# AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher

Yun Chen*
School of Computing, National University of Singapore
Singapore
yun.chen@u.nus.edu

Lingfeng Pei*
School of Computing, National University of Singapore
Singapore
lfpei@nus.edu.sg

Trevor E. Carlson
School of Computing, National University of Singapore
Singapore
tcarlson@comp.nus.edu.sg

## ABSTRACT

Research into processor-based side-channels has seen both a large number and a large variety of disclosed vulnerabilities that can leak critical, private data to malicious attackers. While most previous works require speculative execution and the use of cache primitives to transmit data, our new approach, called AfterImage, requires neither, capitalizing on vulnerabilities in Intel's IP-stride prefetcher to both expose and transmit victim data. By training this prefetcher with attacker-known values, and watching for changes to the prefetcher state when execution returns to the attacker, it is now possible to monitor and leak critical data from a large number of common userspace applications and kernel routines without speculation and additional cache accesses. To demonstrate the novel capabilities of AfterImage, we (1) present proof-of-concept attacks that leak data across different isolation levels, (2) present an end-to-end attack that leaks an entire RSA key from a modern, timing-balanced algorithm, and also (3) show how AfterImage can significantly improve the effectiveness of other attacks, such as power side-channel attacks, by using this technique as a high-precision marker.

In addition to an extensive evaluation of these and other cache-based attacks, we also present a full reverse-engineering of the Intel IP-stride prefetcher which was required to enable AfterImage, and describe how AfterImage can be used as a covert channel. Finally, we present several mitigation techniques that can be used to block this side-channel on machines today. Taken together, this work explores a full set of techniques to utilize the prefetcher to leak previously protected information between different protection domains (SGX, kernel and other user spaces) and across many important applications, including security and non-security-related workloads.

## CCS CONCEPTS

• **Security and privacy** → **Security in hardware**; **Hardware reverse engineering**; • **Computer systems organization** → **Architectures**.

## KEYWORDS

side-channel attack, prefetcher, hardware security

**ACM Reference Format:**
Yun Chen, Lingfeng Pei, and Trevor E. Carlson. 2023. 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 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, Vancouver, Canada, 17 pages. https://doi.org/10.1145/3575693.3575719

*These authors contributed equally to this work.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
*ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada*
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575719

## 1 INTRODUCTION

Modern systems leverage a variety of methods to achieve high performance, from new instructions to microarchitectural enhancements. Unfortunately, many of the techniques designed to speed up processors have new microarchitectural side-channel attacks exposed as the unintended consequence, typically exemplified by caching [15, 22, 39, 47, 61, 71] and speculative execution [5, 31, 37, 58, 66]. Secret information such as browser activity, user data, or even encryption keys used by security libraries can be leaked through these side-channels.

Among the list of modern hardware enhancements, however, a deep investigation of the vulnerability of an extremely important hardware component today - the prefetcher - is still missing. Prefetchers work by preloading data into the cache before it is requested by the processor to mitigate the effects of extremely long DRAM load latencies. In the vast majority of current Intel processors, an Instruction Pointer-based stride (IP-stride) prefetcher can be found [16], as it is a small structure that can provide a significant performance benefit [3]. This prefetcher learns repetitive strides between load addresses requested by the same IP. When the prefetcher obtains sufficient confidence, it will prefetch the next address into the cache, which is the sum of the current address and the previously learned stride.

Previously discovered attacks that are related to the IP-stride prefetcher generally fall into two categories. Some works [12] build covert channels inside the processor where both the sender and receiver are controlled by an adversary. By periodically flushing out history entries of the prefetcher, the adversary can enable covert communication. Another technique [59] captures a special cache footprint after software execution and reveals secrets by analyzing the footprint variations. These works attempt to exploit the IP-stride prefetcher, however, have been either algorithm-specific or can only be used as a covert channel; they cannot be used as a broadly applicable side-channel for a large class of applications.

In this paper, we demonstrate a quick-to-train and highly applicable hardware prefetcher-based side-channel, called AfterImage,

that can be exploited in today's processors[1] to extract secret information across program and user-kernel boundaries. By using an attacker-based load instruction that targets a victim load, the attacker can mistrain the prefetcher with specific strides. This allows the attacker to infer critical information from the victim after it is run with a simple prefetcher status check.

In this work, we make the following contributions:

- We describe a new hardware prefetcher side-channel, AfterImage, that can track load instructions and is both algorithm-agnostic and independent of the use of traditional cache primitives for side-channel attacks.
- We demonstrate how two variants of AfterImage can leak control flow and branch-based secrets (a) across threads/process, and (b) across the user-kernel boundary, and can achieve a high attack success rate (from 91% to 99%). In addition, a covert channel based on a variant of AfterImage (variant 1) can achieve a transfer rate of 833 bps with an error rate of less than 6%.
- We provide an end-to-end cache-primitive independent attack against timing-constant RSA [33] and reveal the private key in only 188 minutes. We further show that using AfterImage can allow an attacker to precisely time when to perform power attacks against the OpenSSL RSA algorithm, and demonstrate that this information can improve the utility of the power attack.
- We present an in-depth study of the IP-stride prefetcher used in modern Intel processors with custom-designed microbenchmarks. Many features are revealed for the first time.
- We propose and analyze a lightweight mitigation strategy for future designs that shows an extremely small (0.7%) slowdown.

In the rest of this paper, we first provide the attack surface, threat model, and an overview of AfterImage (Section 2) and then provide background information in Section 3. We study the IP-stride prefetcher present in today's hardware in Section 4. Next, we present leaking information via AfterImage by our proof-of-concept variants, timing constant RSA attack, and locating load timing for improving power attacks in Section 5 and Section 6. The experimental setup and results are discussed in Section 7. Finally, we discuss the effectiveness of existing defenses in Section 8, outline related work in Section 9, and conclude in Section 10.

## 2 AFTERIMAGE OVERVIEW

### 2.1 Attack Surface

The attack surface of AfterImage consists of load instructions in either a user workload or in the kernel. The key targets are branch-dependent load instructions (an example is shown in Listing 1). With these instructions as a starting point, we will demonstrate how AfterImage can leak this secret information across different levels of isolation.

In the real world, branch instructions are one of the most commonly encountered instructions with as many as 17% of dynamic instructions executed in an application [8][10]. Private data, such as user input, or secret keys, are often used by branch instructions that

```
1 if (secret)
2     // Load on one path req'd
3     char temp0 = array[address];
4 else
5     // Optional
6     char temp1 = array[address];
```

**Listing 1: Example of branch-dependent load instructions**

control the execution of subsequent load instructions, which can be found in commonly-used applications that include encryption algorithms and the Linux kernel. For example, bluetooth connection activity [6] (see Figure 1) and battery properties [4] (see Figure 2) in the Linux kernel use user data to determine their control flow.

The timing-constant coding style, which has been proposed as a technique to hide the timing variation for different directions in a branch in many security libraries [43, 46, 65], can still have branches affected by AfterImage. For example, the timing-constant RSA encryption engine [65] (see Figure 3 and Figure 4) performs memory accesses both on the `if` and `else` path. Although they call the same function with different parameters to balance the timing, some load instructions will still be generated before the function call in different directions to prepare the essential information (e.g., function/pointer address) required by the function. It is worth noting that the number of load instructions in different directions remains constant, indicating that the algorithm stays timing-constant. Since the IPs of these `load` instructions are different, AfterImage can still distinguish the actual control flow and infer the secret.

Instead of branches, AfterImage also targets the timing information of load instructions, which can benefit other types of attacks. For instance, power analysis normally needs to get the accurate encryption/decryption time [30, 41] and samples the power consumption on time to extract the key. We will show that AfterImage can help track certain load operations and assist power attacks.

### 2.2 Threat Model

We consider that the attacker aims to track the execution of certain load instructions from a victim process or kernel context to leak secret information or perform additional attacks. We assume the victim runs on the same logical core as the attacker to share the hardware prefetcher. The attacker does not need OS capabilities obtained when running in an Intel SGX enclave, super-user access, or even socket communication ability with the victim, but does have user access to the system.

In the standalone attacking scenarios, the victim contains control-flow instructions, whose direction is determined by its own secret data, and at least one load instruction occurs under a branch (inside an `if()` or `else` block, for example).

In another scenario, i.e., the attacker would like to leverage AfterImage to track timing information of load operations from the victim, and then launch timing-sensitive attacks. We assume that the victim code can be disassembled by the attacker.

In both scenarios, we assume that memory pages requested by the load instructions of the victim should already be cached in the TLB. This occurs frequently in our test cases (evidenced by a high success rate) and is of high probability for streaming applications.

[1] AfterImage has been disclosed to Intel, and approved for distribution.

```
1  switch (pkt_type(skb)) {
2    case HCI_COMMAND_PKT:
3      hdev->stat.cmd_tx++;
4    case HCI_ACLDATA_PKT:
5      hdev->stat.acl_tx++;
6    case HCI_SCODATA_PKT:
7      hdev->stat.sco_tx++;
8  }
```

**Figure 1: Vulnerable code pattern in the Bluetooth connection activity source code [6].**

```
1  switch (prop) {
2  case PROP_ONLINE:
3    val->intval = 1;
4  case PROP_CAPACITY:
5    ...
6    val->intval = value;
7  case PROP_MODEL_NAME:
8    val->strval = dev;
9  case PROP_SCOPE:
10   val->intval = SUPPLY;
11 }
```

**Figure 2: Vulnerable code pattern in the source code of battery properties [4].**

```
1  for(i=0; i<len(key); i++)
2  {
3      if(key[i] & 1)
4         ...
5         multiply_add();
6         clflush();
7      else
8         ...
9         multiply_add();
10        clflush();
11 }
```

**Figure 3: Vulnerable code pattern in Montgomery-Ladder RSA engine [33].**

```
1  if(secret)
2  {
3      // normal execution
4      X->s = s;
5  }
6  else
7  {
8      // preventing RSA
9      // timing attack
10     X->s = -s;
11 }
```

**Figure 4: Vulnerable code pattern in timing-constant RSA engine [65].**

Address Space Layout Randomization (ASLR) and kernel ASLR (KASLR) can be enabled to further improve the security of the system but will not affect AfterImage.

