed by the victims, similar to caches. The goal of prefetching is to bring the data closer to the core if it has high confidence to be needed in the near future. This technique can be implemented on either hardware [4], [22] or software [20]. Recent attacks [7], [8], [12], [15], [18], [19], [37], [42] exploit a variety of prefetching mechanisms to leave persistent malicious and secret-dependent changes in the system that can be later inferred by the attacker. In this paper, we explore a prefetcher in Intel processors, called eXtended Prediction Table (XPT), that is located in parallel to the LLC. The XPT prefetcher is not documented in the latest Intel official architecture manual [26] but only has a brief functionality description in the HPC optimization manual [25].

In this paper, we first reverse-engineer the XPT prefetcher and reveal its prefetching mechanism. We investigate the interaction of different cores through the prefetcher. Our analysis has resulted in the first side-channel using the XPT prefetcher, called PREFETCHX, which can be exploited in 3^{rd} generation Xeon processors[1] to effectively monitor the victim's page activities. This is achieved by deliberately mistraining the XPT prefetcher on specific pages and subsequently examining the prefetcher's status. Consequently, the attacker can reconstruct the victim's sensitive information, like cryptographic keys.

Unlike many standard threat models that necessitate the victim and attacker to share the same physical core [7]–[9], [13], [27], [35], [37], [42], [48], PREFETCHX enables *cross-core* side-channel attacks. This substantially broadens the scope of potential targets and amplifies the potential security impact. In addition, the PREFETCHX attacks do not rely on the cache subsystem; we do not rely on the caches as a source of leakage nor as a primitive to check the prefetcher's status. To the best of our knowledge, we are the first to identify, explore, and reverse-engineer the XPT prefetcher. This discovery highlights the existence of hidden channels that could expose security risks in processor architectures.

Our key contributions in this work are as follows:

- We uncover a briefly-mentioned prefetcher, named the XPT prefetcher, in the 3^{rd} generation of Intel server processors. We reverse-engineer this prefetcher, and provide a detailed characterization of its features and behaviors, which might be helpful for future security and performance research (Section IV).
- We construct four end-to-end cross-core side-channel attacks using the XPT prefetcher as the leaky source. These attacks include an RSA attack (the square-and-multiply RSA used in the latest MbedTLS [3] and GnuPG 1.4 [14]), a keystroke attack, and a network traffic monitoring attack. Our results demonstrate the practicality and effectiveness of PREFETCHX in real-world scenarios[2] (Section VI).
- We develop two cross-core covert-channel attacks with low error rates, further demonstrating the high applica-

[1]The latest Xeon processors at the time of paper submission.

[2]We have responsibly disclosed our findings to the Intel PSIRT team and have received approval to distribute these details.

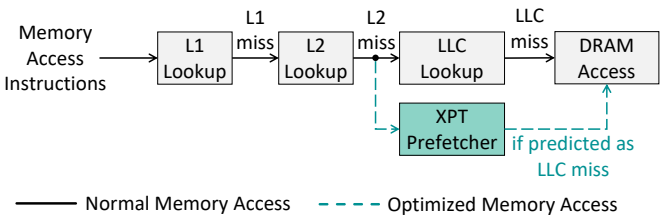

Figure 1. Overview of the XPT prefetcher operation.

bility and potential security implications of PREFETCHX (Section VII).

II. PREFETCHX MOTIVATION AND OVERVIEW

A. The XPT Prefetcher as a Leakage Source

PREFETCHX exploits the XPT prefetcher in Intel processors, and the root cause of the attack is the special mechanism of this component. The XPT prefetcher resides in parallel with the LLC and its main goal is to predict LLC misses and to send speculative requests to the DRAM to bypass LLC lookups. This can reduce memory access latency in the case of a correct prediction. Hence, the memory accesses sent to the LLC subsystem exhibit three levels of access latency (we use the `RDTSC` to compute the latency): (1) an *LLC hit* (less than 160 cycles in our setup and experiments), (2) *LLC miss* (350+ cycles), and finally (3) *optimized LLC miss* (170-330 cycles). The optimized LLC misses occur in cases when the XPT prefetcher has a correct prediction and is enabled. Figure 1 shows an overview of the XPT prefetcher alongside the cache hierarchy. This timing variation is tightly coupled with the memory activities of the executing program that enables easy status monitoring of the prefetcher.

In addition, our detailed experiments in Section IV reveal that (1) XPT prefetching can be trained and triggered *across different cores*, that (2) the LLC miss prediction is based on a miss counter in the prefetcher for a 4 KiB page granularity, and that (3) different entries in the prefetcher are indexed based on the physical address of the data and distinct by Address Space ID (ASID) and CoreID[3].

In this work, we make two key observations from the XPT prefetcher behavior that enables us to build side-channel and covert-channel attacks. First, ***we observe that the XPT prefetcher entries are shared across different cores and are evicted based on the Least Recently Used (LRU) policy if an XPT-uncached page in different ASIDs is accessed***. If Process 1 primes all entries in the XPT, Process 2 will evict the oldest entry touched by Process 1 when it accesses the physical page (i.e., LLC or memory access) that is not presented in the XPT. Second, ***we observe that flushing the TLB often resets the XPT prefetcher's entry status***. We demonstrate that if Process 1 trains the XPT prefetcher with a shared page A and Process 1 then flushes that page mapping from its TLB, then the XPT prefetcher will not be triggered for the previously trained page A for any process. In Section IV-F, we provide the details of our observations.

[3]Note, each process has a unique ASID, whereas different threads in the same process share an ASID but can run on different cores that have different CoreIDs

Table I
INTEL HARDWARE PREFETCHERS [11].

Intel Prefetcher	Location	Pattern	Attacks
Data Cache Unit (DCU)	L1-D	Next cache line [22]	-
Instruction pointer (IP)-stride	L1-D	Load instructions with regular stride [7]	[7], [8] [37]
Data prefetch logic (DPL)	L2	128-bytes-aligned [22] pair cache line	-
Streamer	L2	Several cache lines backward or forward [36]	[36]
XPT prefetcher	**LLC**	**LLC miss predictor**	**This work**

B. PREFETCHX Threat Model and Attack Surface

In this paper, we assume that a victim process contains secret information (e.g., crypto key, user private information). The attacker process runs on another physical core and tries to infer the secret information without accessing the victim's address space directly (e.g., running on the unprivileged level).[4]

Based on our key observations, we develop two attack primitives: (1) PREFETCHX-Evict and (2) PREFETCHX-Flush. Launching side-channel or covert-channel attacks via PREFETCHX-Evict primitive does not require any shared resources (e.g., shared memory). Regarding the PREFETCHX-Flush primitive, which is tailored for higher-throughput and lower-noise cross-core covert-channel compared to PREFETCHX-Evict based channel, we assume that the two parties hold shared memory, similar to prevalent hardware attacking scenarios [16], [19], [42], [46]. We will introduce the workflow of these two attack primitives in Section V.

As we leverage `RDTSC` to check the XPT prefetcher state variation, which is running with a constant frequency and does not change with the CPU frequency [23], we make no assumptions on the CPU frequency of the targeted cores.

III. BACKGROUND

A. Prefetching

The main goal of a prefetcher is to predict the data and instructions that will be required by the processor in the near future and to fetch them from the main memory into the cache in advance, improving system performance. There exist a variety of prefetching styles. Software prefetching allows programmers to insert prefetching instructions in desired locations of the code [15], [18], [19], [23]. Hardware prefetchers, such as the next-line prefetcher [22], the stride prefetcher [7], spatial and temporal prefetchers [5], [6], [36], [40], each have different strategies for predicting and fetching data and can be beneficial based on the application running in the hardware.

According to Intel's whitepaper [11], their processor designs feature four hardware prefetchers, which are listed in Table I.

[4]The interplay with Intel Software Guard Extension (SGX) is considered to be outside the scope of this work as SGX was not designed to be secure against side-channel attacks [24] and is not supported in AWS (our test platform).

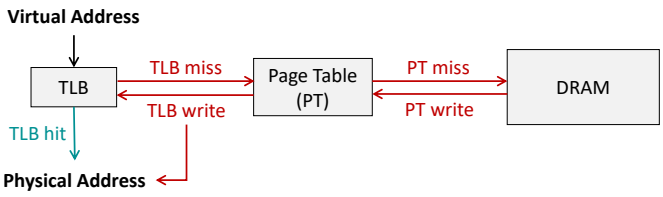

Figure 2. Process of a page walking.

