PPA: Novel Page Prefetcher-Based Side-Channel Attacks

Tengjiao Fu[1], Yu Jin[2*], Dongsheng Wang[2,3], Shuwen Deng[2,3†], Yun Chen[1†]
[1]The Hong Kong University of Science and Technology (Guangzhou)
[2]Tsinghua University, [3]Zhongguancun Laboratory

***Abstract*—Hardware data prefetchers are designed to fetch memory in advance to reduce cache misses and mitigate memory bottlenecks. By analyzing historical memory access patterns, these prefetchers predict and prefetch likely data targets. Major commercial CPU vendors, e.g., Intel, Arm, and AMD, incorporate various prefetchers in their products to optimize memory latency. While these prefetchers can significantly enhance performance, they can also introduce security concerns by accessing unintended data. In this paper, we reveal new features of the prefetcher in recent Intel Xeon processor, termed the page prefetcher. We find that this page prefetcher is indexed by the instruction pointer (IP) and can prefetch page translations (into the TLB) and cache lines across page boundaries. To investigate its implications, we propose several attacks built upon our page prefetcher attack (PPA) primitives. We demonstrate that PPA can be leveraged to expose kernel information to user space and to leak secrets from SGX enclaves to untrusted domains, such as control-flow details. Furthermore, when combined with transient attacks, PPA can extend information leakage. Our findings uncover a significant vulnerability in the page prefetcher and highlight the broad applicability of PPA in various attack scenarios.**

***Index Terms*—hardware security, prefetcher, side-channel attacks.**

I. Introduction

Within the domain of CPU microarchitecture, prefetchers have emerged as a critical component to enhance the performance of modern processors [9], [16]. Prefetchers predict and load data into the cache before it is actually requested by the CPU, thereby reducing latency and improving overall system throughput [2], [12], [18]. Hardware prefetchers, in particular, have been shown to significantly boost computational efficiency by anticipating memory access patterns and prefetching data accordingly [1], [14], [26], especially to maintain high performance in data-intensive applications.

However, recent research has discovered several vulnerabilities associated with hardware prefetchers, highlighting that these components are not immune to security threats [4]–[6], [11], [28], [29], [32]. These studies have demonstrated that prefetcher-related weaknesses can be exploited to launch sophisticated side-channel attacks, such as Augury [32], which can compromise speculative load hardening (SLH) proposed by Chandler [3]. Such findings underscore the importance of addressing security concerns in the design and implementation of prefetchers to prevent potential exploits.

In this work, we propose PPA, *a novel attack targeting a hardware prefetcher whose security properties have not been previously explored, designed to prefetch memory pages*. The page prefetcher predicts future page accesses, issues the page-walk process ahead of time, and prefetches the target page info as well as cache line into the Translation Lookaside Buffer (TLB) and L1 Cache.

We first demonstrate that this page prefetcher is a physically and functionally different prefetcher compared with hardware prefetchers explored by all existing prefetcher-related attacks [4]–[6], [11], [28], [29], [32].[1] We go further and show that: ❶ The entries of the page prefetcher are isolated from traditional IP-stride cache-line-based prefetchers; ❷ The page prefetcher requires more indexing bits than the traditional IP-stride prefetcher, which in turn provides a lower-noise environment for covert and side-channel attacks.

Based on these findings, PPA enables the following capabilities: ❶ PPA first performs a systematic and extensive reverse engineering of the page prefetcher to disclose all its parameters, including indexing policy, confidence, stride, and activation policy. ❷ PPA then uses and trains the page prefetcher across the privilege boundary, which can transmit secrets between the normal world and privilege world, e.g., kernel to the user, SGX enclave to untrusted zone. ❸ PPA is also demonstrated to be able to perform Spectre attacks [19] by triggering the page prefetcher during speculation. Finally, we report the results to the affected vendors.

Our key contributions are as follows:

- We conduct an in-depth reverse-engineering study of the page prefetcher on the Intel Xeon Processor.
- To the best of our knowledge, we are the first to demonstrate that this page prefetcher is physically and functionally distinct from hardware prefetchers explored by all existing related attacks.
- We propose PPA, a novel attack targeting a security-unexplored hardware prefetcher designed to perform page prefetching.
- Using PPA, we achieve a prefetcher-based Spectre attack, build a kernel-user covert channel that has higher bandwidth compared to previous work, and attack Intel SGX to leak secrets from its secret-dependent branch.