## 2.3 AfterImage Workflow

There are four main steps needed to leak information via this prefetcher side-channel.

**Prepare:** The attacker locates load operations in the victim context to track, and generates a local version of the targeted load instructions. These `loads` masquerade as the target `loads` and share the same hardware entry in the prefetcher. In this work, we present two techniques to locate the load instructions in the victim. First, if the binary of the victim is available to the attacker, such as a shared encryption library, the attacker can use disassembly tools such as `objdump`. Second, based on our reverse-engineering results that will be presented in Section 4, the IP-stride prefetcher is indexed only with the least significant 8-bits of IP. We further propose an IP searching technique; the details are discussed in Section 5.2.

**Train Prefetcher:** The attacker then trains the IP-stride prefetcher locally by executing a strided address sequence to obtain a sufficient level of confidence in the prefetcher. Our training is again based on our reverse-engineering results.

**Trigger Prefetcher:** When the victim executes the targeted code region, the prefetcher will be automatically triggered with the previously trained stride.

**Observe Secret:** We present two techniques to observe secret data. First, similar to previously discovered microarchitectural side-channels, we use traditional cache primitives, including Flush+Reload [71] and Prime+Probe [47, 50] to expose the strides from the cache lines (AfterImage-Cache). They provide higher accuracy in demonstrating the secret leakage, as we will show in Section 5. The second technique is to check the prefetcher's status (AfterImage-PSC), which makes this attack standalone. Based on our reverse-engineering results, the prefetcher will exhibit different statuses depending on the victim. We give a detailed example in Section 6.

# 3 BACKGROUND

## 3.1 Cache Timing Side-Channel Attacks

Cache side-channel primitives, such as Prime+Probe [47, 50], Flush+Reload [71], Flush+Flush [22], and Prime+Abort [15] are often used as the basis for modern hardware attacks today[2] [59, 66, 67, 69].

Flush+Reload is one of the most common cache primitives that takes advantage of shared memory between different processes. The attacker first **flushes** the shared memory from cache into DRAM. After the victim performs its normal execution, the attacker then **reloads** the shared memory and observes the timing differences, when addresses that hit in the cache indicate accesses by the victim.

The Prime+Probe cache side-channel does not require the existence of shared memory. The attacker **primes** the cache sets with its own data, and **probes** whether these cache sets are still occupied after the victim program has been scheduled. Prime+Probe is more general than Flush+Reload but it is less noise-resilient, since any activity in the system can evict the priming data as well.

Priming the complete LLC cache might not be easy to achieve by accessing a chunk of data whose size is larger than the LLC[3]. This is because most recent microarchitectures divide the LLC into smaller **slices** using a hash function to reduce contention and those hash functions are not often publicly known [29, 61, 68]. The **eviction set**, as a formal term of priming data, is a collection of addresses that map to the same cache set and slice that guarantees its complete eviction [29, 61, 68]. A minimal eviction set (MES) has a number of elements equal to the cache associativity. Eviction sets need to be correctly built to allow Prime+Probe to be performed.

## 3.2 Prefetchers in Intel Microprocessors

Intel has described four hardware prefetchers in their processor designs [16]. The data cache unit (DCU) prefetcher, also known as the next-line prefetcher [60], attempts to automatically prefetch a single, subsequent cache line. Data prefetch logic (DPL), i.e., the adjacent prefetcher, regards data as 128-byte aligned blocks. A cache miss to one of the two cache lines in this block will trigger a prefetch to the pair line. The Streamer prefetcher records sequential positive and negative offset streams and prefetches the next or previous several cache lines based on the system status (e.g., bandwidth, streaming direction), respectively. Therefore, the operation of these three hardware prefetchers does not have as much flexibility as the Instruction Pointer (IP)-based strided prefetcher, i.e. the IP-stride prefetcher.

The basic structure of the IP-stride prefetcher is shown in Figure 5. This prefetcher keeps track of `load` instructions with regular strides from the same IP. Its operation is composed of three steps.

[2]Cache side-channels cannot leak the control-flow information unless the actually requested addresses are known to the attacker beforehand.

[3]Although L1 and L2 cache are not indexed with such nonpublic hash functions, we have empirically discovered that priming L1 or L2 cache does not provide a distinguishable latency gap. What's worse, the existence of the line fill buffer between the L1 and L2 cache can add significant uncertainty to the measured timing.

**1. Index and Replace.** When a `load` instruction is present, it will be indexed into an entry with the same IP tag. If no such IP tag exists, a victim entry will be selected and evicted. **2. Update.** If the difference between the current address and the *Last Addr* is equal to the $Stride$, the $Confidence$ will be increased, otherwise it will be decreased. However, if the $Confidence$ drops below a threshold, the $Stride$ value will be updated to a new stride as well. **3. Prefetch.** If the $Confidence$ exceeds a certain threshold, a prefetch request will be sent to the next address which is the sum of the current access address and the recorded $Stride$ value.

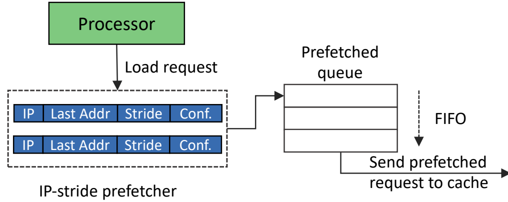



**Figure 5: General architecture of the IP-stride prefetcher.**

# 4 REVEALING INTEL'S IP-STRIDE PREFETCHER

Based on the documentation of Intel's Sandy Bridge microarchitecture [16], prior works [12, 59, 69] have disclosed some basic characteristics such as strided prefetching and process sharing through reverse engineering the IP-stride prefetcher. However, in this work, we take an additional step to reverse-engineer the major components of the IP-stride prefetcher in the Haswell and Coffee Lake microarchitectures. To the best of our knowledge, this is the first work to reveal the index, update and trigger mechanisms in detail. We additionally investigate the effects of cross-page address prefetching, determine the number of entries in the history table, and reverse engineer the IP-stride prefetcher's replacement policy.

## 4.1 Indexing into the IP-stride Prefetcher

Since older generations of Intel processors are indexed with the least significant 8-bits of the load instruction address, we first aim to verify whether this is still true in newer processor generations. Moreover, we would like to investigate whether any other factors should be taken into account during indexing.

```
void idx_detect_train(int stride, int train)
{
    for(int i = 0; i < train; i++)
    {
IP_1:       int temp0 = array[i * stride];
    }
    // Not shown: add IP offset using NOPs
IP_2:int temp1 = array[r];
}
```

**Listing 2: Microbenchmark pseudo-code for detecting the indexing mechanism of the IP-stride prefetcher.**

We use a microbenchmark, similar to that shown in Listing 2, which first trains the IP-stride prefetcher using `IP_1` with a constant multiple cache line-sized stride and then accesses the $r$-th

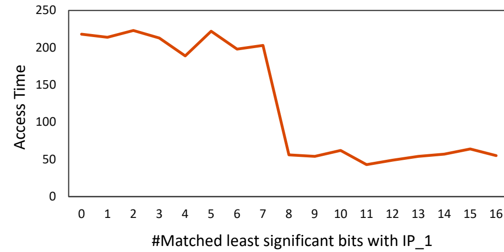



**Figure 6: The Coffee Lake IP-stride prefetcher triggering a result on `IP_2` when trained with `IP_1`. Note that an *Access Time* higher than 120 cycles means that the prefetcher has not been triggered to prefetch the address into cache.**

cache line in the *array* with `IP_2`. `IP_2` is also offset such that the least significant n-bits match those of `IP_1`. If the load at `IP_2` can activate the prefetcher to prefetch *array[r + stride]*, i.e. map to the same entry with `IP_1`, we conclude that the indexing of the IP-stride prefetcher is dependent on these n-bits of IP.

Figure 6 shows that the `IP_2` `load` can trigger the prefetcher if its lowest 8-bits are the same as that of `IP_1`, confirming our understanding that the IP-stride prefetcher uses **the least significant 8-bits** to index the entry. Furthermore, this example verifies that the IP-stride prefetcher lacks a tag field to verify the full IP.

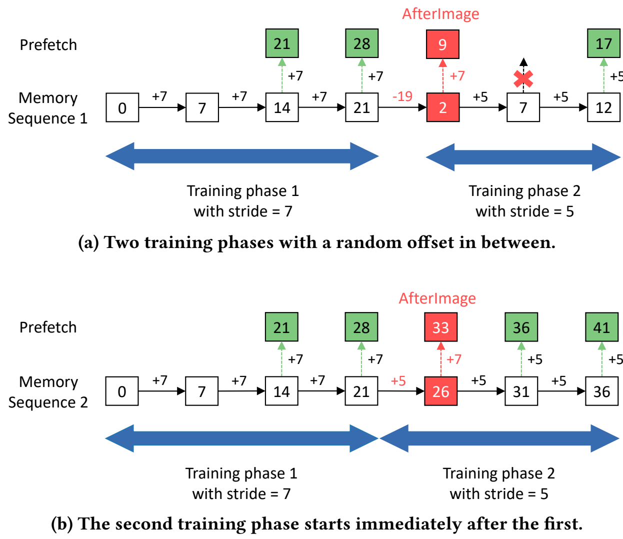
```
void const_ip_load(int index, void* array)
{
IP_1:     int temp = array[index];
}

void policy_cs(int st_1, int st_2, int tr_1, int tr_2)
{
    char* array = mmap(4096, ...);
    int i, offset;
    for(i = 0; i < tr_1; i++)
        const_ip_load(i * st_1, array);
    flush(array);
    for(i = 0; i < tr_2; i++)
        const_ip_load(offset + i * st_2, array);
    // test whether the stride now is updated to st_2
    time(array[i * st_2]);
    // test whether the stride now is still st_1
    time(array[offset + (i -1 ) * st_2 + st_1]);
}
```

**Listing 3: Microbenchmark pseudo-code for detecting the confidence and stride updating policy of the IP-stride prefetcher.**

## 4.2 Confidence and Stride Details

According to our basic test, the **`confidence`** has two bits and the threshold is 2. We use the same `for` loop shown in Listing 2 but different values for the `train`. After training, we check *array[i*stride]* to see if it is cached. The **`stride`** has (1+12) bits, with the most significant bit being used to differentiate between negative and positive strides, while the other twelve bits reflect the maximum stride, which cannot be more than 2KiB (1 << 12). It should be noted that the stride of Intel's IP-stride prefetcher does not need to align to a cache line [28]. As a result, the prefetcher requires up to 13 bits to deliver the stride. However, because we train the prefetcher using cache-line-sized data offsets in this work, a stride




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

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

**Figure 7: Experimental results of the IP-stride prefetcher triggering mechanism on Coffee Lake.**

of 7 means that the stride recorded in the IP-stride prefetcher has a length of $7 \times 64$ bytes, or 7 cache lines in total.