In this work, we focus on the eXtension Prediction Table (XPT) prefetcher, which is not documented by Intel in any of their hardware reference manuals [23], [26], but instead was briefly described in the HPC optimization reference manual [25] for the 3^{rd} generation of Xeon processors. The key difference of the XPT prefetcher with others is that it is placed at the last-level cache (LLC) (see Table I) which makes it a potential channel for cross-core attacks.

B. Translation Lookaside Buffer and Page Tables

Translation and page walking. Figure 2 shows the virtual address to physical address translation process in x86 processors. The memory management unit (MMU) has a cache, called translation lookaside buffer (TLB), that keeps the most recently accessed page mappings for a fast translation in case of a TLB hit. In case of a TLB miss, the MMU performs a page walk over the page tables (PT) to find the mapping. The operating system (OS) is responsible for maintaining the mappings from the virtual addresses provided by each process to the physical addresses in dynamic random-access memory (DRAM). As these mappings typically have a granularity of 4 KiB, the page walking process involves primarily the upper 48 bits of the virtual address, referred to as the *page offset*. Meanwhile, the lower 12 bits remain identical to those of the physical address and are called the *in-page offset*.

TLB flush. In modern processors, the TLB is flushed if the CPU needs to update the TLB [23]. However, there are two main events that require a TLB flush:

- **Context Switch:** When the OS switches from one process to another, the TLB needs to be updated with the new virtual-to-physical address mappings for the new process[5].
- **Page Table Changes:** If the page tables are updated, such as when a page is swapped in or out of memory, the TLB will be flushed to ensure the correct page mappings are used.

TLB shootdown. When a core changes the property of a shared page (e.g., virtual-to-physical mapping, read-write permissions, etc.), the OS launches a TLB shootdown event to inform other cores to invalidate that mapping in their TLB.

[5]It is worth noting that frequent TLB flushes during context switches are avoided in modern processors as every TLB entry is tagged with an ASID, ensuring minimal impact on process performance.

```
1  void train(char *ptr) {
2      int n = 32;
3      int random_num[n];
4      generate_random(random_num);
5      for (int i = 0; i < n; i++) {
6          int index = random_num[i];
7          char junk = ptr[index * CACHE_LINE];
8      }
9  }
10 void test(char *ptr) {
11     int test_index = rand() % 10 + 53;
12     int start = rdtsc();
13     asm volatile("mfence");
14     char junk = ptr[test_index * CACHE_LINE];
15     asm volatile("lfence");
16     int diff = rdtsc() - start;
17 }
18 void main() {
19     int page_size = 4096;
20     char *ptr = (char *)mmap(page_size, ...,
       MAP_LOCK, ...);
21     for (int i = 0; i < page_size; i+=64)
22       ptr[i] = 'x';
23     flushall(ptr);
24     train(ptr);
25     test(ptr);
26 }
```

Listing 1. Determining how to trigger the XPT prefetcher. The results shown in Figure 3.

We will later show that TLB flush/shootdown impacts the XPT prefetcher status and is the root cause of a number of our cross-core covert-channel attacks.

IV. Reverse-Engineering the XPT Prefetcher

We perform extensive microbenchmarking to better understand the XPT prefetcher. In this paper, we use an AWS EC2 m6i.4xlarge instance powered by a 16-core Intel(R) Xeon(R) Platinum 8375C CPU (Ice Lake generation, Sunny Cove microarchitecture).

A. Triggering the XPT Prefetcher

Based on our experiments, the XPT prefetcher is enabled after a fixed number of LLC cache misses. To determine the number of LLC cache misses required to trigger the XPT prefetcher, we use a microbenchmark outlined in Listing 1. The microbenchmark first initializes a page (line 20) and flushes the initialized data from the cache using `clflush` instructions (line 23). We then train the XPT prefetcher up to a specific number of LLC misses, and finally, test the memory access latency for an LLC cache miss (*i.e.*, DRAM access) to test if XPT is enabled. The parameter `n` in the `train()` function specifies the number of cache misses to be generated, which is achieved through the use of the `generate_random()` function that generates `n` unique random numbers as indices for accessing the page. This ensures that the memory accesses are irregular and will not trigger other potential prefetchers, such as the next-line, IP-stride, adjacent, and streamer prefetchers. After generating `n` cache misses, a `test_index` is set, computed at runtime and distant from `n`, to test the DRAM latency of the access at `ptr[test_index]`.

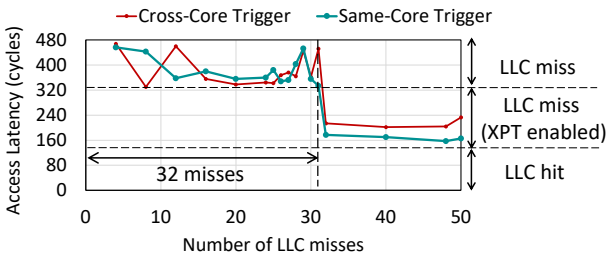

Figure 3. Triggering the XPT prefetcher from either the same core or a different core. In both cases, the XPT prefetcher is triggered after 32 LLC misses.

```
void main() {
    int page_size = 4096;
    char *ptr = (char *)mmap(page_size, ...,
    MAP_LOCK, ...);
    pid_t sub_process = fork();
    if (sub_process > 0) // main process
        train(ptr);
    else if (sub_process == 0) // sub-process
        test(ptr);
}
```

Listing 2. Determining how the XPT prefetcher table is indexed. We use semaphore to control the train/test sequence

The results of our experiments are presented in Figure 3 (green line) which runs our microbenchmark with an increasing number of LLC misses during training. One can see that the XPT prefetcher begins to start requesting data early after 32 cache misses, leading to a reduction in DRAM access latency from approximately 350+ cycles to an optimized LLC cache miss (between 160 to 300 cycles) which has triggered the XPT prefetcher.

As the XPT is located in parallel to the LLC, it is reasonable to assume that it is shared between two different cores. To verify this assumption, we modify the Listing 1 to run the `train` and `test` functions on different cores. The result is presented in Figure 3 (red line), which is similar to our observation from triggering the XPT from the same core (green line). This implies that *the XPT prefetcher is shared between different cores.*

B. Indexing into the XPT Prefetcher

Intel commonly employs the instruction pointer (IP) [7], virtual address [26], or physical address [10] as an index for accessing hardware tables or caches in its processors. Our objective is to examine and understand the indexing mechanism employed by the XPT prefetcher. Given its position as a bypass for LLC lookup and its location between the LLC and L2 cache, our initial focus is to determine if the XPT utilizes physical address indexing, similar to the LLC [10], [26].

We developed a microbenchmark (See Listing 2) to evaluate the indexing policy of the XPT prefetcher. Our microbenchmark involves allocating a shared memory page and a child process. The `train` and `test` functions, as shown in Listing 1, are reused in this experiment. The main process trains the XPT prefetcher on the shared page and the child process accesses an uncached data block in the shared page to determine if the XPT prefetcher has been triggered.

Table II
THE XPT PREFETCHER TRIGGERING RESULTS WITH DIFFERENT INDEXING POLICIES.

Tested Scenario	Indexing Policy	Is XPT triggered?
Scenario-1	Physical Page	✓
Scenario-2	Virtual Page	✗
Scenario-3	Instruction Pointer	✗

```
void main() {
    char *huge_1GiB_page = (char *)mmap(1 << 30,
                        ..., MAP_HUGE_1GB|MAP_LOCK);
    char *huge_2MiB_page = (char *)mmap(1 << 21,
                        ..., MAP_HUGE_2MB|MAP_LOCK);
    char *normal_4KiB_page = (char *)mmap(8192,
                                    ..., MAP_LOCK);
    char *ptr; //refers to one of pools above
    train(ptr);
    test(ptr[4096]); //cross 4KiB boundary
}
```

Listing 3. Determine the page boundary of the XPT prefetcher.

A shared page has a unique mapping in the virtual address spaces of both the main process and the child process (in Listing 2), because the main process and child process have distinct address spaces. However, the physical page mapping is the same for both processes. The results of our experiment, as shown in the first row of Table II (Scenario-1), indicate that the XPT prefetcher is consistently triggered when two processes access the same physical page, suggesting that *the XPT is indexed by the physical page*. We also tested the possibility of indexing by the virtual address and IP (Scenario-2 and Scenario-3 Table II, respectively). For instance, we use the same PC to generate 32 LLC misses on different physical pages and use the same PC to test the latency. The experiments wouldn't have failed if the tags were determined by the PC. but the results indicate that these methods do not trigger the XPT prefetcher and are not used for indexing the XPT prefetcher.