II. Background

A. Hardware Prefetcher

Prefetching is a widely adopted technique in modern processors that is used to mitigate the latency gap between the CPU and the memory subsystem. Prefetchers can hide long DRAM latency by predicting and prefetching data from slower memory into the high-speed cache before the CPU requests the data. Intel processors provide both software prefetching instruction interfaces and dedicated prefetching hardware components. Software prefetching requires the use of programmer knowledge or compiler information by inserting `PREFETCH` instructions into the program with an explicit memory address, whereas hardware prefetchers automatically predict the memory access address by learning the run-time memory access patterns. The speculation that occurs in the hardware prefetcher is different from the speculation of the branch predictor. If the prediction of prefetching is wrong, the useless memory accesses may waste bandwidth or pollute the cache. However, the data will not be used by the processor and will not affect the execution of the program.

*Equal contribution joint first authors.
†Shuwen Deng and Yun Chen are co-corresponding authors.

[1]The closest work is FetchBench [28], which describes a phenomenon that shares some common features with our cross-page prefetching but does not specify it or explore anything further than that. Among the documented prefetchers in Table I, only TLB and Next-Page prefetcher could cross page, but the former would not access data, while the latter has a limited prefetch distance.

979-8-3315-8924-0/25/$31.00 ©2025 IEEE

TABLE I: Documented Intel In-Core Hardware Prefetchers.

Intel Prefetcher	Location	Pattern
Data Cache Unit	L1-D	Next cache line (CL)
Instruction-Pointer Stride	L1-D	Stride pattern in CL granularity
TLB Prefetcher	L1-D	Linear address TLB prefetch
Next-Page Prefetcher	L1-D	Sequential Accesses to CL
Data Prefetch Logic	L2	128-bytes-aligned pair CL
Streamer	L2	CL forward/backward

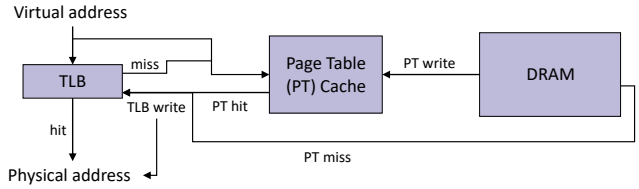


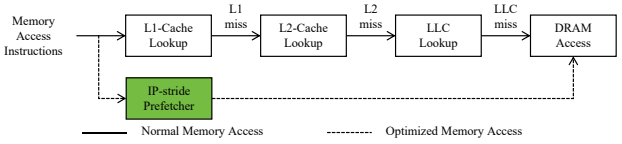
Fig. 1: Virtual address to physical address translation.

Intel has integrated at least five in-core hardware data prefetchers and a TLB prefetcher into their processor designs [9], [10], with the features of these prefetchers outlined in Table I. The Data Cache Unit (DCU) prefetcher, also known as the next-line prefetcher [30], automatically prefetches the subsequent cache line when a cache miss occurs. The IP-Stride prefetcher tracks `load` instructions that exhibit regular strides from the same IP. The Data Prefetch Logic (DPL), or adjacent prefetcher, treats data as 128-byte aligned blocks. When a cache miss occurs in one of the two cache lines within this block, the DPL triggers a prefetch for the adjacent cache line. The TLB prefetcher could cross page boundaries to start translations for TLB misses without data access. The Next-Page Prefetcher (NPP) predicts page-boundary crossings and prefetches only the next page early, although it can also perform cross-page prefetching, it cannot learn strides farther than the next page; therefore it is different and weaker than this page prefetcher. The Streamer prefetcher tracks sequential positive and negative offset streams, prefetching the subsequent or previous cache lines accordingly. Notably, prior research [25] has shown that the Streamer prefetcher retains its state even after a context switch. It operates at the L2 cache level, indexed by the physical memory address, and dynamically adjusts the number of cache lines prefetched based on system conditions such as bandwidth and streaming direction. However, these hardware prefetchers lack the flexibility of the Instruction Pointer (IP)-based stride prefetcher, also known as the IP-stride prefetcher, which offers more adaptable prefetching behavior.