After determining the prefetcher's supported `confidence` and `stride` values, we use a microbenchmark (Listing 3) to reverse-engineer the confidence and stride **update policy**. The microbenchmark first trains the `IP_1` *tr_1* times (*tr_1* > 2 to guarantee the confidence is equal or larger than the threshold) with stride *st_1*, and then uses a new stride (*st_2*) to train the prefetcher *tr_2* times. Finally, the results from the microbenchmark run allow us to determine which stride is currently being used in the prefetcher, with the results listed in Figure 7.

In our experiment, *st_1* and *st_2* are set as 7 and 5, respectively. Normally, a stride of *st_1* will be triggered even after the first iteration of the second loop. This means that *regardless of whether the new stride is identical to the recorded stride, if the confidence reaches the required threshold, the prefetcher always issues a new prefetch request.* We refer to this behavior as the **key** component of AfterImage. This allows for the triggering of the prefetcher to occur unconditionally, allowing the result to appear in a separate execution context.

For the second iteration of the second loop, no matter how large the value of *tr_1* is, neither *st_1* nor *st_2* is triggered. When we get to the third iteration, the *st_2* is finally active. However, we discover that if we set the `offset` directly to the stride of the second phase, i.e. start the second training earlier, the prefetcher will become fully trained at the second iteration. These results imply that the `stride` will always be updated as *current address* − *last address*, and once the newly computed `stride` is different from the previous stride, the `confidence` will be reset to 1 at the same time. Therefore, for a `load` instruction with an IP and a request to *current address*, the workflow inside the prefetcher is shown in Algorithm 1.

**Algorithm 1:** Confidence and stride updating policy and triggering strategy of the IP-stride prefetcher.

```
   Data: IP, current_address
 1 if IP tag existed in the history table then
 2     distance = current address − last address
 3     if confidence ≥ 2 then
 4         Prefetch current address + stride
 5         if distance != stride then
 6             stride = distance confidence = 1
 7         else
 8             if confidence != 3 then
 9                 confidence += 1
10             end
11         end
12     else
13         if distance != stride then
14             stride = distance
15             confidence = 1
16         else
17             confidence += 1
18             if confidence == 2 then
19                 Prefetch current address + stride
20             end
21         end
22     end
23 else
24     Create_New_Entry(IP, confidence = 0, stride = 0)
25 end
```

## 4.3 Page Boundary Checking

The benchmark shown in Listing 4 first `MMAP` two memory pools, named `recl_array` and `lock_array`. `recl_array` will be a resource-saving pool that automatically reclaims used physical page frames. The `lock_array` is allocated with `MAP_LOCKED`, which will always lock the page frame. We leverage `IP_1` and `IP_2` to train the prefetcher with a given stride on one page (e.g., *p*-th page), and then access the next *offset*-th page, and verify whether the target address (i.e., *recl_/lock_array[p + offset + stride])* is in the cache or not.

Table 1 shows the result of this experiment. The first column is the virtual address offset between the testing page and the training page. The second column indicates whether these testing pages have the same physical address as the training page or not. The last column represents the testing results, i.e. successfully prefetched or not. We find that even though the destination address in `recl_array` spans several logical page boundaries, the prefetcher does not invalidate the entry `IP_1` and they are all successfully triggered. If the physical page frame is crossed, the prefetcher may invalidate the entry and re-learn the stride and confidence. More specifically, if the newly accessed page (e.g., *(p+1)*-th page) misses in the TLB, the first access for this page will create the page table entry and will not impact the prefetcher status (e.g., decrease the confidence). The second memory access on the *(p+1)*-th page then can directly activate the prefetcher to prefetch *current address + stride*. If the page mapping hits in TLB, the prefetcher will be activated immediately.

**Table 1: The Coffee Lake IP-stride prefetcher triggering results on different logic pages and physical page frames.**

| Virtual Addr Offset | Share Physical Page | | Prefetchable | |
|---|---|---|---|---|
| | recl | lock | recl | lock |
| 1 Page | ✓ | ✗ | ✓ | ✓ |
| 2 Page | ✓ | ✗ | ✓ | ✗ |
| 3 Page | ✓ | ✗ | ✓ | ✗ |
| 4 Page | ✓ | ✗ | ✓ | ✗ |

We also notice that the next page of the trained page sometimes can trigger the prefetcher at the first memory access even if we never accessed it before. We infer that the next page is special because of the use of the next-page prefetcher that was introduced in the Haswell microarchitecture [26].

As a result, the prefetcher uses the page frame to determine whether the new address crosses the page boundary and processes the next page separately.

```
1  void two_ip_loads(int index, void* array1, void* array2)
2  {
3  IP_1:     int temp0 = array1[index];
4  IP_2:     int temp1 = array2[index];
5  }
6
7  void page_policy(int offset, int stride)
8  {
9      char* recl_array = mmap(n * 4096, ...);
10     char* lock_array = mmap(n * 4096, MAP_LOCKED, ...);
11     // do not cross page
12     for(int i = 0; i < 4; i++)
13         two_ip_loads(i * stride, recl_array, lock_array);
14
15     two_ip_loads(offset, recl_array, lock_array);
16     time(recl_array[offset + stride]);
17     time(lock_array[offset + stride]);
18 }
```

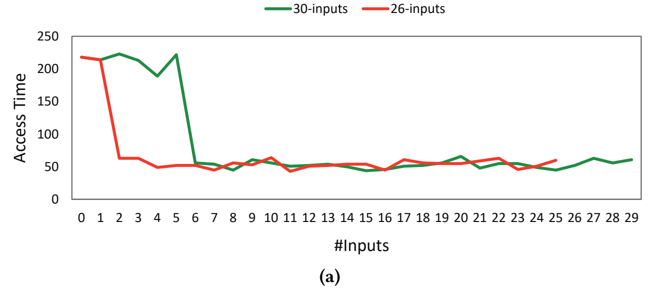
**Listing 4: Microbenchmark pseudo-code for detecting the page checking strategy of the IP-stride prefetcher.**

## 4.4 Number of Entries

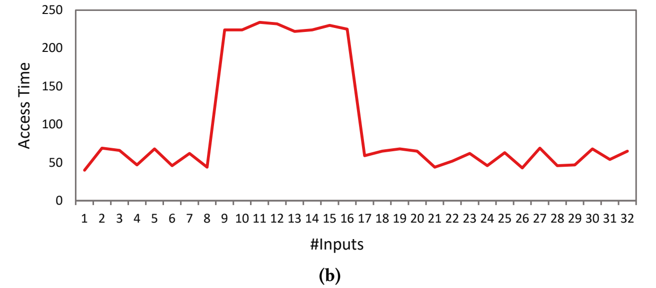
We construct a microbenchmark (See Listing 5) that executes a loop with a varying number of `load` instructions. Every `load`'s

```
1  void n_ip_loads(int index, int N, void* array)
2  {
3  IP_1:     int temp1 = array[4096 + index];
4   ...                 ...
5  IP_N:     int tempN = array[4096*N + index];
6  }
7
8  void num_entry(int N, int stride, int offset)
9  {
10     //contains N pages
11     char* array = mmap(N * 4096, MAP_LOCKED, ...);
12     for(int i = 0; i < 5; i++)
13         n_ip_loads(i * stride, N, array);
14
15     n_ip_loads(offset, N, array);
16     for(int i = 0; i < N; i++)
17         time(array[4096*i + offset + stride]);
18 }
```

**Listing 5: Microbenchmark pseudo-code for determining number of entries of the IP-stride prefetcher.**




(a)




(b)

**Figure 8: Reverse-engineering results on Coffee Lake: (a) The IP-stride prefetcher triggering results for 26 IPs and 30 IPs to determine the prefetcher's number of entries. (b) The IP-stride prefetcher triggering result for 32 IPs, with 8-16th IPs revisited, to demonstrate the prefetcher's replacement policy.**

least significant 8-bits of its IP are unique, and their data access patterns are constant. After training each of these IPs on different page frames (to avoid false positives), we re-access them and test the access time to $page_i[offset + stride]$ to determine if they can still activate the prefetcher.

The experimental result is shown in Figure 8a. Specific IPs are no longer able to trigger the prefetcher. More specifically, if the number of test IPs is 26, the first two IPs will no longer be able to activate the prefetcher. If the number of test IPs is 30, the first six IPs cannot trigger the prefetcher. As a result of the prefetcher's restricted size, some IPs get evicted. Thus, the number of prefetcher entries is the number of IPs that can still activate the prefetcher after training all of them, which is 24, in our case.

## 4.5 Prefetcher Replacement Policy

As we only see the least recent IPs being evicted, to determine whether the prefetcher's replacement policy is First-In-First-Out (FIFO) or least recently used (LRU), we update the number of entries detection microbenchmark by adding a number of `jmp` instructions. The number of test IPs is increased to 32 and 32 page frames are allocated for the training of each IP. The first 24 IPs will be trained on various page frames to occupy the whole table, and then the caches will be flushed. Next, the first 8 IPs will be re-trained to update them to a more recently used position. Following that, we train another 8 new IPs to evict some entries and flush the cache once again. Finally, we execute these 32 load instructions again to read a random cache line $L$ in the corresponding page and see

```
for(int i = 0; i < 3; i ++)
{
    IP offset1
    // to match if-path
    int temp0 = array[i * S1];
    IP offset2
    // to match else-path
    int temp1 = array[i * S2];
}
```

**Listing 6: Gadget used in AfterImage. The use of different strides (S1 and S2) allows the attacker to differentiate the two cases.**

if the $(L + stride)$-th cache line is prefetched. The first eight IPs should have been evicted if the prefetcher uses a FIFO policy. If not, these IP addresses should still be able to trigger prefetching. The experimental result is shown in Figure 8b. We observe that the evicted IPs are between the 9th and 16th position, indicating that the IP-stride prefetcher is using a form of the LRU replacement strategy. In addition, because the replacements have always been contiguous, it follows that it will most likely not use a tree-based pseudo-LRU (PLRU) replacement policy. Further, as a true LRU implementation can be expensive to implement in hardware, we suspect that the hardware is implementing a Bit-PLRU-based replacement policy.