To gain a deeper understanding of how many bits of the physical address are utilized for indexing the XPT, we create a 32 GiB memory pool so that we can manipulate the least significant 35 bits of physical addresses. We then identify two physical pages with identical least significant M bits for their page offsets (*i.e.*, physical addresses from 13^{th} bit to the $(13+M)^{th}$ bit are the same). By training the XPT prefetcher on one page, we expect that if the table is indexed by the lower M bits of the page, another page should also trigger the prefetcher. Nevertheless, we do not observe this behavior even when M is increased to 20, suggesting that *the XPT prefetcher should be either indexed by the full physical address or possesses a tag for each physical page.*

C. Page Boundary

To determine if the XPT prefetcher is enabled in the case of huge-paging (larger than normal size of 4 KiB) and cross-4 KiB accesses. The benchmark shown in the Listing 3

Table III
THE XPT PREFETCHER TRIGGERING RESULTS WITH VARIOUS PAGE TABLES SETTING.

Page Table Size	Cross 4 KiB-Page Boundary Trigger		
	Contiguous on Virtual Address	Contiguous on Physical Address	Is XPT Triggered?
1 GiB	Yes	Yes	✗
2 MiB	Yes	Yes	✗
4 KiB	Yes	No	✗

```
void train(char *ptr, int num) {
    int n = 32;
    int random_num[n];
    generate_random(random_num);
    for (int i = 0; i < num; i++) {
      for (int j = 0; j < n; i++) {
          int idx = random_num[j];
          char junk = ptr[i*4096+idx*CACHE_LINE];
      }
    }
}
```

Listing 4. Determine the number of entries of the XPT prefetcher.

first maps three memory pools named `huge_1GiB_page`, `huge_2MiB_page`, and `normal_4KiB_page`. The first two memory pools will be allocated with huge page tables (1 GiB or 2 MiB), and the last one will be allocated with a normal page table, *i.e.*, 4 KiB. All of these pages are tagged with `MAP_LOCK` to avoid unused pools being reclaimed and impacting the experimental results. We then train the XPT prefetcher on the 4 KiB region and test it on another 4 KiB region. In this case, only `normal_4KiB_page` exhibits the cross-page prefetches and the XPT prefetcher should be re-trained on the new page. The result presented in Table III, however, shows that even `huge_1GiB_page` and `huge_2MiB_page` cannot trigger the XPT prefetcher after crossing the 4 KiB boundary, which indicates that *the XPT prefetcher has a 4 KiB boundary tag in every entry that prevents cross 4 KiB-page prefetch.*

D. Number of Entries and Set Associativity

We construct a microbenchmark that trains the XPT prefetcher on varying numbers of 4 KiB pages (Listing 4). By giving different numbers to `num`, we can create `num` entries in the XPT prefetcher. After training, we then use the `test` function shown in Listing 1 to test if the first trained page can still trigger the XPT prefetcher.

The results of the experiment are presented in Figure 4. The DRAM access latency increases sharply to approximately 350+ cycles after accessing 256 pages, and then remains stable. *This indicates that the number of entries in the XPT prefetcher is 256.* It is worth noting that the gradual increase in latency with the linear slope between the first page and the 256^{th} page appears to be the result of queuing delay. We hypothesize that the queuing delay observed is not caused by the memory controller, as the latency remains unchanged after priming more than 256 entries. However, if the delay is attributed to the memory controller, we expect a continuous latency increase beyond 256 entries. More specifically, the

Figure 4. The number of entries of the XPT prefetcher.

delay can be attributed to either the constrained MSHRs of the prefetcher or the limited number of MSHRs within the core. Our experimental investigation reveals a critical observation: upon priming 12 entries, memory latency consistently exceeds 200 cycles, and subsequently continues to rise. This indicates that if the number of in-flight requests exceeds 12, the excess requests will have to wait in a queue, leading to the queuing delay. To further investigate if the delay is caused by the core MSHRs, we generate different cache misses on the same page, and then test the memory latency of an uncached data block on this page. If this queuing delay is caused by core MSHRs, we would anticipate an observable elevation in the access latency of the test data block subsequent to generating a number of in-flight memory accesses. However, we did not find any significant difference, even with the number of cache misses growing to 60. This experiment suggests that the queuing delay originates from the restricted number of MSHRs of the XPT prefetcher, most likely capped at a count of 12.

In order to accurately determine the set associativity of the XPT, we allocate a 16 GiB memory pool and generate $N+1$ physical pages with identical bits from the 13^{th} bit to the $(13+M)^{th}$ bit. We test with varying values of N, specifically $N = 4, 8, 16, 32, 64$, in an attempt to infer the number of cache ways. Additionally, we define $M = 256/N$, which is used to compute the set number of the corresponding way.

For example, if we guess that the XPT prefetcher uses a 4-way set-associative cache, we will prime 5 (4+1) entries to map to the same set, and at least one entry should be evicted, thereby stopping prefetching on that particular page. However, we observed that no eviction occurs even when we increased the guessed number of ways to 128. Based on these findings, *we hypothesize that the XPT operates as a fully-associative cache.*

E. Benefits of the XPT Prefetcher

The XPT prefetcher operates based on physical page indexing. This means that when a memory access is not found in the L2 cache, it triggers a search in both the XPT prefetcher and the LLC. In the XPT prefetcher, once it detects that a particular page has generated 32 (or more) LLC misses, the prefetcher begins to access the DRAM to retrieve the data in memory. Simultaneously, the memory access also undergoes an LLC lookup, a process that involves several cycles for computing the LLC slice hash, determining the set index, and finally, searching the target set to determine if the data resides there.

Table IV

THE XPT PREFETCHER INVALIDATING RESULTS ON DIFFERENT SCENARIOS. NOTE THAT WE BUILD A SHARED MEMORY REGION TO ENABLE TWO PROCESSES TO ACCESS THE SAME PHYSICAL PAGE. *IN OUR EXPERIMENTS, WE EVALUATE VARIOUS CORE PAIRINGS, *e.g.*, CORE 0 AND CORE 1, AND CORE 0 AND CORE 15, TO INVESTIGATE THE POTENTIAL IMPACT OF CORE DISTANCE ON THE RESULTS. OUR FINDING SHOWS THAT WE ALWAYS SEE SIMILAR RESULTS, WHICH IMPLIES THAT THE XPT PREFETCHER REMAINS UNAFFECTED BY CORE DISTANCE ON THE SAME SOCKET ON THIS MICROARCHITECTURE.

Scenario	Cross Process	Cross Thread	Cross Core*	Same Core	Training Operation		Invalidating Operation		Invalidate?
Scenario-1	✓	✓	✓	✓	Process 1	Page A	Process 2	page A	Process 1, page A, ✓**trigger**
Scenario-2	✓	✓	✓	✓	Process 1	Page A	Process 2	Page B	Process 1, page A,B, ✓**trigger**
Scenario-3	✓	✓	✓	✓	Process 1	256 Pages	Process 2	Page A	Process 1, first page ✗**no trigger**
Scenario-4	✓	✓	✓	✓	Process 1	Page A	Process 2	Change A's Permission	Process 1, page A, ✗**no trigger**
Scenario-5	✓	✓	✓	✓	Process 1	Page A	Process 2	Change B's Permission	Process 1, page A, ✓**trigger**
Scenario-6	✓	✓	✓		Process 1	Page A	Process 2	Flushes A from TLB	Process 1, page A, ✓**trigger**
Scenario-7	✓	✓	✓	✓	Process 1	Page A	Process 1	Flushes A from TLB	Process 1,2, page A, ✗**no trigger**
Scenario-8	-	✓	✓		Thread 1	Page A	Thread 2	page A	Thread 1, page A, ✓**trigger**
Scenario-9	-	✓	✓		Thread 1	Page A	Thread 2	page B	Thread 1, page A, ✗**no trigger**
Scenario-10	-	✓		✓	Thread 1	Page A	Thread 2	page B	Thread 1, page A, ✓**trigger** Thread 1, page B, ✓**trigger**
Scenario-11	-	✓	✓		Thread 1	Pages A, B	Thread 2	page C	Thread 1, page A, ✗**no trigger** Thread 1, page B, ✓**trigger**

✓:This configuration is enabled for the scenario. -: This configuration is disabled for the scenario.

If the data hits in the LLC, the processor gets back the data. In contrast, in case of an LLC miss and without XPT enabled, the processor needs to request the data from the main memory. However, the XPT prefetcher optimizes the access latency for LLC misses since **the memory request has been pre-issued as the prefetcher has predicted that this memory access would result in an LLC miss, effectively optimizing the memory latency.** This LLC miss then increases the miss counter within the XPT prefetcher.