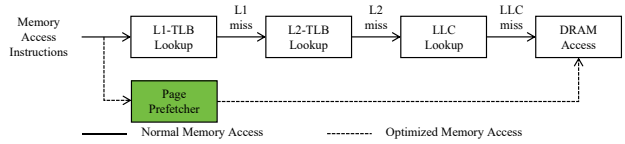
B. Virtual Memory, Page Tables, and the TLB

Most modern desktop and server systems provide each workload with the illusion of a large, continuous memory address space through the use of virtual memory. Physically, each process's memory may be divided into multiple non-contiguous physical page frames. The operating system is responsible for maintaining the mappings from the virtual addresses provided by each process to the physical addresses in dynamic random-access memory (DRAM). These mappings are stored in the page table, with a typical granularity of 4096 bytes.

Translation. The CPU's memory management unit (MMU) keeps a cache of recently used mappings from the operating system's page table in a commonly set-associative cache known as the translation look-aside buffer (TLB). The translation process is illustrated in Figure 1. More concretely, when a virtual address needs to be translated into a physical address, the TLB is checked first. If there is a hit in the TLB, the physical address is returned, and memory



(a) Overview of the IP-stride prefetcher operation.



(b) Overview of the page prefetcher operation.

Fig. 2: Comparing between IP-stride and page prefetcher.

access can resume. If a TLB miss occurs, in x86, the MMU will look up the address mapping in the page table to see if a mapping exists, which is referred to as a page walk. If one exists, it is written back to the TLB, allowing a TLB hit for the subsequent translation. However, page table lookup may fail for any of three reasons: 1) an invalid virtual address, 2) a permission issue, and 3) an unmapped physical page.

For the first two problems, the MMU or the OS will deny access if the application attempts to access an invalid address. In terms of the physical page, the OS will create a new mapping between the virtual page and the physical page or load the existing page from the DRAM.

C. Timing Side-Channel Attacks

Whenever the time taken for the processor to perform certain operations is dependent on secret values, timing side-channels can exist [34], [35]. Instruction-based timing side-channels [22] rely on the correlation between the secret and the number of CPU cycles needed to execute an instruction segment. Cache-based timing side-channels [21], [23], [27] exploit the latency gap between the cache and memory subsystems. When the secret value is related to the memory access behavior of the system, attackers have the potential to extract the secret by observing timing differences.

III. PPA Motivation

PPA exploits the page prefetcher introduced in the Intel's 3^{rd} generation Xeon processors. The page prefetcher is an extension of the IP-stride prefetcher, and its main goal is to predict future page access, pre-issue the page-walk, and finally prefetch the target page's translation into the TLB. This can reduce page access latency in the case of a correct prediction. Hence, the page accesses exhibit three levels of access latency (we use *RDTSC* to compute the latency): (1) a *TLB hit (L1 cache hit)* (less than 100 cycles in our setup and experiments), (2) *TLB miss (LLC miss)* (350+ cycles), and finally (3) *Page fault* (8000+ cycles). The TLB hits of an untouched page occur in cases where the page prefetcher has a correct prediction and is enabled. Figure 2 (a) and Figure 2 (b) show an overview of the IP-stride prefetcher and the page prefetcher alongside the cache and TLB hierarchy. This timing variation is tightly coupled with the memory activities of the executing program, which enables easy status monitoring of the prefetcher.

In this work, we make three key observations from the page prefetcher behavior that enables us to build side-channel and covert-channel attacks. First, ***we observe that the well-trained entry in the page prefetcher will be reused in different domains (e.g.,***

cross-hyperthreading, and cross-kernel). In other words, if process `p1` trains the prefetcher using a load operation with a specific IP `ip0` and another load instruction `ip1` in another privilege domain could trigger this trained entry if some specific bits of `ip1` match with the `ip0`. Second, ***the page prefetcher can load the untouched page's translation into TLB***. We find that the page prefetcher will automatically launch the page walk if the prefetched page is missed in the TLB. Third, ***we observe that the page prefetcher can be triggered during speculative execution***. We demonstrate that if process 1 trains the branch predictor and the page prefetcher, the page prefetcher can prefetch data in the misprediction path without boundary or permission checks. In Section V and Section VI, we provide the details of our observations.

IV. Threat Model

In this work, we consider a threat model where the victim process contains confidential information that the attacker aims to infer without direct access authorization. The attacker has the capability to run arbitrary code on the same machine and same logical core as the victim. This means the attacker can execute code with user-level privileges but does not have permission to access or modify privileged areas directly. The attacker can deploy processes that generate specific memory access patterns to train the prefetcher and observe the effects on shared hardware resources. This capability is crucial for manipulating the state of the prefetcher to extract information from the victim process. The key aspects of our threat model are described as follows:

a) Gadget Code Existence: We assume the existence of gadget code within the victim's process that can be exploited. Gadget code refers to small, useful pieces of code that can be leveraged by an attacker to perform unintended actions.

b) Co-residency: The attack process must reside on the same physical machine as the victim process. Co-residency allows the attacker to exploit shared hardware resources, such as the CPU cache and prefetcher, to infer the victim's confidential information.