### 4.6 Interplay with SGX

To test how AfterImage interacts with SGX, we pass a memory region into SGX and let an in-enclave thread access it with a certain pattern to see if SGX can trigger the prefetcher and if the prefetched data will still be validated once the SGX is switched out. To verify this assumption, We access the prefetched cache line in the untrusted zone and measure the access time. The result reveals that we always get a cache hit for the prefetched cache line, proving our hypothesis.

## 5 LEAKING BRANCH SECRET DATA VIA AFTERIMAGE

In this section, we describe how AfterImage leaks control flow and secret-dependent branch information from a victim to the attacker via an AfterImage-Cache flow. We assume that the victim has a branch that contains load instructions, whose direction is determined by its private secret. The attacker aims to leak this secret across different isolation layers, including threads, processes, and the user-kernel boundary, each more critical than the previous one. To demonstrate this, we present two proof-of-concept variants built on top of different cache primitives to show the effectiveness and accuracy of AfterImage in the ideal case.

### 5.1 Variant 1

**Observation 1**: The IP-stride prefetcher trained by IP1 can be triggered by IP2, even when a new stride is seen, or across threads or processes that operate on the same physical core, as long as the least significant 8-bits of IP2 match those of IP1.

Based on this observation, we build AfterImage variant 1, presented in Figure 9, to demonstrate that the control flow of the victim from different threads/processes can be leaked through this side-channel. In the first case, the attacker is in the same address space as the victim and can execute a code gadget in a sandboxed manner. This assumption is also widely exploited in previous attacks [14, 31]. In the second, i.e., cross-processes case, the attacker and the victim are in separate subprocesses and different address spaces and we will show that the leakage is still successful.

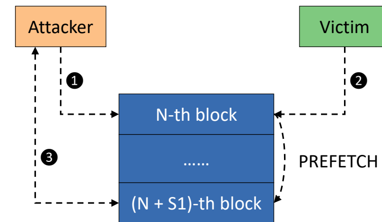



**Figure 9: AfterImage-Cache: Mistrain the IP-stride prefetcher from another thread/process and extract the secret via cache primitives.**

We first design a `gadget`, shown in Listing 6, that consists of two `load` instructions with different IPs. The least significant 8-bits of IPs of these two `load` instructions are specially tailored to match the memory access instructions in the if-path and else-path in the victim's code region, respectively, i.e., line 2 and line 4 in Listing 1, corresponding to the different control flow of the victim.

The first secret extraction technique utilized by AfterImage-Cache is Flush+Reload. The attacker will first mistrain the prefetcher by executing the `gadget` with two constantly strided memory address sequences for two `load` instructions, e.g., stride $S1$ and $S2$ (step ❶ in Figure 9). Therefore, two entries will be pushed into the IP-stride prefetcher for these two IPs and their confidence counters for these strides will be saturated. The flush stage happens before the execution of the victim to prepare the cache status. The shared pages are flushed out from cache using the `clflush` instruction. If the victim process then executes the secret-dependent branch, as the lowest 8-bits of secret-dependent memory instructions' IPs will be indexed to an entry that is trained by the attacker, the IP-stride prefetcher will prefetch the cache line from *current address* + *stride* into the cache (step ❷ in Figure 9). After the victim executes the targeted branch, the attacker can observe the cache by reloading used pages cache-line-by-cache-line to check for the existence of the pre-determined strides among the cached lines (step ❸ in Figure 9) ($S1$ exists in the example).

Moreover, to make this attack more general without the need of shared memory, we further successfully utilize Prime+Probe to extract secrets in the cross-thread demonstration. The eviction sets (ESs), which are essentially sets of memory addresses, are computed beforehand because they are machine-determined. Similar to Flush+Reload, in the prime stage, the attacker accesses the eviction sets within its local memory space, and in the probe stage, the attacker traverses the ESs again to check for a high accessing latency, which indicates a victim's access. However, for cross-process demonstration, we empirically found that Prime+Probe suffers from

```
1  void vulnerable_syscall1(void* memory_space)
2  {
3      int num = random();
4      if(num)
5      {
6          char *address = get_address(memory_space);
7          Memory[address];
8      }
9      return 0;
10 }
```

**Listing 7: The customized kernel function.**

heavy noise interference from the intensive memory activities during context switches, i.e., over half of the MESs are touched by the system.

## 5.2 Variant 2

**Observation 2**: When a process switches between user and kernel privilege modes, the trained entries of the IP-stride prefetcher are retained.

The strict isolation between kernel and user mode protects privileged hardware and system status from being exposed to users. Sensitive information in the kernel represents the data that should not be visible to an arbitrary user, e.g., tokens, passwords, and encryption keys. This variant demonstrates how the IP-stride prefetcher can bridge the isolation gap between kernel and user space which leads to a side-channel that can potentially leak kernel secrets.

To demonstrate the feasibility, we build a kernel function as the target code pattern as a straightforward example in Listing 7. In this function, *num* represents the secret in the kernel and determines the *if* branch, in which a `load` instruction is followed. The syscall function shares memory with the user via a *memory_space* parameter that allows Flush+Reload to take place. Shared memory is more common in the kernel than in other processes. For example, the Linux kernel itself already has privileges to access memory pages of the user process, such as the copy_from_user() kernel function [35].

To establish this kernel-user side-channel, we first create and propose a new microarchitectural component, **IP matching**[4]. Since IPs of system call functions are normally unknown to the user or hard to determine, we cannot directly deduce the `offset` in the `gadget`. In this case, we designed an IP search method to create the correctly matched IP as the training object, which has the same function as the `gadget`. This IP should have the same least significant 8-bits with the load instruction of the syscall function, which fortunately shrinks the searching space to 256 possibilities. Due to the hardware capacity limitation, we search for the IP in groups, 24 IPs as a group to fit the size of the IP-stride prefetcher as indicated in Section 4. One group will be trained simultaneously with the same stride on different pages. When the correct group gets trained, the syscall can trigger a strided footprint in the cache if it executes the load instruction. The target IP will be the intersection of these groups. This process can be repeated multiple times until the IP is found in case of too many *not taken* branches during the testing.

**Attacking**. After the target IP is well-trained in the aforementioned phase, the attacker calls `clflush` instruction to flush the shared data out of cache. Then it calls this syscall and passes the control right to the kernel. If the branch in the kernel is taken, the following load instructions will be executed on the same shared memory space. The IP-stride prefetcher automatically checks its history table and finds a matched entry with a high confidence value. Therefore, it will send a prefetch request to the next address, which is *current address + stride*. When the syscall service is finished, the process goes back to the user state. The attacker reloads the data to see which addresses are cached. If two addresses with our selected stride are both hits, we can infer that this branch has been *taken* by the kernel, and vice versa.

[4]This IP matching method is general when the victim's IP is difficult to be accessed. It completely runs as a normal private function without touching any data to which it is not privileged. Especially for kernel functions, once the system is booted, the IPs of instructions will not be changed. In addition, as the ASLR or KASLR on Linux has at least a granularity on one page (assuming a page size of 4KiB), these techniques will not change the least significant 12 bits of the IPs. Since the IP-stride prefetcher uses the lowest 8-bits to index its history, ASLR does not impact IP matching.

```
1  void sgx_magic(void *pms)
2  {
3      ...
4      // copy memory from untrusted zone
5      volatile int *_tmp_arr = pms->arr;
6      volatile int *_in_arr;
7      volatile int stride;
8      volatile int secret;
9      if (secret)
10         stride = 3;
11     else
12         stride = 5;
13     for(i = 0; i < 8; i++)
14     {
15         _in_arr = _tmp_arr[i * stride * 64];
16     }
17     ...
18 }
```

**Listing 8: The example vulnerable code segment for AfterImage SGX side channel**

## 5.3 Covert Channel

With some minor modifications, AfterImage could also be harnessed as a cross-process covert channel to allow a sender and a spy to communicate.

We use the stride value as the covert information that is transferred after each round[5]. Once the sender finishes training the prefetcher, the spy then accesses one cache line on the shared page. The stride (i.e., secret) can be observed by computing the distance of cache lines that hit the cache.

## 5.4 Attacking SGX

To enable AfterImage for use as a side channel, to leak the control flow information from the enclave to the untrusted zone, the enclave is considered to include the code region shown in Listing 8 in the PoC. In the enclave, the stride is set based on the secret. In the untrusted zone, we will detect if the expected prefetched cache lines are in the cache or not. For example, if we find (5x8)-th cache line in the cache, we will know that the stride is set to 5 and then reveal that the secret is 1. The whole workflow is shown in Figure 10.

[5]Due to hardware limitations, the stride can not exceed 2KiB, and can encode any 5-bits of the secret into a cache line granularity. Although the sender can theoretically transmit up to 12-bits by training the prefetcher with cache line unaligned stride values, the spy usually reloads the memory at a cache line granularity and thus cannot observe the least significant 6-bits of transmitted data.

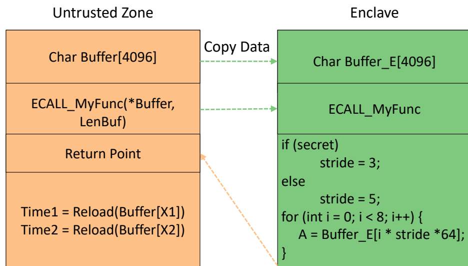



**Figure 10: SGX side/covert channel workflow**

If we assume that the in-enclave thread is a sender and would like to transmit a secret from SGX to the untrusted zone, the side channel can also be used as a covert channel to achieve this target. More concretely, the in-enclave thread can train the prefetcher with two alternative strides to represent 1 or 0. The receiver in the untrusted zone can access the prefetched cache line to determine if the relevant stride (i.e., X1 or X2 in Figure 10) is triggered or not.