F. Key Observations and Leakage Sources

As we discussed in Section II-A, there are two main root causes leading to data leaks from the XPT prefetcher (XPT evictions and TLB flushes). In this section, we will discuss our key observations. Table IV consists of eight distinct scenarios. The first two scenarios involve the creation of two threads running in separate processes, with one of the processes (Process 1) trained on shared page A. Another process (Process 2) is then trained on shared page A or B, and we assess the ability of Process 1 to continue triggering the XPT prefetcher on page A. The results of these two scenarios allow us to conclude that *the XPT is shared among multiple processes*. In the third scenario, Process 1 trains the XPT prefetcher using its private pages. Process 2 then accesses a private page. We find that the first page trained by Process 1 (i.e., the oldest page) will no longer trigger the XPT prefetcher. These results indicate that the XPT prefetcher is using a Least Recent Usage (LRU) replacement policy.

> ***Observation 1***. The eXtension Prediction Table (XPT) is a global structure with 256 entries shared by all processes and all cores. An LRU replacement policy is used to evict an entry when the table is full.

In order to comprehensively examine the influence of the page management unit, particularly the TLB, on the behavior of the XPT prefetcher, we conducted the fourth, fifth, and sixth scenarios. First, Process 1 running on Core 1 trains on a shared page A. Then, Process 2 running on another core changes the permission of this shared page A to generate a remote TLB shootdown on Core 1. Page A will thus be invalidated in Core 1's TLB. If Process 2 is running on the same core as Process 1, it flushes Page A in the TLB directly using `MOV CR3`. However, if Process 2 changes other pages' permission, page A will not be impacted (see the fifth scenario). We also notice that if Process 2 only flushes its local TLB (not the remote TLB shootdown), Process 1 can still trigger the prefetcher on Page A (see the sixth scenario). This implies that the entry will be invalidated only if the corresponding ASID's TLB entry is invalidated/flushed. Lastly, if Process 1 flushes Page A from TLB (e.g., `INVLPG`), we found that Process 2 cannot trigger the prefetcher. Our results indicate that if a page trained by a process is invalidated or flushed from this process's TLB, this page can no longer trigger the prefetcher.

> ***Observation 2***. When a page mapping is flushed/invalidated from the TLB by the trainer, the corresponding page in the XPT prefetcher will also be flushed/invalidated.

To further experiment with an in-depth study of the characteristics of the XPT prefetcher, we conducted the remaining scenarios presented in Table IV. The eighth scenario highlights that the XPT prefetcher is similarly *shared between different threads within the same process space*.

The ninth scenario suggests that the XPT prefetcher should possess a thread or core identification tag for controlling its status. As a result, if two threads are executing in the same process space but on different cores, *i.e.*, they share the same Address Space ID (ASID), and one of the threads accesses a page not present in the XPT, then the well-learned entries associated with that ASID will be invalidated. This raises two questions:

1) Is the XPT prefetcher tagged by thread ID or core ID?
2) Will all entries related to the ASID be invalidated?

To further investigate the first question, we ran two threads Thread 1 and Thread 2 on the same core (the tenth scenario)

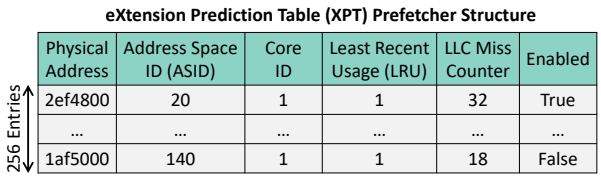

Figure 5. The architecture of the XPT prefetcher

and discovered that the XPT entries trained by Thread 1 are not evicted by Thread 2. This means that the XPT is tagged by the CoreID. This also makes sense as the hardware only records CoreID and ASID.

The eleventh scenario was conducted to provide an answer to the second question (i.e., will the invalidation caused by different threads impact all well-trained entries?). In this scenario, Thread 1 trains the XPT prefetcher on pages A and B, and then Thread 2 trains the XPT prefetcher on page C. The results showed that only the prefetching of page A was stopped. In another experiment, where Thread 1 trains the XPT prefetcher on pages A, B, and C and then Thread 2 trains on pages D and E, the prefetching of pages A and B was found to have been stopped.

These results indicate that the intra-process replacement policy is also using LRU but the replacement is determined by the CoreID. Additionally, this finding implies that if a process is moved to a different core, the entry trained by the previous core will be reclaimed (invalidated and then reused), as the XPT doesn't know if the entry trained by the previous core will still be used. This approach creates more room for holding additional entries.

G. Summary of the XPT Prefetcher Operations

The XPT prefetcher, depicted in Figure 5, operates as follows. Upon accessing a physical page, it checks for a hit in the table. If it is present and the cache miss counter is 32 or higher, then the prefetcher is triggered and a speculative load is issued. When a table miss occurs, the XPT prefetcher checks if the ASID of the physical address is recorded. If present, it then checks CoreIDs. The new page replaces the oldest entry of other CoreIDs under the same ASID using the LRU replacement policy. We call this policy a **core-tagged LRU policy**. If there are no other CoreIDs associated with this ASID or if the ASID is not presented in the table, the page is allocated with a new entry with a cache miss counter of 1, unless the prefetcher is full, in which case the global oldest entry is replaced using the LRU policy.

In summary, experimental results support that the eviction policy should follow the core-tagged LRU policy along with the global LRU policy. Our key observations in this section lead to various successful and practical side-channel and covert-channel attacks.

V. PrefetchX Attack Primitives

In this section, we introduce two attack primitives based on our reverse-engineering observations from the XPT prefetcher.

A. PrefetchX-*Evict*

The first attack primitive is named PrefetchX-Evict, which consists of four steps:

- **Step 1**: Priming the XPT entries;
- **Step 2**: Waiting for victim execution (it will either evict or not evict the oldest primed XPT entry);
- **Step 3**: Probing the oldest primed XPT entry to assess the victim eviction.
- **Step 4 (*Optional*)**: Evict the victim's entry.

More concretely, the attacker first primes all entries in the XPT and waits for the victim to execute the target code. The attacker then probes the oldest entry to infer if the target code is executed (e.g., `square` in RSA).

Note that the initial three steps offer the attacker only a single opportunity to detect the victim's behaviors. In scenarios where repeated inference of the victim's behaviors is necessary (e.g., multiple iterations in RSA decryption), the attacker must proceed to *Step 4*. To evict the victim's entry, the most efficient and general method involves accessing the remaining 255 pages under the attacker's control (i.e., generating cache misses on these 255 pages). Subsequently, the victim's entry becomes the least recently used and will be displaced once the attacker establishes a new entry (generating 32 cache misses on a fresh page). This eviction technique mandates the generation of 287 cache misses by the attacker to successfully displace the victim's page. Due to the memory- and instruction-level parallelism, on our test platform, this capability is harnessed to execute the eviction procedure efficiently, requiring a maximum of 16,000 cycles (5 us) to complete. This temporal aspect holds significant importance, as it governs the synchronization mechanism in side-channel attacks and establishes the upper bandwidth limit of our presented covert-channel, which will be introduced in subsequent sections.

B. PrefetchX-*Flush*

The second attack primitive, PrefetchX-Flush, is constructed for building higher throughput and better noise-resilient cross-core covert-channels compared to PrefetchX-Evict primitive. PrefetchX-Flush requires shared memory and consists of three steps:

- **Step 1**: Sender primes the XPT using a shared page;
- **Step 2**: Receiver tests the prefetcher availability;
- **Step 3 (*Optional*)**: Sender flushes the trained entry.

Using PrefetchX-Flush, the sender and the receiver could transmit secret information with higher throughput and without using caches. We will detail how to establish a covert-channel using PrefetchX-Flush primitive in Section VII.

By following the steps in these two attack primitives, PrefetchX is able to effectively leak the victim's sensitive page-related actions through side-channel analysis, while also enabling covert-channel attacks. Section VI describes our side-channel attacks, while Section VII describes the details of how to use PrefetchX for covert-channel attacks.

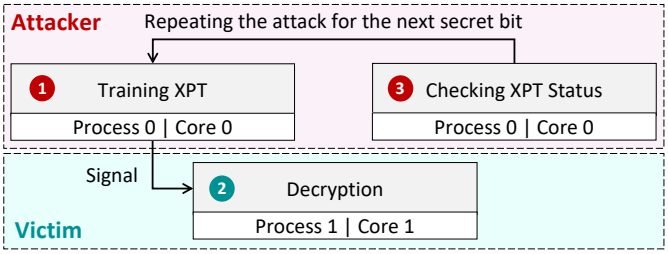

Figure 6. Overview of attacking RSA applications. After the attacker trains the XPT prefetcher, the victim decrypts the message and the attacker then can check the prefetcher status.