V. Characterizing Page Prefetcher on Intel

In this section, we provide a comprehensive characterization study of the page prefetcher. We investigate the effects of the cross-page prefetching policy in the page prefetcher, demonstrating that the page prefetcher is a separate prefetcher alongside the IP-stride prefetcher, which can trace page-grained memory access patterns.

A. Page-Grained Prefetcher

AfterImage [6] demonstrates that the Intel Core processor does not support cross-page prefetching when the target prefetched page is missed in the TLB. To explore if the novel Xeon processors support cross-page prefetching, we designed a novel microbenchmark. The microbenchmark is shown in Listing 1.

```
1 void prime(int range, int range_2, int stride, int page) {
2   uint8_t *ptr = (uint8_t *)mmap(NULL, 4096 * 4096, ...);
3   for (int i = 0; i < range; i++)
4     flushAll(ptr + 1 * i * 4096, 0, 64); // flush pages
5   for(int i = 0; i < range_2; i++)
6     MEM(ptr + stride * i * 4096); // train the prefetcher
7   // test whether the prefetcher is effective
8   time(ptr[page * 4096]); }
```

Listing 1: Microbenchmark pseudo-code for detecting the cross-page prefetcher's effectiveness.

In the initial phase of our experiment, we allocate a memory pool consisting of 256 pages. To ensure each virtual page corresponds uniquely to a physical page, thereby mitigating the risk of page reclamation, we programmatically write a distinct value to the first byte of every page. To evaluate the prefetcher's accuracy and efficiency, we subsequently flush TLB and caches. This guarantees that subsequent accesses to these pages request the data from DRAM.

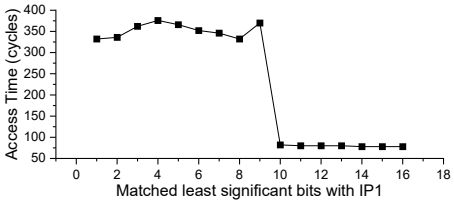


Fig. 3: Activation of the page prefetcher on IP2 following training with IP1.

We experiment to examine whether the L1 cache is loaded alongside the TLB entry in the page prefetcher. After flushing both the TLB and caches, we measure the page access time and subsequently train the prefetcher. We then evaluate the access time for a new page on the N+stride path. The page access time is 420± cycles after flushing, and it decreases to 60± cycles following the training of the prefetcher. This reduction in access time shows that the L1 cache entry for the new page is prefetched concurrently with the TLB entry.

Following this setup, we employ a series of controlled memory access patterns to scrutinize the effectiveness of the system's prefetching mechanisms. By varying the stride and the number of training iterations across the allocated pages, this experiment aims to evaluate the prefetcher's effectiveness in preemptively loading the necessary data into the cache. We found that a page prefetcher is introduced in Intel 3^{rd} and 4^{th} (Sapphire Rapids) Xeon processors, which can prefetch page-level information.

We use the same loop function shown in Listing 1 but different values for the train. We noted that the page prefetcher can prefetch up to 4 pages as the stride under page-granularity prefetching conditions, and it can achieve prefetching with both positive and negative strides.

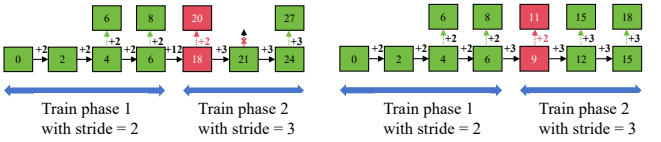
B. Indexing Policy of Page Prefetcher

Previous works [6] demonstrate that the traditional IP-stride prefetcher is indexed by the lower 8 bits of the instruction pointer. As the page prefetcher is also indexed by the IP, to determine if the indexing policy is the same as the IP-stride prefetcher, we use the same microbenchmark as described in AfterImage [6] to recover the indexing policy. Figure 3 demonstrates the test result. We note that IP2 initiates prefetching when it aligns with at least the lower 10 bits of IP1. This finding shows that the page prefetcher in the 3^{rd} Xeon processor is indexed by the lower 10 bits but not the lower 8 bits used by the IP-stride prefetcher.