## 6 ATTACKING REAL WORLD APPLICATIONS

In this section, we present AfterImage-PSC flow with two real world attacking examples. The AfterImage-PSC is built on our proposed Prefetcher Status Checking (PSC) method to extract secrets instead of building a cache side-channel basis, which makes this attack standalone and different from previous cache primitive dependent attacks. We first present an end-to-end attacking example against the timing-constant RSA algorithm, which contains load instructions in different control flows to balance the observable timing variation. Next, we show how AfterImage can track load instructions in the OpenSSL library and leak critical timing information.

### 6.1 Prefetcher Status Checking

To overcome the limitations of cache primitives (e.g., microarchitectural defenses, longer detection times), we propose a new secret extraction method that directly exploits the hardware prefetcher's features, as detailed in Section 4.2, called the **Prefetcher Status Checking (PSC) methodology**. It comes from the fact that, according to our reverse engineering results, after the victim executes a well-trained IP in the prefetcher (step ❶ and ❷ in Figure 11), the IP's confidence will be updated immediately, and that IP will no longer be able to trigger the prefetcher. The attacker can re-execute the targeted IPs to determine which ones are no longer triggerable and infer the victim's execution information (step ❸ in Figure 11). In this detection method, the attacker only needs to test the latency of a single destination address, which makes it faster than Flush+Reload or Prime+Probe, and it can bypass detection methods focused on cache primitives as well.

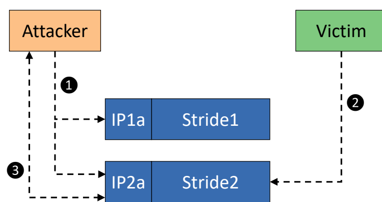



**Figure 11: AfterImage-PSC: Mistrain the IP-stride prefetcher from another thread/process and extract secret via checking prefetcher status.**

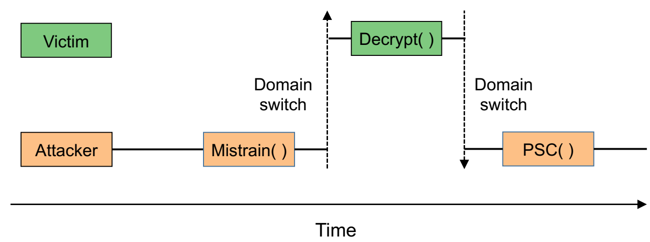



**Figure 12: Overview of attacking timing-constant RSA. After the attacker mistrains the prefetcher, the victim decrypts the message and the attacker then can check the prefetcher status.**

### 6.2 Leaking Private Key from Timing-Constant RSA

The Montgomery-Ladder RSA [33], used in MbedTLS, is a real-world application that we target with AfterImage-PSC. The IP in the `if` path (see Figure 3) is obtained by the disassembly tool `objdump`. Next, we construct an attacker thread that repeatedly trains and triggers the IP-stride prefetcher in his/her **own memory region**. The execution flow is shown in Figure 12. The prefetcher is mistrained with *stride* using the same gadget described in Listing 6, and the IP is manually set to have the same least-significant 8 bits with the target one. After training, the attacker thread calls a `sched_yield()` system call to give back control of the CPU and allow the victim thread to run and decrypt the ciphertext. When the branch is finished, the attacker follows the AfterImage-PSC flow to re-execute the load to address *addr* and measure the timing of accessing *addr* + *stride* to see whether the prefetcher is still triggered. A cache hit indicates that the victim executed the `else` path and the load in the `if` path is not touched, and vice versa. In our demonstration, the victim thread also calls `sched_yield()` after the branch to allow the detection of the attacker. This method is often used as a simplified synchronization technique in many attacks [23, 66] and has been demonstrated to not be a critical limitation and can be dealt with by a Linux scheduler attack [23]. Other synchronization with the victim may also be feasible such as simultaneous multithreading (SMT), accurate time-multiplexing, interrupt, or debug signal [51, 59, 66].

### 6.3 Tracking Timing of Load Instructions from OpenSSL

As AfterImage can track load operations across different contexts, we can show that this information can be used for other purposes other than extracting secrets directly, such as assisting other attacks. For instance, power attacks exploit power variations to extract the secret. However, due to the noise level, the power analysis normally

**Table 2: Architecture and system configurations.**

| | i7-4770 | i7-9700 |
|---|---|---|
| Architecture | Haswell | Coffee Lake |
| CPU cores | 4 | 8 |
| Last Level Cache | 8 MB | 12 MB |
| Operating System | Ubuntu 18.04 | Ubuntu 18.04 |
| ASLR/KASLR | Enabled | Enabled |
| DRAM | DDR4, 2 x 4 G | DDR4, 2 x 8 G |

needs to locate the accurate timing information of AES S-Box operations, i.e., loading plaintext byte-by-byte and XOR with the secret key, and then permuting the initial ciphertext via S-Box, or RSA multiplication-addition operations, i.e., loading private key bit-by-bit and executing timing-constant branch shown in Figure 3 [41]. In this paper, we target the timing information of key loading and the multiplication-addition operations at decryption from the most recent OpenSSL-RSA library (OpenSSL-AES can also be attacked using the same attack flow), which is a commercial-grade for general-purpose encryption shared library [46].

By calling the `sched_yield()` more frequently, we can check the prefetcher status with much finer granularity to properly determine the time, and thus leak the accurate timing of interested load operations. To further reduce the time consumed by the attacker, instead of training the prefetcher before each detection, we solely mistrain it before the victim runs. In this paper, we train the IP-based prefetcher with stride N and we always access $current_address + N$ in the detection phase to guarantee that the prefetcher status will not be reset by us. As a result, the prefetcher state will change only if the target load operation is executed.

We summarize all variants and scenarios in Table 3 to give an overview of AfterImage.

# 7 EXPERIMENTS

## 7.1 Experimental Setup

We perform all the experiments on Haswell and Coffee Lake machines. The architecture details and OS configuration of these two machines are shown in Table 2.

Except for the IP-stride prefetcher, we found that the other three hardware prefetchers can introduce false positives into the results by loading additional cache lines unexpectedly. Fortunately, these prefetchers do not have the address range reach compared to the IP-stride prefetcher [69]. The DCU (next-line) prefetcher prefetches only the next cache line. DPL (adjacent) prefetcher prefetches the previous or next cache line. And the streamer prefetches the previous or next several sequential cache lines.

To avoid the impact of the noise introduced by these three prefetchers, concerning the choice of stride, we use a stride that is greater than four cache lines. Additionally, the use of uncommon stride values (e.g., a larger prime number) can provide additional noise resilience because they can be easily differentiated. In our experiments, we generally train the prefetcher with stride values of 7, 11 and 13.

To prepare a Prime+Probe side-channel, we utilize the slice-selection algorithm found in the Haswell microarchitecture [29] to generate minimal eviction sets (MESs) to cover multiple cache sets for building Prime+Probe. For the Flush+Reload, the shared memory is created by using MMAP function with `MAP_SHARED` label.

## 7.2 AfterImage Variants

For AfterImage variant 1, we show results for both `if` and `else` branch, on Prime+Probe and Flush+Reload measurements. For variant 2, we present Flush+Reload measurement.

Figure 13a shows the experimental results of leaking the if-path via Prime+Probe after one round of observation. The x-axis represents the cache set number, i.e. cache line index, in our observing page (4 KiB page with 64 cache lines), whose distance directly represents the offset between memory addresses in a cache-line-length unit. The y-axis shows the time taken, between the probing phase and priming phase, to access each MES of the cache set. As depicted in the figure, most cache sets have not been accessed. The two cache sets with the highest time delta show a clear stride of 7, demonstrating that the targeted `load` on the if-path was executed, triggering a trained prefetch response of 7 cache lines.

We then call the proposed `gadget` to train the prefetcher for both paths and try to consistently leak control flow round-by-round. From Figure 13b and Figure 13c, we observe clear signals (strides) after the victim performed the branch. During the first round, we see that the victim took the else-path. The victim then executed the if-path in the following cycle. If the branch is security-related, we then know the secret is `b'10`.

In terms of AfterImage variant 2, we perform the IP search as described in Section 5.2. When a matched IP is found, we perform the side-channel workflow, as has been introduced in Section 5.2. In this example, we set the training stride in the user space to 11. The detected stride is shown in Figure 14a, which indicates that the kernel function executed the if-path.

To further evaluate the attack success rate, we perform two variants with 200 rounds on a set of sample data. We conduct this evaluation on the platform described in Table 2. The attack success rate of variant 1 (cross-thread), variant 1 (cross-processes), and variant 2 are 99%, 97%, and 91%, respectively.

Figure 14b demonstrates an example of using the proposed covert channel to transmit a 5-bit secret. The attacker trains one entry in the prefetcher with a stride at `b'11110`. We can see a stride at 30 (`b'11110`) from the result, which corresponds to the secret the attacker wants to transmit. The bandwidth of the proposed covert channel can achieve 833 bps with an error rate of less than 6%. By training additional entries with varied strides (secrets), the bandwidth can be further improved. The prefetcher, however, may be affected by the process context switch since numerous memory accesses occur here. As a result, the error rate is greater than 25%. But, the maximum bandwidth will be close to 20 Kbps (i.e., train 24 entries).

To extract the control flow from SGX, in this work, we set stride in the taken branch is 5 and the non-taken branch is 3, and train the IP-stride prefetcher 8 times. If the `secret` is set to 0, we always find that the `Time1` (in Figure 10) is lower than 50 cycles, and `Time2` is higher than 200 cycles, and vice versa. This implies that the attacker can observe that the IP-stride prefetcher is trained with stride 3, and know the `secret` is 0.