VI. Cross-Core Side-Channel Attacks

In light of our observations from the XPT prefetcher behavior, we propose a new side-channel attack that exploits these characteristics to leak secret keys from real-world cryptographic applications. In this section, we focus on ***Observation 1*** and deploy our PrefetchX-Evict primitive to exploit the contention and eviction policies of the XPT prefetcher.

A. Attacking Real-World Square-and-Multiply RSA

In this attack, we leak the private exponent (exp) of the Square-and-Multiply RSA application used in the latest MbedTLS [3] and the GnuPG 1.4 [14], as real-world applications that have been used as proof-of-concept by recent work [13], [19]. Listing 5 shows how MbedTLS [3] performs square-and-multiply. To optimize performance, MbedTLS introduces a sliding window to allow the algorithm to operate on multiple bits of exp during one iteration. However, prior works [13], [31] have shown that attacks on window length '1' can be extended to an arbitrary length. Therefore, we set the window length to 1 to make sure only one bit of exp is operated in every iteration. For every iteration, if the least significant bit (LSB) of the exp is equal to b'1, the base exponent_bits_in_window will be loaded from memory and be used to perform multiplication (lines 19–22). Otherwise, only a square operation will be executed (lines 9 and 10). GnuPG also has a similar structure.

The least recently used XPT entry will be evicted if a new process causes a cache miss on an XPT-uncached page. (see ***Observation 1***). Hence, an attacker first primes the XPT and then measures the prefetcher status after each decryption iteration to know if the secret-dependent branch (line 7 in Listing 5) is executed or not. Figure 6 shows an overview of our attack to reveal the RSA private exponent. For a successful attack, we need to address two challenges: (1) how to synchronize with the victim thread to guarantee we can measure the XPT prefetcher status at a proper time. (2) How to flush exponent_bits_in_window (as it is initialized first, i.e., not a copy-on-write variable, it is stored in a separate page with other not-copy-on-write variables) after each iteration (as we need to let the victim insert the page stored exponent_bits_in_window into the XPT prefetcher after each iteration).

```
1  while (E->p[nblimbs] is not 0) {
2      /* Not copy-on-write, in a sperated page */
3      size_t exponent_bits_in_window = 0;
4      ...
5      /* E->p[nblimbs] is exp */
6      E->p[nblimbs] = E->p[nblimbs] >> bufsize
7      ei = E->p[nblimbs] & 1;
8      ...
9      if (ei == 0 && state == 1) {
10         /* Square */
11         MBEDTLS_MPI_CHK(mpi_select(...));
12         mpi_montmul(...);
13         continue;
14     }
15
16     /* Square */
17     MBEDTLS_MPI_CHK(mpi_select(...));
18     mpi_montmul(...)
19
20     /* Multiply */
21     size_t exponent_bits_in_window |= (ei << (
       window_bitsize - nbits));
22     ...
23     MBEDTLS_MPI_CHK(mpi_select(...,
       exponent_bits_in_window));
24     mpi_montmul(...);
25 }
```

Listing 5. Code segment of the Square-and-Multiply Exponentiation in latest MbedTLS.

Since the victim and the attacker are running on different processes and different cores without any shared resources, the most common synchronization technique is adding a timing-fixed waiting delay [19], [45]. As evicting the victim's entry consumes 5 us, we thus add a 100 us waiting delay for the victim after each iteration to ensure we have enough time to sample the XPT prefetcher variation and prime the XPT again. Since the waiting delay is constant, it will not help the attacker infer any extra information from the victim directly.

There are many existing techniques designed to delay a victim. For instance, the task scheduler attack [17] can achieve this by executing a denial-of-service attack on the Linux scheduler, thereby allocating very small execution intervals to the hot region of the victim process (e.g., decryption) to cause delays. Additionally, the attacker can leverage the ring bus attack [33] to exhaust bus resources to delay the victim. Moreover, the performance degradation attack [2] can identify the hot region of the victim's process and occupy the resource of the coherence directory, memory channels, or cache during the hot region execution to slow down that region. It is important to note that neither of these methods requires OS privileges. It is also worth noting that this delay is not strictly required for synchronization, especially if the attacker can prime the XPT prefetcher more rapidly than each decryption iteration, such as through the use of hyper-threading and/or multiple cores to prime the XPT prefetcher in parallel.

Moreover, concerning the second challenge, inspired by prior work [7], we also found out that the victim's cached data is highly likely to be flushed after the context switch. Thus, the attacker can implicitly flush exponent_bits_in_window by sending a signal to the victim to cause a context switch

after each iteration. Note, that sending a signal to another process on another core is a standard function to processes without requiring special privileges if these processes run with the same user-id. The signal itself will not cause a context switch. However, the OS will typically stop the normal flow of the process and execute the signal handler associated with that signal even if the signal does nothing for the victim (e.g., `kill(PID, SIG_CONT)` or `kill(PID, 0)`). Thus, a context switch happens during the signal handler processing in the kernel model. Although almost all cached data could be evicted at the context switch, the attacker is still able to recognize the victim's access pattern, as one extra entry trained by the attacker will be evicted if the multiplication is executed (`exponent_bits_in_window` goes to a separate page).

In this attack, the attacker utilizes the same user-id as the victim, as sending the attack signal requires the attacker to either be a privileged user or under the same user-id as the victim. However, it is important to note that due to the isolation mechanisms between different processes, even if an attacker uses the same user-id as the victim, direct access to sensitive data remains infeasible.

Note, that purpose of sending a signal is to direct the victim's memory requests to look up the LLC/DRAM, thereby accessing the XPT prefetcher. Alternative techniques exist for achieving this without the same user-id requirement. For example, the utilization of a coherence directory primitive [45] allows to reset cache coherence status on other processor cores, even within a non-inclusive memory system. Subsequently, memory accesses will retrieve data from the DRAM, making them training the XPT prefetcher.

In Section VIII-A, we present our proof-of-concept (PoC) attack results that led to the successful extraction of one byte of the RSA secret key in 20 seconds.

B. Attacking Low-Resolution User Privacy without Synchronization

We found that certain events related to drivers (*e.g.*, keystrokes, Bluetooth connections, network packet transmissions via a network card, etc.) will have long triggering intervals (i.e., low resolution). More concretely, the time interval between two events is very long (microsecond- or millisecond-grained). In addition, many of these events are related to user privacy. Thus, it is worthy to attack these events without any synchronization requirement.

1) Attacking Keyboard Activities: Our first driver-event attack focuses on leaking the precise timing of keystrokes, *i.e.*, when the keyboard is activated by the victim. This leakage is very important since it can assist in reconstructing typed words from users [28], [47].

We consider a victim that receives user input from the keyboard (*e.g.*, `getchar`, `scanf`, etc.), and writes the input into a private page (e.g., a file mapped into memory by `mmap`). In the case of keyboard activation, we assume that the victim is appending characters to the file via keyboard, thereby generating a succession of cache misses and repeatedly evicting the oldest entry trained by the attacker. The attacker

```
1 void victim_client() {
2     int sd = socket(AF_INET,SOCK_STREAM,0);
3     struct sockaddr_in serveraddr,clientaddr;
4     set(serveraddr); //set network protocol
5     socklen_t len = sizeof(clientaddr);
6     int acceptfd = accept(sd,(struct sockaddr *)&
7                           clientaddr,&len);
8     int recvbytes = recv(acceptfd,private_page,
9     ...);
}
```

Listing 6. Code segment of the victim TCP client. Line 6 establishes the connection with the client and line 8 receives the packets.

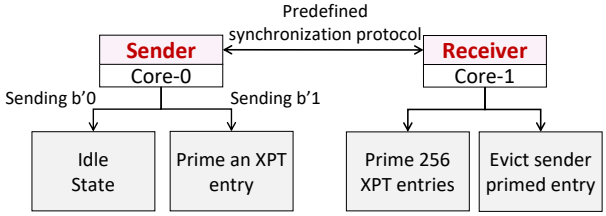

Figure 7. Attack flow of PREFETCHX-Evict based covert-channel.

repeatedly executes the PREFETCHX-Evict primitive to infer if the victim is writing into her private file. We discuss the results of our attack in Section VIII-B.

2) Attacking Network Traffic: Network traffic analysis attacks [1], [21], [39] represent a serious threat to online security. Although an attacker cannot get the content of the message directly, an attacker can determine the location of both sides of the communication, deanonymize communicating parties, and deduce sensitive information by observing the patterns of datagrams (*e.g.*, packet transmission timing interval, number of transmitted packets, etc.).