C. Confidence and Stride Details

Using microbenchmarking, we note that the confidence has two bits and the threshold is 2, which is the same as the IP-stride prefetcher [6]. More concretely, we train the prefetcher using page-sized data offsets in this work, a stride of 2 means that the stride recorded in the page prefetcher has a length of 2×4096 bytes or 2 pages in total.

The microbenchmark used to reveal the confidence and stride update policy is designed as Listing 2. The microbenchmark employs $stride\_1$ to train the prefetcher for tr_1 iterations, and then $stride\_2$ is used to train the prefetcher for tr_2 iterations. Finally, the results from the benchmark allow us to determine whether the prefetcher's



(a) Two training phases with a random offset in between.

(b) The second training phase starts immediately after the first.

Fig. 4: Experimental results of the page prefetcher triggering mechanism. The square represents the n^{th} target page. The upper arrow indicates that a prefetch to the target page is triggered.

current stride is $stride\_1$ or $stride\_2$. The experimental results are shown in Figure 4.

```
void prime(int range, int stride_1, int stride_2){
  uint8_t *ptr = (uint8_t *)mmap(NULL, 4096 * 4096, ...);
  for (int i = 0; i < range; i++)
      flushAll(ptr + 1 * i * 4096, 0, 64); // flush pages
  // train the novel prefetcher
  for(int i = 0; i < tr_1; i++)
      MEM(ptr + stride_1 * i * 4096);
  flush(ptr);
  for(int i = 0; i < tr_2; i++)
      MEM(ptr + stride_2 * i * 4096);
  // test whether the stride is stride_2
  time(ptr[offset + stride_2 * 4096]);
  //test whether the stride is still stride_1
  time(ptr[offset + stride_1 * 4096]); }
```

Listing 2: Microbenchmark pseudo-code for detecting the confidence and stride updating policy of the page prefetcher.

In our experiment, as shown in Figure 4, $stride\_1$ and $stride\_2$ are configured to 2 and 3, respectively. It was observed that training the prefetcher with the same stride twice is sufficient to increase the confidence to the threshold, thereby triggering a prefetch request. For Xeon, the maximum stride could be 16,384 bytes (4×4096 bytes).

D. Page Prefetcher v.s. IP-stride Prefetcher

It seems that the page prefetcher is similar to the traditional IP-stride prefetcher (i.e., prefetching data with cache line granularity) in Intel processors. We can find that the IP-stride prefetcher leverages the least significant (LS) 8 bits of the IP to index instead of the LS 10 bits used by the page prefetcher.

E. When the Page Prefetcher be Activated?

As shown in Augury [32], the prefetcher in Apple's chips could be activated using only speculative accesses. To explore whether the page prefetcher can also be triggered during speculative execution, we conduct following experiments.

The testing gadget is presented in Listing 3. For the last i that equals $array\_size$, the branch prediction unit (BPU) for the condition within the loop has been trained to predict taken. A subsequent access causes a misprediction, which conducts the speculative execution of the load. If the prefetcher can be triggered via speculation only, the $(i+1)*stride$ will be prefetched into the cache. The result that the last line timing function shows lower latency indicates that the data has been prefetched to L1 after speculative execution, as we showed in Figure 5.

```
flush_all_mem(mem, sizeof(mem))
for (int i = 0; i <= array_size; i++)
  // i will be iter in 0, 1, ..., array_size
  // 0, 1, ..., array_size-1 will train BPU to be taken
  if (i < access_evicted_memory_containing(array_size))
    z = mem[i * stride]
timing(load(mem[array_size * stride]))
```

Listing 3: Trigger hardware data prefetcher even within the misprediction execution.

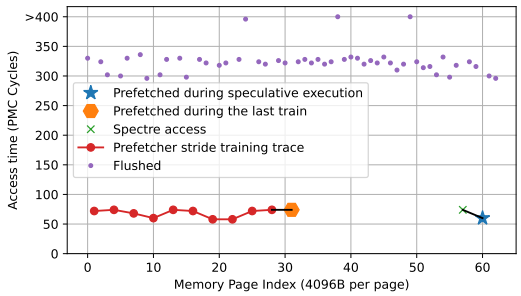


Fig. 5: Prefetcher could be triggered in both architectural execution and speculative accesses. An access time of less than 100 cycle indicates that a cache hit.