**Table 3: A Summary of the AfterImage Variants and Applications. *TC, P+P, F+R, and PSC represent Timing-Constant, Prime+Probe, Flush+Reload, Prefetcher Status Checking, respectively.**

| | Proof of Concept | | | | Realistic Attack | |
|---|---|---|---|---|---|---|
| | V1 | | V2 | Covert Channel | TC-RSA | OpenSSL-RSA |
| Goal | Cross-thread control flow leakage | Cross-processes control flow leakage | Cross user-kernel boundary control flow leakage | Transmit secrets between different processes | RSA key leakage | Locate timing to benefit other attacks |
| Available Secret Extraction Tech. | P+P, F+R, PSC | F+R, PSC | F+R, PSC | F+R, PSC | F+R, PSC | F+R, PSC |
| IP pre-analysis | Disassembly | Disassembly | IP searching | Disassembly | Disassembly | Disassembly |
| Evaluation | P+P in Section 7.2 Figure 13a and Figure 13b | F+R in Section 7.2 Figure 13c | F+R in Section 7.2 Figure 14a | F+R in Section 7.2 Figure 14b | PSC in Section 7.3 Figure 14c | PSC in Section 7.4 Figure 15 |

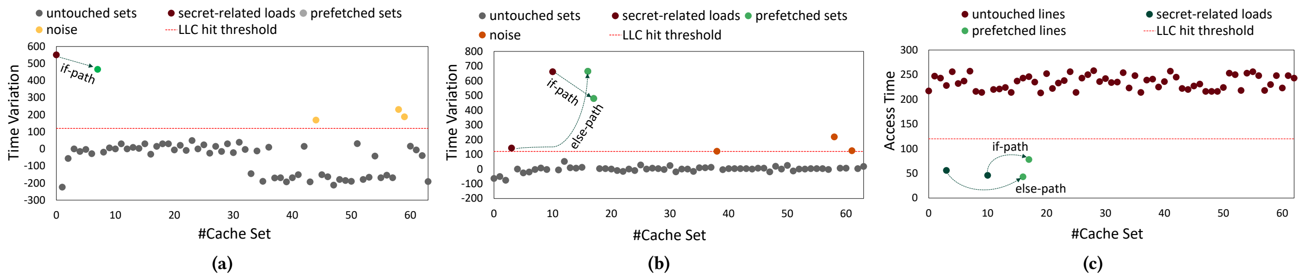



**Figure 13: Attack results of AfterImage-Cache variant 1: (a) cross-thread single bit extraction from `if-path` (b) cross-thread round-by-round extraction from real execution flow with Prime+Probe. (c) cross-processes round-by-round extraction from real execution flow with Flush+Reload.**

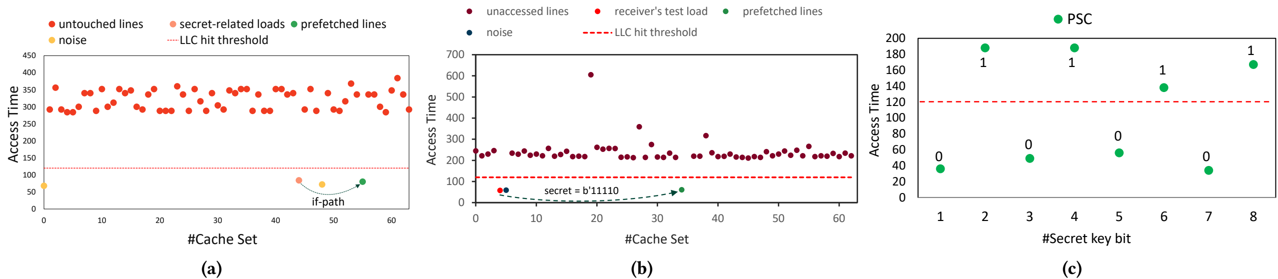



**Figure 14: (a) Attack result for AfterImage-Cache variant 2 with Flush+Reload. (b) AfterImage-Cache covert channel stride detection in receiver's space with Flush+Reload. (c) RSA private key is revealed through AfterImage-PSC. If the private key's bit = 1, the targeted load instruction will be executed, the prefetcher status will be updated, and it will no longer be triggered.**

## 7.3 Timing-Constant RSA

Part of the RSA attack result is illustrated in Figure 14c (an 8-bit secret sequence (b'01010101)). In our experiments, it takes at most 5 iterations (about 10 seconds) to leak one bit, and for the 1024-bit private exponent, or key, it takes about 188 minutes. Multiple iterations per bit are needed because the success rate of AfterImage-PSC (82%) is slightly lower than AfterImage-Cache using Flush+Reload.

## 7.4 Tracking Load Timing from OpenSSL

Figure 15 demonstrates when the key is loaded in the program and when the decryption proceeds. We can clearly see that the prefetcher status is changed by the target memory operations. Note that there are two misses because the prefetcher needs one more

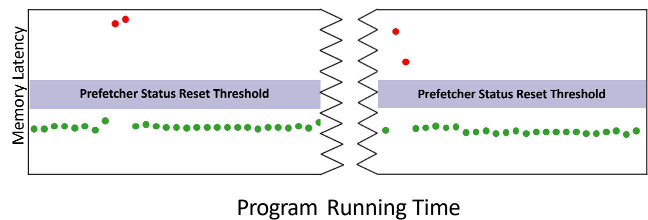



**Figure 15: Tracking when the key is loaded in the OpenSSL-RSA (left) and when the decryption happens in the OpenSSL-RSA (right) via AfterImage-PSC.**

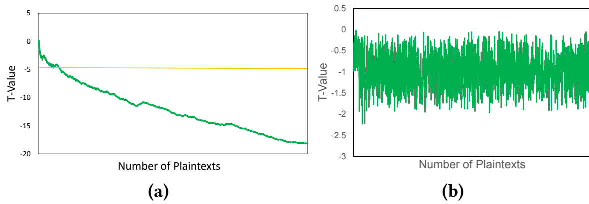



**Figure 16: T-test result (a) with accurate timing information and (b) with randomly picked timing information. The yellow line represents the leakage threshold (-4.5).**

step to re-train its `stride` (see Section 4.2 for a more detailed description of the stride update policy). To further show that the timing information of memory operations leaked by AfterImage is critical for power analysis [30, 41]), we performed a t-test [57] to measure the power leakage in the AES encryption (t-test on RSA is not supported). The t-test was proposed as a PASS/FAIL test, which checks if the t-value crosses the predefined threshold (proposed as ±4.5 [57]), i.e., whether data-dependant information can be potentially exploited. We perform a t-test on the Rocket Chip processor [2], an open source RISC-V in-order processor, not an x86 processor, as Rocket Chip allows us to collect cycle-accurate power traces via PrimePower and thus get a more reliable and clearer t-test result. Figure 16a shows the t-test analysis result with accurate timing information (i.e., the timing of AES S-Box operation happens). We can see that the leakage is close to -18.8 which by far outperforms the predefined leakage threshold of -4.5. In contrast, the timing information is picked randomly, and the t-test result (see Figure 16b) fluctuates around -2.

# 8 MITIGATING AFTERIMAGE

## 8.1 Impact of Existing Defenses

We first discuss the effectiveness of AfterImage in the presence of state-of-the-art defenses that aim to mitigate microarchitectural side-channel attacks. **Control flow integrity** [5, 32, 40] based defenses use protection models that automatically check whether the runtime control flow is deviating from the control flow graph. However, only the backend prefetcher is affected by AfterImage, and this occurs during the non-speculative path of execution. Leveraging **performance counters**, the defender might be able to identify abnormalities in vulnerable hardware components during runtime [11, 55, 73]. However, the sampling frequency of the Intel performance monitor [54] may not be enough to capture the prefetcher training event, since AfterImage requires just two to three iterations of training at a minimum. Some works [11, 44, 49, 73] have been proposed to **enhance cache** to prevent cache primitives by tracking abnormal events (e.g., a large number of fetching or LLC/L1D cache misses) and by prohibiting data or instruction fetching, or by using randomized cache [52]. However, AfterImage can leak the secret via PSC, which is cache primitive independent.

## 8.2 Mitigation Options

A straightforward defense is to **disable** the IP-stride prefetcher to prevent possible security risks with high performance overhead. **Augmenting the history table with extra tags** that include execution context-specific information such as the process ID prevents hardware sharing. It requires hardware modification and an increased hardware budget. **Redesigning the application** by the developer to avoid secret-dependent branches can also prevent this issue [27]. Similarly, **oblivious execution** [53, 72] removes any control flow and most data dependencies. Nevertheless, the use of oblivious code faces practical difficulties, as it leads to significant overhead in many applications [51]. **Secure timer** can obfuscate the cache access latency by adding noise [25, 42] to measured timing. But it is built on a specific kernel and extended ISA that is often costly to implement. In addition, caching page table entries of sensitive data in an **isolated cache** rather than traditional caches (e.g., CATalyst [38]) can also mitigate AfterImage at the cost of high hardware overhead [20].

## 8.3 Proposed Mitigation Evaluation

We propose a privileged `clear-ip-prefetcher` instruction that can be used on a context switch to flush all entries in the IP-stride prefetcher. The performance penalty of this mitigation is the cost to flush the prefetcher state, plus the penalty for potential misses due to that flush. This penalty would occur at each domain switch. We model the upper-bound cost as: $(C_{clear} + C_{miss} \times 3 \times 24)/Domain_Switch_Period$. $C_{clear}$ is the number of cycles to clear the IP-stride prefetcher and $C_{miss}$ is the latency gap between cache and DRAM. Since IP-stride prefetchers have 24 entries, we assume that $C_{clear} \approx 24$, one cycle for clearing each entry. The value of $C_{miss}$ is on the magnitude of 300 cycles [24], and the period of calling a system call in a modern operating system is approximately 100 microseconds [64]. Using these values, we compute the upper-bound, estimate of the time penalty to be less than 7.3% on a 3GHz machine.

We then model a Coffee Lake-like processor, and implement the Intel IP-stride prefetcher on top of ChampSim [7], a cycle-level simulator for evaluating a number of state-of-the-art prefetcher designs [13, 48]. We emulate a more frequent prefetcher flushing (10 microseconds). One billion instructions on the applications from SPEC CPU2006 and 2017 [62, 63] benchmark traces [13] are then run on the processor to verify the performance penalty. By comparing the normalized IPC, we find that the average performance reduction is only 0.7% for the top 8 prefetching-sensitive applications, and 0.2% across all tested applications.