Listing 6 shows the code segment of the TCP client as our victim. The key parts are lines 6 and 8. Line 6 establishes the connection with the client, while line 8 is used to receive packets and write these packets into the victim's private page. The victim on Core 1 always tries to receive packets. Similar to our keystroke attack, the attacker on Core 0 repeatedly performs the PREFETCHX-Evict primitive. In case of receiving a packet by the victim, the attacker can infer the timing if the oldest entry in the prefetcher does not trigger.

VII. CROSS-CORE COVERT-CHANNEL ATTACKS

Attack assumptions. Similar to previous covert-channel attacks [18], [19], the sender and receiver synchronize with each other using a timing-fixed waiting delay. The sender and receiver should agree on predefined protocols, including synchronization, encoding, and error correction up-front. In addition, the PREFETCHX-Flush covert-channel assumes that the sender and the receiver share data via a shared page.

A. Cross-Core Covert-Channel Attack via PREFETCHX-Evict

Figure 7 shows a novel cross-core covert-channel built on top of the PREFETCHX-Evict primitive, without using shared memory. The receiver initiates training the XPT prefetcher across 256 distinct pages, thereby priming the entire XPT.

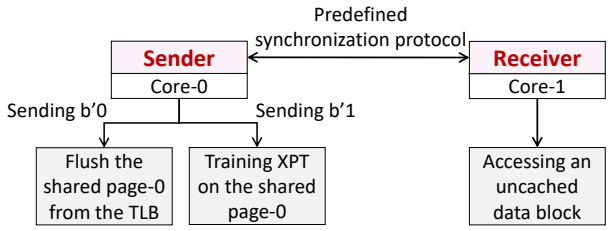

Figure 8. Attack flow of PREFETCHX-Flush-based covert-channel.

In the event that the sender intends to transmit signal b'1, it accesses a private page, triggering the eviction of the oldest page trained by the receiver. In contrast, should the sender opt to transmit b'0, it maintains an idle state. Upon detecting the eviction of its oldest entry, the receiver employs the technique delineated in *Step 4* mentioned in Section V-A, effectuating the eviction of the page accessed by the sender. This strategy guarantees that the receiver always occupies all entries before the sender sends a new bit of the secret.

For the purpose of synchronized operation within a fixed timeframe, it is imperative to ensure that this temporal window is longer than the most time-intensive step of a secret transmission, namely, the eviction of the sender's entry. In the pursuit of this objective, in this paper, a synchronization waiting period of 18,000 cycles is chosen, which consists of both the eviction delay (16,000 cycles) and error correction overhead.

Concretely, the sender initiates secret transmission following a precise delay of 16,000 cycles, subsequently followed by an additional latency of 2,000 cycles. On the receiver side, a determination is made whether an eviction of the sender's entry is needed (i.e., if the sender sent b'1 last round). If deemed necessary, the *Step 4* is executed, and subsequent waiting continues until the culmination of $16,000$ cycles. Conversely, if eviction is unnecessary, the wait spans precisely $16,000$ cycles. Subsequently, the receiver allocates $2,000$ cycles to the iterative monitoring of the oldest entry's status, thereby effecting error correction. The throughput can achieve 21 KiB/s with an error rate that is lower than 8%.

B. Cross-Core Covert-Channel Attack via PREFETCHX-Flush

To further leverage the reverse-engineered characters of the XPT prefetcher to build a higher throughput and lower noise covert-channel, we then construct a PREFETCHX-Flush based covert-channel. Figure 8 depicts the communication flow. If the sender wants to transmit b'1, it first trains the XPT on the first half of the shared page on Core 0. In the next round, if the sender wants to send b'0, it will flush his trained page from TLB, and the XPT entry on this page will also be flushed. The receiver, running on Core 1, randomly accesses a cache line on the last half page of the shared page to observe if the cache miss is an optimized LLC miss or a normal LLC miss (XPT is not triggered). An optimized cache miss indicates that the sender did not reset the XPT, sending b'1, and vice versa. As the training of a page and TLB flush requires a maximum of 900 ns in our test platform, a fixed 1 us synchronization delay is used in the PREFETCHX-Flush covert-channel, i.e., the receiver checks the XPT prefetcher after every 1 us (another 100 ns is used to correct the error). The highest bandwidth of this covert-channel is 122 KiB/s with an error rate lower than 1%. Our results demonstrate the feasibility and robustness of our proposed covert-channel exploiting ***Observation 2***.

Table V
ARCHITECTURE AND SYSTEM CONFIGURATIONS.

AWS EC2 Instance	m6i.4xlarge
Processor	Intel(R) Xeon(R) Platinum 8375C
Architecture	Ice Lake (Sunny Cove)
CPU cores	16
Last Level Cache	Non-inclusive, 54 MiB
Operating System	Ubuntu 20.04
ASLR/KASLR	Enabled
DRAM	DDR4, 64 GiB

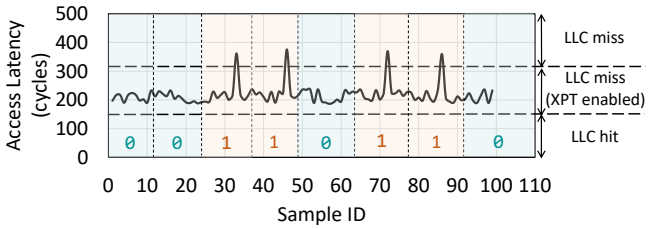

Figure 9. RSA attack results (second Multiplication page eviction shown). The x-axis orders the secret bits from most significant bit (MSB) to least significant bit (LSB). Note, that in case of b'1 as secret bit, an extra well-trained entry in XPT is evicted which results in a normal LLC miss.

VIII. EXPERIMENTAL RESULTS

Experimental environment. We perform Proof-of-Concept (PoC) side-channel experiments on an Ice Lake machine. The system details are shown in Table V.

A. RSA Side-Channel Attacks

In our PoC, the attacker first primes all entries and waits for decryption to occur. For each round of the decryption, a page will be accessed for the Square operation no matter what the secret bit is; however, an extra page will be accessed for the Multiplication operation when the secret bit is 1. Thus, because of the LRU replacement policy (see Section IV), the first trained entry is always evicted and we need to monitor if the second trained entry is evicted (as shown in Figure 9) after each decryption entry, and the attacker does not need to know the exact page address of the victim.

As we operate on an AWS EC2 instance, our environment is within a virtual machine (VM). The VM and all other background processes (e.g., network manager, daemons) generate system noise, which may impact the attack result [13], e.g., well-trained entries may be evicted by the VM or system activities. To remove the effects of system noise, we collect 1000 traces in order to reveal one bit of the private exponent. The Square-and-Multiply RSA attack result is demonstrated in Figure 9.

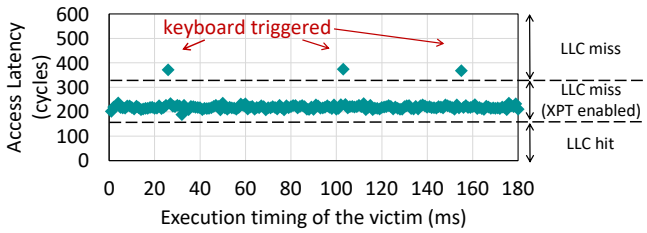

Figure 10. Side-channel attack to monitor user keyboard activities.

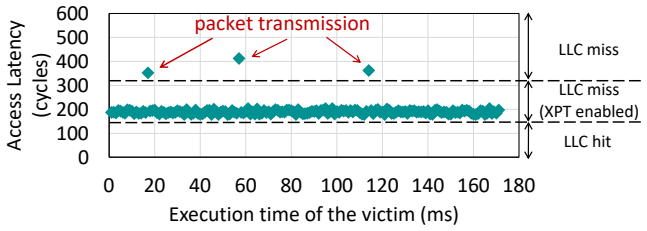

Figure 11. Side-channel attack to monitor user network traffic.

Table VI
AVERAGE MEMORY ACCESS LATENCY WHETHER A KEYSTROKE HAS OCCURRED IN DIFFERENT SCENARIOS: XPT ENABLED OR DISABLED.

Scenario	Memory Access Latency (cycles)		Distinguishable?
	keyboard activated	keyboard inactivated	
XPT	393	193	✓
no-XPT	388	383	✗

We attempted to leak a 1-byte private exponent (`b'01101100`). The results show that we successfully revealed it within 20 seconds. As every bit's computation is independent in RSA decryption, it is reasonable to estimate that PREFETCHX can break a 2048-bit (256-byte) RSA engine in approximately 80 minutes (1.4 hours), which is a practical time window for an attacker.