VI. Case Study

A. Exploiting with Spectre V1 Attack

Various defenses have been proposed at the hardware level [7], [8], [13], with a focus on achieving good performance and low cost than software mitigation such as using *lfence* memory barrier instruction. Among these, taint tracking of transient data loads and the elimination of secret-related cache side effects are notable. However, prefetcher side effects, triggered by cache access, are usually not considered in these defenses. If a prefetcher is maliciously trained, it may lead to defense failure. The residual effect of a stride prefetch (i.e., the prefetched index result minus the stride) can reveal secrets.

Considering that this kind of defense has not yet been deployed to the real hardware for the time being. For a convenient experiment, We used the *clflush* instruction to ensure that the direct load instructions' effect within speculative execution is eliminated. This simple method ensures that there are no cache side effects caused by loads within shared memory, rendering this Spectre V1 gadget unexploitable. In short, we can achieve an effect similar to the above defense mechanism. This could also apply to Spectre V2 [19], [33].

```
void victim_func(i) {
  if (i < access_evicted_memory_containing(array1_size))
    x = array1[i]; z = array2[x];
  flush(&array2[x]) }
for (int i = 0; i < array_size; i++)
  victim_func(i) // training spectre gadget
for (int i = 0; i < train_size; i++)
  stride_train(i) // training stride-prefetcher
// prepare cache state
flush_all_mem(array2, sizeof(array2))
// conduct Spectre V1 attack
victim_func(array1_size + secret_offset)
// retrieve attack results
timing(load(array2[potential_secret * stride]))
```

Listing 4: Using hardware data prefetcher to benefit the Spectre V1.

B. Exploiting as Covert Channel between Kernel and User

As we have verified that this prefetcher is not isolated between different privilege modes, we can exploit this characteristic to construct a covert channel between two security levels, such as between user and kernel modes. To create IP collisions, we can use IP matching, as proposed by previous work, to deduce the low 10-bit offset of the stride load instruction. Fortunately, the search space for this prefetcher is limited to only 1024 possibilities. If the IP probe is wrong, it will fail to trigger the prefetch, but it will not cause an exception.

In detail, if the sender wants to transmit a bit 1, it triggers multiple cache misses at one loading IP using a loop to train the page prefetcher. The receiver triggers the sender's gadget, executes the

collision IP, and probes if this IP has page-prefetched data to infer the secret information from this channel.

a) Evaluation: In the experimental evaluation on Xeon 8488C, we used this page prefetcher to send 100 random bits 3 times, as we showed one of them on Figure 6. Ultimately, we achieved an average transmission rate of 3332 bps (std: 219.2), with an error rate of about 8.9% (std: 2.2). Compared to previous work, such as AfterImage (833 bps, 6% error rate), this page prefetcher achieves significantly higher maximum bandwidth with similar error rates.

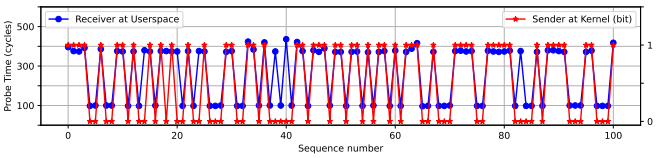


Fig. 6: The timing of sending "0" and sending "1" with PPA. Sending 100 random bits with a 6.9% error rate.

C. Exploiting the Kernel Gadget to Leak Secrets

Previous research like AfterImage has demonstrated that the IP-stride prefetcher could be exploited to leak secret-dependent branch information. We followed the same setting and used the IP matching method to create a low 10-bit collision between the user and kernel's load instruction.

Listing 5 shows the vulnerable kernel gadget. The attacker running in the userspace calls the syscall or other API that would execute the vulnerable kernel gadget. After the execution of that gadget is trained, the attacker flushes the data out of the cache and reloads the data to see in which stride the data will be prefetched to caches. Finally, using the prefetched stride result from the userspace execution, the attacker can infer whether the branch is taken or not to get the secret.

```
void vulnerable_kernel_function(int secret) {
  int stride = 1
  if (secret == 0x42) { stride = 2 }
  for (int i = 0; i < 10; i++)
    data[i] = arr[i * stride * page_size]
}
```

Listing 5: Vulnerable kernel secret-dependent branch.