# 9 RELATED WORK

## 9.1 Prefetching Side-Channels

Table 4 summarizes previously published prefetcher/prefetch related side-channel attacks (SCAs) or covert-channels (CC). The work of Shin et al. [59] first discovered an information leakage from IP-stride prefetchers. They observe that, when the victim program shows a stable memory access behavior, e.g. table look-ups, the IP-stride prefetcher can be triggered and leave a special footprint in the cache. This attack requires at least 19 CPU hours for recovering a 568-bit key in the ECDH algorithm [45]. Two software prefetch side-channels [21, 36] aim to bypass Supervisor Mode Access Prevention (SMAP) and KASLR on Intel processors or break

**Table 4: Overview and comparison of prefetching-based attacks. The second and third columns indicate whether the corresponding technique can leak the control flow (CF) of applications and any load instruction timing (LT) information, respectively. ⋆Algorithm agnostic represents that the technique does not target a specific algorithm.**

| Technique | CF | LT | Cache Primitives Agnostic | Type | Algorithm Agnostic | Side-channel Source | Target Vulnerability |
|---|---|---|---|---|---|---|---|
| Prefetcher SCA [59] | ✗ | ✗ | ✗ | Side-channel | ✗ | Intel HW prefetcher | Table look-ups |
| Augury [67] | ✗ | ✗ | ✗ | Side-channel | ✗ | Apple M1 Data Memory-Dependent Prefetcher | ASLR, leaked pointer, etc. |
| Prefetch attack [21, 36] | ✗ | ✗ | ✗ | Side-channel | ✗ | Intel/AMD SW PREFETCH instruction | (fine-grained) KASLR, SMAP |
| Prefetcher CC [12, 56] | ✗ | ✗ | ✗ | Covert-channel | ✓ | Intel HW prefetcher | - |
| AfterImage (This work) | ✓ | ✓ | ✓ | Side-channel | ✓ | Intel HW prefetcher | Any (branch owned) LOAD |

fine-grained KALSR and dump kernel memory layout with Spectre on AMD processors, respectively. They exploit the timing of `prefetch` instructions to leak the translation level of the virtual address and infer the physical mapping. However, the main purpose of these works is different from AfterImage, where we aim to leak secret data directly. In addition, one recent work [67] exploits a different prefetcher, the data memory-dependent prefetcher, in the Apple M1 processor to perform out-of-bounds read and retrieve leaked pointers. However, this attack assumes the same memory space for the attacker and victim, and relies on cache primitives.

## 9.2 Other Microarchitectural Side-Channels

The branch predictor has been exploited extensively to conduct speculative execution-based attacks [5, 18, 31, 37, 55, 58, 66]. In these works, the branch predictor can be mistrained by the adversary to force speculative execution of mispredicted instructions, and therefore trigger transient attacks. BPU-based attacks can leak control flow but require a much longer time of training. For example, Spectre needs 26000 cycles in the mistraining [34], while AfterImage requires only 3 to 4 iterations of a load loop (1000-2000 cycles in the presence of page misses). Besides, since the BTB normally uses 20 IP address bits [31], these attacks need several rounds of testing to bypass ASLR, while AfterImage is not affected by ASLR.

Apart from the cache, BPU and prefetcher, other attacks exploiting different microarchitectural components are gradually discovered, such as line-fill buffer [58, 66], TLB [19, 70], ports [1], and front-end [14, 55, 70].

## 9.3 Reverse-Engineering the Hardware Prefetcher

The IP-stride prefetcher, as implemented in Intel's Sandy Bridge microarchitecture, was previously described in an Intel white paper [16]. Haswell, a newer generation microarchitecture, uses enhanced data prefetchers [28], but the details of these updates remain undocumented. Shin et al. [59] disclose the basic strided prefetching manner but did not provide more information. Cronin et al. [12] discover that the entry of the IP-stride prefetcher is shared by different processes running on the same core. They did not, however, provide an in-depth analysis of the IP-stride prefetcher. Our reverse engineering goes beyond these previous efforts. We designed a series of micro-benchmarks to validate or discover features of Intel IP-stride prefetcher. We then disclose the detailed indexing, capacity, learning, triggering, page boundary checking and replacement mechanisms, most of which have never been disclosed before.

# 10 CONCLUSION

In this work, we observe that the widely present IP-stride prefetcher in Intel processors can be intentionally trained and triggered across domain switching. We leverage this feature to introduce a novel side-channel attack named AfterImage that can track the load operation in other threads/processes and kernel context. To accomplish the attack, we present an in-depth study of the Intel IP-stride prefetcher, revealing a number of undocumented details. We demonstrate that AfterImage can leak the victim's control flow and branch secret across threads/process spaces, and user-kernel boundary. We show that AfterImage achieves a success rate of up to 99%, depending on the variant. The adapted cross-processes covert channel has a low error rate (<6%). We further apply AfterImage to leak the secret key in the timing-constant RSA algorithm which takes only 188 minutes to reveal the private key. We also show that AfterImage can track load instructions in OpenSSL and this information can significantly improve the effectiveness of power attacks. AfterImage works on the non-speculative path, and our methodology is independent of cache primitives, which are the basis for many hardware attacks. Finally, we find that the current defenses against speculative or cache side-channels are insufficient to block AfterImage, and we propose and evaluate a low overhead solution that prevents IP-stride prefetcher leakage on hardware platforms today.

# 11 DATA AVAILABILITY STATEMENT

The data that support the findings of this paper are openly available in Zenodo at https://doi.org/10.5281/zenodo.7218907 [9], reference number 7218907.

# ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their detailed feedback which allowed us to improve this work. This work was supported by a grant from the National Research Foundation (NRF) of Singapore (NRF2018NCR-NCR002).

# A ARTIFACT APPENDIX

## A.1 Abstract

AfterImage is a hardware side-channel inside specific Intel processors. In this artifact, we provide the needed information to reproduce the main results presented in the paper. The required hardware and software configurations are shown in Table 5. We demonstrate the information leakage through observing load timing.

**Table 5: Artifact Requirements**

| Parameter | Value |
|---|---|
| CPU | i7-4770, i7-9700 (SGX Supported) |
| Compiler | GCC 8.4.0, -O0 |
| Software | Python3, Make |
| Operating System | Ubuntu 18.04 |
| Linux Kernel | 5.4.0 |

## A.2 Artifact Check-List (Meta-Information)

- **Compilation:** GCC 8.4.0
- **Run-time environment:** Ubuntu 18.04, Linux kernel 5.4.0, Python3, make, sudo
- **Hardware:** Intel CPU i7-4770, i7-9700 (Support SGX)
- **Run-time state:** enabled Level-2 ASLR. Sensitive to cache activities.
- **Execution:** 5 cases, each can be finished in seconds. But they may need multiple rounds.
- **Metrics:** measured load timing and the occurence of specific pattern.
- **Output:** cacheline indexs and corresponding timing value
- **Experiments:** scripts provided.
- **How much disk space required (approximately)?:** 20MB
- **How much time is needed to prepare workflow (approximately)?:** 1 day
- **How much time is needed to complete experiments (approximately)?:** 2 days
- **Publicly available?: yes**
- **Code licenses (if publicly available)?: Creative Commons Attribution 4.0 International**
- **Archived (provide DOI)?: 10.5281/zenodo.7218907**

## A.3 Description

*How to access*: We run our experiments on i7-4770 and i7-9700 processors, and thus recommend using these two machines to reproduce these results. One can find the code for this PoC at https://doi.org/10.5281/zenodo.7218907.

## A.4 AfterImage V1

In this section, we introduce how to run the PoC of AfterImage V1, which is AfterImage-Prime+Probe in the same memory space. The compilation command is shown below. The Prime+Probe needs to build eviction set, which is a time-consuming phase as the LLC slice hash function is unknown. The slice hash function in Haswell, however, has already been revealed by other researchers. Thus, the AfterImage-Prime+Probe can be performed on the i7-4770 using these details.

To run the code, first, go to the `prefetching_attack/prefetching_poc/V1.A` directory.

After compiling, we can quickly test our attack through the following command:

```
$V1.A: make all
```

```
$V1.A: sudo ./main
```

We added admin capabilities to that binary as we are using `/proc/pid/pagemap` for virtual address and physical address translation for building the ESs for the specific machine. In fact, only Prime+Probe flow here requires sudo for this reason, while Flush+Reload or PSC does not require. If a `segment fault` error is reported, we provide two methods to solve it: 1) re-boot the machine and re-make the binary or 2) increase the size of `space` pool and re-make the binary. The reason for this error is that the memory pool for building the ESs may not be large enough as the OS may automatically collect some useless pages, causing many pages to have the same physical address.

The attack result will be printed on the screen. The first column is the normalized set number, the second column represents the memory access time (clock cycles) recorded in `current_probe` of corresponding set. The last column denotes *atv*, i.e., average time variation. We assume that the cache miss happens if the *atv* is larger than 120. In the PoC, we set strides in `if-path` and `else-path` to 7 and 13, respectively. An example output is in Figure 13 (b).

## A.5 AfterImage V1-Cross-Processes

Before testing the PoC of AfterImage-V1-cross-processes, we firstly go to `prefetching_attack/prefetching_poc/V1.B` directory, an d then enable the ASLR level-2 by running the command on i7-9700:

```
$V1.B: echo 2 | sudo tee /proc/sys/kernel/rando
mize_va_space
```

To reproduce the results in Figure 13c and Figure 14b, we run the following command:

```
$V1.B: python poc.py
```

The python script will automatically run the attacker and victim multiple times and measure one cache line every round (to get the most accurate results). To get a stable result, we recommend running the script multiple times (run `poc_e2e.py` directly).

```
$V1.B: python poc_e2e.py
```

## A.6 AfterImage V2

To reproduce the result in Figure 14a, we build a simple kernel function to conduct a straight-forward proof-of-concept to demonstrate how the prefetcher can leak information between user and kernel space. Listing 7 includes the customized kernel function. Please change directory to `prefetching_attack/prefetching_poc/V2`.

To add this function into the kernel, you should firstly check the available system call number in your system (and the system call table address)[6]. We recommend run this experiment on i7-4770 as it is easy to mount the custom kernel by running following commands:

```
$V2: sudo python syscall_ins.py

$V2: gcc -o verify -O0 test.c

$V2: sudo ./verify
```

We use sudo as we need to install the new syscall function into the kernel. In fact, the attack program doesn't require sudo.