B. Keystroke Side-Channel Attack

Figure 10 presents the results of our keystroke attack. We clearly observe that invoking the keyboard activity function writing to the private page evicts the oldest entry of the XPT prefetcher. Specifically, the significant difference between the access latency of normal LLC misses and optimized LLC misses allows us to identify the precise timing of each keystroke. Table VI summarizes the average latency we obtained under different scenarios. The results indicate that without the XPT prefetcher, we could not detect keyboard activity. This finding provides further evidence that the XPT prefetcher plays a crucial role in our attacks.

C. Network Traffic Side-Channel Attack

Our PoC involves a client-server setup where the server communicates with the victim client (see Listing 6) via port 8888. The server randomly generates packets and then sends them to the client's private buffer. The outcome of this attack is shown in Figure 11, which clearly shows the timing of packet reception by the victim. Our results demonstrate that by observing the XPT prefetcher status, we can detect network traffic patterns that reveal information about the victim's behavior.

Resolution Analysis. In the real world, the frequency of network packet transmission can be very high, and we need to ensure that training the XPT prefetcher is shorter than packet transmission interval[6]. To investigate this, we sampled IP packets from our network interface card (NIC) and observed that the timing interval between sending two packets are consistently around $26,000$ ns. As mentioned in Section V-A, evicting a victim entry only requires 5 us, i.e., $5,000$ ns. The significant gap between the packet transmission interval and the XPT entry eviction time demonstrates that PREFETCHX's resolution is high enough to track network packets in real-world applications.

[6]Keystrokes require milliseconds to seconds to complete, which trivially the XPT prefetcher can be trained in this interval.

IX. POTENTIAL MITIGATION

While disabling the prefetcher blocks the security leaks of the XPT prefetcher, it may introduce unexpected performance overheads for memory-intensive applications with highly irregular memory accesses. A more efficient mitigation is to partition the prefetcher. To achieve leakage-free partitioning, the system needs to partition the XPT prefetcher based on the ASID and the CoreID tags that already exist in the XPT structure. This ensures to block of information leakage across different cores and different processes on the same core.

X. RELATED WORK

Cache timing attacks. Cache timing side-channels exploit the timing differences between cache hits and cache misses, enabling attackers to infer the victim's memory activities. There are two primary types of cache timing side-channels: Prime+Probe based [10], [32], [34], [43] and Flush+Reload based [16], [46]. For the Prime+Probe type, the attacker primes the cache with their data and then probes it to observe the victim's access time changes. Flush+Reload involves the attacker flushing a cache line and then waiting for the victim to reload it, thereby providing insight into the victim's memory access patterns. Cache timing side-channels are often referred as cache primitives since they serve as a building block for other hardware side-channels [37], [41], [42].

Prefetcher attacks. Side-channel attacks introduced by prefetching have received extensive attention in recent years. From the software perspective, modern processors often provide specific prefetch instructions that programmers can use to improve performance. Some attacks [15], [29] aim to bypass Supervisor Mode Access Prevention (SMAP) and KASLR on Intel and AMD processors. They exploit the timing of `PREFETCH` instructions to leak the translation level of the virtual address and infer the physical mapping. Leaky Way [18] exploits `PREFETCHNTA` in Intel processors to change the cache status and build a covert-channel attack through conflicted

Table VII
SUMMARY OF PREFETCHER/PREFETCH-BASED ATTACKS.

Trigger From	Paper	Cross Core?	Side-Channel? *	Hardware Leakage Source	Requirements					
					Cache Primitives	Shared Memory	Address of Instruction	Address of Data ‡	Speculative Execution	Algorithm Specific
Sofware	Prefetch Attack [15], [29]	✗	✓	L1/L2/LLC	●	●	○	◐	○	○
	Leaky Way [18]	✓	✗	LLC	○	○	○	●	○	○
	Adversarial Prefetch [19]	✓	✓	LLC	●	●	○	●	○	○
Hardware	Fetching Tale [8]	✗	✗	IP-Stride Prefetcher	○	○	●	○	○	○
	Unveiling Prefetcher [37]	✗	✓	IP-Stride Prefetcher + L1/L2/LLC	●	●	○	●	○	●
	Augury [42]	✗	✓	Pointer-Chasing Prefetcher + L1/L2/LLC	●	○	●	○	●	○
	AfterImage [7]	✗	✓	IP-Stride Prefetcher	○	○	●	○	○	○
	PREFETCHX (*this work*)	✓	✓	XPT Prefetcher	○	○	○	○	○	○

* ✓ : can be used as both side-channel and covert-channel attacks, ✗: can only be used as a covert-channel attack
‡ ○ : no need for the address of the victim's data, ◐ : need the page granularity address, ● : need the cache-line granularity address

cache ways. Another instruction, PREFETCHW, can leak cache coherent status and allow cross-core attacks [19].

On the contrary, hardware prefetchers in real processors typically cannot be explicitly controlled and normally require more understanding through reverse-engineering. Augury [42] investigated the data memory-dependent prefetcher in the Apple M1 processor to perform out-of-bounds reads. AfterImage [7] studied Intel IP-based stride prefetcher that enables tracking load instructions of the victim. Other works [8], [38] are either algorithm dependent or can only be used as a covert-channel. However, all existing prefetcher side-channels are single-core and they are more or less dependent on the cache hierarchy, while PREFETCHX can launch cross-core side-channel and covert-channel attacks that do not rely on the cache system. Table VII shows a summary of existing prefetcher/prefetch-based side-channel and covert-channel attacks.

XI. CONCLUSION

In this work, we uncover details of the XPT prefetcher in Intel processors, which can be intentionally trained and triggered across different cores within the same processor. Capitalizing on these features, we introduce a novel attack, named PREFETCHX, capable of leaking victims' page activities. PREFETCHX is a cross-core attack, independent of cache primitives as the foundation of many hardware attacks. To achieve this, we conducted an in-depth study of the XPT prefetcher, revealing undocumented details. We demonstrate that we can extract secret keys in the real-world Square-and-Multiply RSA application. Furthermore, we apply PREFETCHX to effectively leak the victim's driver-related events. Additionally, we showcase the applicability of PREFETCHX as a method to create cross-core covert-channel attacks, achieving low error rates when transmitting secrets. Finally, we conclude that disabling or partitioning the XPT prefetcher is required if one wants to mitigate PREFETCHX.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their detailed feedback which allowed us to improve this work. We also appreciate Andreas Diavastos's suggestions on the earlier draft of this paper. This work was supported by a grant from the National Research Foundation (NRF) of Singapore (NRF2018NCR-NCR002).

REFERENCES

[1] P. A., L. F., Z. A., H. M., P. J., W. K., and E. T., "Fingerprinting at internet scale," in *Network and Distributed System Security Symposium (NDSS)*, 2016, pp. 1–15.

[2] T. Allan, B. B. Brumley, K. Falkner, J. van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *Annual Conference on Computer Security Applications (ACSAC)*, 2016, p. 422–435.

[3] "MbedTLS: An open source, portable, easy to use, readable and flexible SSL library," *Arm Holdings plc*, 2019.

[4] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Conference on Supercomputing (Supercomputing)*, 1991, p. 176–186.

[5] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "An efficient temporal data prefetcher for L1 caches," *IEEE Computer Architecture Letters (CAL)*, vol. 16, no. 2, pp. 99–102, 2017.

[6] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 131–142.

[7] Y. Chen, L. Pei, and T. E. Carlson, "AfterImage: Leaking control flow data and tracking load operations via the hardware prefetcher," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 2, 2023, pp. 16–32.

[8] P. Cronin and C. Yang, "A Fetching Tale: Covert communication with the hardware prefetcher," in *International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 101–110.

[9] S. Deng, B. Huang, and J. Szefer, "Leaky Frontends: Security vulnerabilities in processor frontends," in *Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 53–66.

[10] 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 (USENIX Security)*, 2017, pp. 51–67.

[11] J. Doweck, "White paper inside Intel® Core™ microarchitecture and smart memory access," *Intel Corporation*, pp. 72–87, 2006.

[12] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani, "Prefetch-Guard: Leveraging hardware prefetches to defend against cache timing channels," in *Symposium on Hardware Oriented Security and Trust (HOST)*, 2018, pp. 187–190.