D. Realistic Attack Targets Intel SGX

a) Victim Code within Intel SGX: The MbedTLS library, implementing the Montgomery-Ladder RSA [6], can be resident in the Intel SGX enclave [31] for better security. The enclave is a trusted execution environment that can protect the code and data from the untrusted host, even the privileged attacker. We use the MbedTLS shown in the previous work [6], [20] to demonstrate the attack.

b) Attack: We verify that the page prefetcher lacks isolation between the untrusted zone and the SGX enclave. For a realistic attack, the attacker first trains the page prefetcher with a load instruction that is aligned with the load within the secret-dependent branch. Then the attacker `ecall` into the enclave to trigger the secret-dependent branch. The attacker can use the SGX-Step [31] framework to trigger page faults and hijack the control flow of the victim for more precise control in single-stepping the victim code. After the enclave finishes the execution, the attacker can re-execute the load instruction and measure the timing of accessing the prefetched target to determine whether the prefetcher is triggered to infer whether the victim has executed that branch.

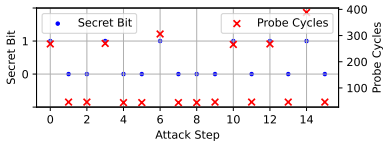


Fig. 7: Attack results of `modpow` within Intel SGX via secret-dependent branches (an 16-bit secret sequence, b'1001001000101010).

TABLE II: Summary of hardware data prefetcher attack.

Name	Target	What's New
Unveiling Prefetcher [29]	IP-Stride Prefetcher + L1/L2/LLC	First Side-Channel Attack via Prefetcher
Fetching Tale [11]	IP-Stride Prefetcher	Covert Channel
Augury [32]	Pointer-Chasing Prefetcher + L1/L2/LLC	Uncover New Prefetcher
AfterImage [6]	IP-Stride Prefetcher	Algorithm Agnostic
GoFetch [4]	Pointer-Chasing Prefetcher + L1/L2/LLC	Characterization & Exploration
Fetchbench [28]	Multiple Prefetchers	Systematic Review
PREFETCHX [5]	XPT Prefetcher	New Prefetcher
ShadowLoad [17]	Stride Prefetcher	New Exploitation
PPA (*This work*)	**Page Prefetcher**	**New Chara. & Explo.**

c) Evaluation: We evaluate the attack on an Intel SGX-enabled machine with an Intel Xeon 6438Y+ chip. We use the SGX SDK to build the enclave and the attacker thread. We use a similar victim gadget as introduced in the previous work [6] within Intel SGX and achieve a success rate of 98.76% in leaking 16-bit secret keys 5 times. We present the leakage results of one of the attacks in Figure 7.

VII. Mitigation Discussion

For Spectre attacks, some work focuses on removing touchable secrets, such as wide-scale deployment of policies like browser's site isolation [24]. A similarly idea, KPTI [15] removed virtual address mappings for the kernel from userspace processes. For PPA, **avoiding** memory accesses that are dependent on secrets could be practical.

In response to identified threats, various potential defenses have been proposed to mitigate the risks associated with prefetchers or related vulnerabilities. These defenses range from architectural modifications to software-based approaches aimed at detecting and preventing malicious exploitation of prefetchers. On the hardware side, some work proposes the elimination of secret-related cache side effects to prevent the information leakage [7], [8], [13]. For safety-critical applications, **disabling** the hardware prefetcher or **isolating** it between privilege boundaries may be an appropriate hardware mitigation.

VIII. Related Work

Extensive research has been conducted on reverse engineering hardware data prefetchers, including studies such as Unveiling Prefetcher [29], Fetching Tale [11], Augury [32], AfterImage [6], GoFetch [4], and PREFETCHX [5]. Each of these studies focuses on specific types of hardware prefetchers, such as IP-Stride Prefetchers and Data Memory-Dependent Prefetchers (DMPs). Due to the widespread deployment and unified model of the IP-Stride Prefetcher, FetchBench was proposed to avoid the need for manual analysis of each microarchitecture. We have summarized the research progress and the key contribution of each related work at Table II.

IX. Conclusion

In this work, we conduct an in-depth reverse-engineering analysis to characterize Intel's page prefetcher, a hardware data prefetcher related to the IP-stride prefetcher but capable of prefetching data at page-level strides. Our analysis revealed critical insights into the functionality and vulnerabilities of this prefetcher.

Open Science. Artifact is available at: github.com/THU-HAS/PPA.