[13] S. Gast, J. Juffinger, M. Schwarzl, G. Saileshwar, A. Kogler, S. Franza, M. Köstl, and D. Gruss, "Squip: Exploiting the scheduler queue contention side channel," in *2023 IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 468–484.
[14] "GNU privacy guard," https://gnupg.org/, 2022.
[15] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Conference on computer and communications security (CCS)*, 2016, pp. 368–379.
[16] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A fast and stealthy cache attack," in *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016, pp. 279–299.
[17] 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, pp. 490–505.
[18] Y. Guo, X. Xin, Y. Zhang, and J. Yang, "Leaky Way: A conflict-based cache covert channel bypassing set associativity," in *Symposium on Microarchitecture (MICRO)*, 2022, pp. 646–661.
[19] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, "Adversarial prefetch: New cross-core cache side channel attacks," in *Symposium on Security and Privacy (S&P)*, 2022, pp. 1458–1473.
[20] D. Guttman, M. Arunachalam, V. Calina, and M. T. Kandemir, "Chapter 21 - prefetch tuning optimizations," in *High Performance Parallelism Pearls*, 2015, pp. 401–419.
[21] A. Hintz, "Fingerprinting websites using traffic analysis," in *Workshop on Privacy Enhancing Technologies (PET)*, 2002, pp. 171–178.
[22] Intel, "Disclosure of H/W prefetcher control on some Intel processors," https://radiable56.rssing.com/chan-25518398/article18.html, 2018.
[23] Intel, "Intel® 64 and IA-32 architectures software developer's manual," *Intel Corporation*, 2019.
[24] "Understanding Intel software guard extensions (Intel SGX)," https://www.intel.sg/content/www/xa/en/architecture-and-technology/software-guard-extensions-enhanced-data-protection.html, 2021.
[25] Intel, "HPC cluster tuning on 3rd generation Intel Xeon Scalable processors," https://www.intel.com/content/www/us/en/developer/articles/guide/hpc-cluster-tuning-on-3rd-generation-xeon.html, 2022.
[26] Intel, "Intel® 64 and IA-32 architectures optimization reference manual," *Intel Corporation*, 2023.
[27] 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.
[28] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical cache attacks from the network," in *Symposium on Security and Privacy (S&P)*, 2020, pp. 1–19.
[29] M. Lipp, D. Gruss, and M. Schwarz, "AMD prefetch attacks through power and time," in *USENIX Security Symposium (USENIX Security)*, 2022.
[30] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium (USENIX Security)*, 2018, pp. 973–990.
[31] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Symposium on security and privacy (S&P)*, 2015, pp. 605–622.
[32] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' Track at the RSA Conference (CT-RSA)*, 2006, pp. 1–20.
[33] 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 (USENIX Security)*, 2021, pp. 645–662.
[34] C. Percival, "Cache missing for fun and profit," in *BSDCan Ottawa*, 2005, pp. 1–13.
[35] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal Attack: leaking control-flow in sgx via the cpu frontend," *USENIX Security Symposium (USENIX Security)*, 2021.
[36] A. Rohan, B. Panda, and P. Agarwal, "Reverse engineering the stream prefetcher for profit," in *European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020, pp. 682–687.
[37] 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 *Conference on Computer and Communications Security (CCS)*, 2018, p. 131–145.
[38] 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 *Conference on Computer and Communications Security (CCS)*, 2018, pp. 131–145.
[39] S. Siby, M. Juárez, N. Vallina-Rodriguez, and C. Troncoso, "DNS privacy not so private: the traffic analysis perspective," in *Privacy Enhancing Technologies Symposium (PETS)*, 2018, pp. 1–19.
[40] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," pp. 252–263, 2006.
[41] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking data on Intel CPUs via cache evictions," pp. 339–354, 2021.
[42] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *Symposium on Security and Privacy (S&P)*, 2022, pp. 1491–1505.
[43] D. Wang, Z. Qian, N. Abu-Ghazaleh, and S. V. Krishnamurthy, "PAPP: Prefetcher-aware prime and probe side-channel attack," in *Design Automation Conference (DAC)*, 2019, pp. 1–6.
[44] H. Xiao and S. Ainsworth, "Hacky Racers: Exploiting instruction-level parallelism to generate stealthy fine-grained timers," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 2, 2023, pp. 354–369.
[45] 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 *Symposium on Security and Privacy (S&P)*, 2019, pp. 888–904.
[46] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security Symposium (USENIX Security)*, 2014, pp. 719–732.
[47] K. Zhang and X. Wang, "Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems." in *USENIX Security Symposium (USENIX Security)*, 2009, pp. 17—32.
[48] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Binoculars: Contention-based side-channel attacks exploiting the page walker," in *USENIX Security Symposium (USENIX Security)*, 2022, pp. 699–716.

APPENDIX A
ARTIFACT APPENDIX

A. Abstract

PREFETCHX is a hardware side-channel inside the 3^{rd} Generation Intel Xeon processor family[7]. In this artifact, we provide the needed environments and information to reproduce the main results presented in the paper, i.e., the RSA attack, the keystroke attack, and the network traffic attack.

B. Artifact Check-List (Meta-Information)

- **Compilation:** GCC 8.4.0 or above with -O1
- **Run-time environment:** Ubuntu 20.04
- **Hardware:** AWS m6i.xlarge
- **Execution:** In three cases, each can be finished in minutes. Some of them may require multiple runs.
- **Metrics**: Memory latency, indicating if the XPT prefetcher is triggered or not.
- **Output:** Terminal output or figures that show if the XPT prefetcher is (not) triggerable.
- **Experiments:** Scripts provided.
- **How much disk space required (approximately)?:** 256MB
- **How much time is needed to prepare workflow (approximately)?:** 10 mins
- **How much time is needed to complete experiments (approximately)?:** Half an hour
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Creative Commons Attribution 4.0 International
- **Archived (provide DOI)?:** 10.5281/zenodo.10118346

[7]the latest Xeon processors at the time of paper submission

C. Description

1) How to Access: We leverage the AWS EC2 m6i.xlarge instance to launch our attacks and thus recommend using this instance to reproduce our attack results.

2) Hardware dependencies: PREFETCHX should run on Intel 3^{rd} Generation Xeon Processors.

D. Installation

The user should first create an AWS EC2 account and switch the region to `us-east-1` (N.Virginia). The user then needs to launch an m6i.xlarge instance with the Ubuntu 20.04 operating system. After logging into the instance and downloading the code from the `10.5281/zenodo.10118346` (https://zenodo.org/records/10118346/files/prefetchx.zip) via `wget`. After that, the user should first install all dependencies by entering the `prefetchx` directory and running:

```
$prefetchx: ./pre-built.sh
```

E. Experiment workflow

We will reproduce RSA attack (Section VIII-A), keystroke attack (Section VIII-B), and network traffic attack (Section VIII-C). All of them are independent; users can start with any of them. We have pre-compiled all codes, but you can also recompile them by typing `make` in their corresponding directories. Additionally, we have prepared a demo of running all the attacks presented in Section A-F: refer to Zenodo DOI. Reviewers can use this video as a reference to reproduce our attacks.

F. Evaluation and expected results

1) RSA Attack: The user can enter the `rsa_attack` directory to reproduce our RSA proof-of-concept by running:

```
$rsa_attack: ./run.sh <secret_key>
```

The expected result and attack result will then be output to the terminal. We have done a further process to make the result more understandable than Figure 9. The input `secret_key` needs to be in hex format, e.g., `d7`, with no prefix, i.e., `0x`. As the AWS virtual machine introduces interference, if you are not able to reproduce the result, you can try multiple times or restart the instance and run the PoC again. Note that, the attack might not fully recover the first bit of the secret (least significant bit), however, since the first bit has only two choices of 1 and 0 the full key can be recovered by testing both choices.

2) Keystroke Attack: The user can reproduce the keystroke attack by running:

```
$prefetchx: ./keystroke
```

The PoC will stop at some point and wait for the keystroke to continue, and the user can type `ENTER` to trigger the keystroke and continue the program. The memory latency measured at different times is output on the screen. The user should be able to see that the memory latency is higher when the keystroke is triggered. Note that, the access latencies might be higher than the numbers in Figure 10 due to extra noise, but the keystrokes would still have visibly higher latency.

3) Network Traffic Attack: In this section, we introduce how to monitor network traffic. To run the experiment, you should first go to `traffic` directory and run:

```
$traffic: ./network_traffic
```

Then, you should open another terminal and run:

```
$traffic: ./client
```

This command will create a localhost client to connect to the server. The server will detect the connection and send a packet. You will see a higher memory latency after the server sends the packet, i.e., the XPT prefetcher status is reset. Similar to the keystroke attack, the reported latency numbers might be higher than Figure 11, but they would still show visibly higher latency for network packets.