Acknowledgements. This work was generously supported by the National Key Research and Development Program of China under Grant (2024YFB4405400), NSFC (No. U24A6009), Beijing Municipal Science and Technology Project (Nos.Z241100004224028), Beijing Natural Science Foundation (L247013), Guangdong Municipal Science and Technology Project (Nos. 2024QN11X196), BNRist. Furthermore, we gratefully thank the anonymous reviewers for their helpful comments.

References

[1] A. Ansari, F. Golshan, R. Barati, P. Lotfi-Kamran, and H. Sarbazi-Azad, "MANA: Microarchitecting a Temporal Instruction Prefetcher," pp. 1–1.

[2] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated load-address predictors," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, pp. 54–63.

[3] C. Carruth, "Speculative load hardening," Available at https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0 (2024/07/27).

[4] B. Chen, Y. Wang, P. Shome, C. W. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, "GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers," in *The 33rd USENIX Security Symposium (2024)*, 2024.

[5] Y. Chen, A. Hajiabadi, L. Pei, and T. E. Carlson, "PREFETCHX: Cross-Core Cache-Agnostic Prefetcher-based Side-Channel Attacks," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, pp. 395–408.

[6] Y. Chen, L. Pei, and T. E. Carlson, "Afterimage: Leaking control flow data and tracking load operations via the hardware prefetcher," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 16–32.

[7] X. Cheng, F. Tong, H. Wang, Z. Zhou, F. Jiang, and Y. Mao, "Speclfb: Eliminating cache side channels in speculative executions," in *33nd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024.

[8] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative privacy tracking (spt): Leaking information from speculative execution without compromising privacy," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 607–622.

[9] I. Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual," 2022.

[10] ——, "Intel 64 and ia-32 architectures optimization reference manual," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2023.

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

[12] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, pp. 152–162.

[13] J. Fustos, F. Farshchi, and H. Yun, "SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[14] B. R. Godala, S. P. Ramesh, G. A. Pokam, J. Stark, A. Seznec, D. Tullsen, and D. I. August, "PDIP: Priority Directed Instruction Prefetching."

[15] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: Long live kaslr," in *Engineering Secure Software and Systems*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2017, pp. 161–176.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[17] L. Hetterich, F. Thomas, L. Gerlach, R. Zhang, N. Bernsdorf, E. Ebert, and M. Schwarz, "Shadowload: Injecting state into hardware prefetchers," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1060–1075.

[18] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," vol. 48, no. 2, pp. 121–133, Feb./1999. [Online]. Available: http://ieeexplore.ieee.org/document/752653/

[19] 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 *S&P*, 2019.

[20] C. Liu, S. Feng, Y. Li, D. Wang, W. He, Y. Lyu, and T. E. Carlson, "Mdpeek: Breaking balanced branches in SGX with memory disambiguation unit side channels," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. New York, NY, USA: ACM, March 30–April 3 2025, p. 17.

[21] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[22] M. Luo, W. Xiong, G. Lee, Y. Li, X. Yang, A. Zhang, Y. Tian, H.-H. S. Lee, and G. E. Suh, "AutoCAT: Reinforcement Learning for Automated Exploration of Cache-Timing Attacks," in *29th Sympisum on High Performance Computer Architecture (HPCA)*, 2023.

[23] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*. Springer, 2006, pp. 1–20.

[24] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1661–1678.

[25] A. Rohan, B. Panda, and P. Agarwal, "Reverse engineering the stream prefetcher for profit," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020, pp. 682–687.

[26] A. Ros and A. Jimborean, "Wrong-Path-Aware Entangling Instruction Prefetcher," vol. 73, no. 2, pp. 548–559. [Online]. Available: https://ieeexplore.ieee.org/document/10337781

[27] G. Saileshwar, C. W. Fletcher, and M. Qureshi, "Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[28] T. Schlüter, A. Choudhari, L. Hetterich, L. Trampert, H. Nemati, A. Ibrahim, M. Schwarz, C. Rossow, and N. O. Tippenhauer, "Fetchbench: Systematic identification and characterization of proprietary prefetchers," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 975–989.

[29] 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 *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 131–145.

[30] A. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[31] J. Van Bulck and F. Piessens, "SGX-Step: An open-source framework for precise dissection and practical exploitation of Intel SGX enclaves," in *ACSAC 2023 Cybersecurity Artifacts Competition and Impact Award Finalist Short Paper*, Dec. 2023.

[32] 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 *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1491–1505.

[33] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida, "Inspectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2," in *USENIX Security*, 2024.

[34] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 15–26.

[35] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 503–516.