To detect whether the kernel function can trigger the prefetcher trained in user space, we first test the memory access time of `memory_space[address + stride x CACHE_LINE_SIZE]`. If the prefetcher trained in the user space can be harnessed directly in the kernel space and the kernel function executed the `if-path`, we should observe one or two group getting cache hits (since we constructed 480 load instructions, some instruction's lower 8 bits are repeated).

We then leverage the Flush + Reload to enable a side-channel aimed at exposing OS secrets (modify the code following the provided `readme`, recompile the binary, and run `attack`).

```
$V2: gcc -o attack -O0 test.c

$V2: ./attack
```

We train the prefetcher with stride at 11. To ensure that the prefetcher will not impact the measurement result, when we reload the page, instead of loading the cache line sequentially, we use the modern version of the Fisher–Yates shuffle algorithm [17] to randomize the index sequence in the searching range (i.e., [0,63]) and add `mfence` to introduce memory barrier (According to the Intel manual, the memory barrier may prevent prefetching from taking place).

## A.7 AfterImage Timing-Constant RSA

In this experiment, we provide an example to show how to accurately leak one-bit of RSA's private key in the timing-constant branch. The experiment is used to generate the result shown in Figure 14c, which is also an experiment to verify the AfterImage-PSC.

We already aligned the last 8 bits of the attacker's memory access instruction (used to train the IP-stride prefetcher) with the memory access instruction in the taken-branch in the encryption function. To change the private key's bit, you can go to `victim()` and change the second parameter (0 or 1) in `modpow`.

You should first go to `prefetching_attack/prefetching_poc/rsa` directory. To re-compile the code, you can run:

```
$rsa: gcc -o st_rsa -O0 st_rsa.c
```

To execute the binary, we can run:

```
$rsa: sudo ./st_rsa
```

## A.8 AfterImage SGX Side/Covert Channel

As the covert channel is similar to the side channel, we only provide the PoC of the side channel. The PoC of the covert channel can be implemented by easily removing the branch, and higher accuracy can be achieved by training the prefetcher multiple times. The experiment is run on the i7-9700. Please first go to the `prefetching_attack/prefetching_poc/intel_sgx` directory. To check, enable SGX:

```
$intel_sgx/sgx-software-enable: sudo ./sgx_enable
```

If it doesn't work, you can also enable the SGX via BIOS. To run PoC:, you should go to `intel_sgx/sgxsdk/SampleCode/AfterImage_SGX_POC` and update the `SGX_SDK` in the `Makefile` to your sgxsdk root directory. To perform the PoC, you can run:

```
$intel_sgx/sgxsdk/SampleCode/AfterImage_SGX_POC: make

$intel_sgx/sgxsdk/SampleCode/AfterImage_SGX_POC: sudo ./app
```

In this experiment, we set the `secret` in `Enclave/Enclave_t.c`. If `secret` is equal to 1, the stride is set to 3, otherwise, the stride is set to 5. We train the prefetcher 7 times (the minimum is 3 times) and then measure the access time of expected prefetched cache lines in the untrusted zone, i.e., the (3*8)-th and (5*8)-th cache lines, to see which is prefetched by the prefetcher. The result shows that if the secret is 1, the untrusted zone can successfully detect that the 24-th cache line is hit in the cache but the 40-th cache line is not, and vice versa. We used this PoC to successfully extract data from the SGX enclave. By using this new technique, we can avoid easier-to-detect and mitigate cache primitives, e.g., Prime+Probe or Flush+Reload.

[6]In our experimental environment, an available system call number is 333. We use a system function to directly check the syscall table addresses.

## REFERENCES

[1] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *Symposium on Security and Privacy (S&P)*. 870–887. https://doi.org/10.1109/SP.2019.00066
[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016).
[3] Jean-Loup Baer and Tien-Fu Chen. 1991. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Conference on Supercomputing (SC)*. 176–186. https://doi.org/10.1145/125826.125932
[4] Battery properties management 2021. Battery properties management in Linux kernel. https://git.kernel.org/pub/scm/bluetooth/bluetooth-next.git/tree/drivers/hid/hid-input.c#n383.
[5] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. 2020. SpecROP: Speculative Exploitation of

ROP Chains. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 1–16.

[6] Bluetooth connection activity management 2021. Bluetooth connection activity management in Linux kernel. https://git.kernel.org/pub/scm/bluetooth/bluetooth-next.git/tree/drivers/hid/hid-input.c#n247.

[7] ChampSim 2020. ChampSim. https://github.com/ChampSim/ChampSim.

[8] Jed Kao-Tung Chang, Chen Liu, Shaoshan Liu, and Jean-Luc Gaudiot. 2011. Workload characterization of cryptography algorithms for hardware acceleration. In *International Conference on Performance Engineering*. 381–390. https://doi.org/10.1145/1958746.1958800

[9] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. 2022. AfterImage Artifact. https://doi.org/10.5281/zenodo.7218907

[10] Razvan T Cheveresan and Stefan Holban. 2009. Workload Characterization an Essential Step in Computer Systems Performance Analysis-Methodology and Tools. *Advances in Electrical and Computer Engineering* 9, 3 (2009), 100–106. https://doi.org/10.4316/AECE.2009.03018

[11] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. 2016. Real Time Detection of Cache-Based Side-Channel Attacks Using Hardware Performance Counters. *Applied Soft Computing* (2016), 1162–1174. https://doi.org/10.1016/j.asoc.2016.09.014

[12] Patrick Cronin and Chengmo Yang. 2019. A Fetching Tale: Covert Communication with the Hardware Prefetcher. In *International Symposium on Hardware Oriented Security and Trust (HOST)*. 101–110. https://doi.org/10.1109/HST.2019.8741033

[13] Data Prefetching Championship 2019. 3rd Data Prefetching Championship (DPC). https://dpc3.compas.cs.stonybrook.edu.

[14] Shuwen Deng, Bowen Huang, and Jakub Szefer. 2022. Leaky Frontends: Security Vulnerabilities in Processor Frontends. In *International Symposium on High-Performance Computer Architecture (HPCA)*. 53–66. https://doi.org/10.1109/HPCA53966.2022.00013

[15] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security Symposium (USENIX Security)*. 51–67.

[16] Jack Doweck. 2006. White paper inside Intel® Core™ Microarchitecture and Smart Memory Access. *Intel Corporation* (2006), 72–87.

[17] Richard Durstenfeld. 1964. Algorithm 235: Random Permutation. *Commun. ACM* (1964), 420–422. https://doi.org/10.1145/364520.364540

[18] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 693–707. https://doi.org/10.1145/3173162.3173204

[19] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security Symposium (USENIX Security)*. 955–972.

[20] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *Network and Distributed System Security Symposium (NDSS)*. 26–41. https://doi.org/10.14722/NDSS.2017.23271

[21] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Conference on Computer and Communications Security (CCS)*. 368–379. https://doi.org/10.1145/2976749.2978356

[22] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[23] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–bringing access-based cache attacks on AES to practice. In *Symposium on Security and Privacy (S&P)*. 490–505. https://doi.org/10.1109/SP.2011.22

[24] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. https://doi.org/10.5555/3207796

[25] Wei-Ming Hu. 1991. Reducing timing channels with fuzzy time. In *Symposium on Research in Security and Privacy*. 8–20. https://doi.org/10.1109/RISP.1991.130768

[26] Intel. 2018. Inconsistency in TLB miss counters. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/593830.

[27] Intel. 2019. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html.

[28] Intel. 2019. Intel® 64 and IA-32 Architectures Optimization Reference Manual. *Intel Corporation* (2019).

[29] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *Euromicro Conference on Digital System Design (DSD)*. 629–636. https://doi.org/10.1109/DSD.2015.56

[30] Demme John, Martin Robert, Waksman Adam, and Sethumadhavan Simha. 2012. Side-channel vulnerability factor: A metric for measuring information leakage. In *International Symposium on Computer Architecture (ISCA)*. 106–117. https://doi.org/10.1109/ISCA.2012.6237010

[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Symposium on Security and Privacy (S&P)*. https://doi.org/10.1109/SP.2019.00002

[32] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2020. SpecCFI: Mitigating spectre attacks using cfi informed speculation. In *Symposium on Security and Privacy (S&P)*. 39–53. https://doi.org/10.1109/SP40000.2020.00033

[33] Ladder RSA 2019. Montgomery-Ladder RSA. https://github.com/merinjo/RSA-Montgomery-Ladder-Implementation.

[34] Congmiao Li and Jean-Luc Gaudiot. 2020. Challenges in Detecting an "Evasive Spectre". *IEEE Computer Architecture Letters (CAL)* (2020), 18–21. https://doi.org/10.1109/LCA.2020.2976069

[35] Linux kernel 2022. Linux kernel API. https://www.kernel.org/doc/htmldocs/kernel-api/API---copy-from-user.html.

[36] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD Prefetch Attacks through Power and Time. In *USENIX Security Symposium (USENIX Security)*. 643–660.

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium (USENIX Security)*. 973–990. https://doi.org/10.1145/3357033

[38] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *Symposium on High Performance Computer Architecture (HPCA)*. 406–418. https://doi.org/10.1109/HPCA.2016.7446082

[39] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *Symposium on Security and Privacy (S&P)*. 605–622. https://doi.org/10.1109/SP.2015.43

[40] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *USENIX Security Symposium (USENIX Security)*. 1394–1414.

[41] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2008. *Power analysis attacks: Revealing the secrets of smart cards*. https://doi.org/0.1007/978-0-387-38162-6

[42] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *International Symposium on Computer Architecture (ISCA)*. 118–129. https://doi.org/10.1109/ISCA.2012.6237011

[43] Mbed TLS 2020. Mbed TLS. https://github.com/Mbed-TLS/mbedtls/tree/v2.16.7.

[44] Samira Mirbagher-Ajorpaz, Gilles Pokam, Esmaeil Mohammadian-Koruyeh, Elba Garza, Nael Abu-Ghazaleh, and Daniel A. Jiménez. 2020. PerSpectron: Detecting Invariant Footprints of Microarchitectural Attacks with Perceptron. In *Symposium on Microarchitecture (MICRO)*. 1124–1137. https://doi.org/10.1109/MICRO50266.2020.00093

[45] National Institute of Standards and Technology 2013. National Institute of Standards and Technology. 2013. FIPS PUB 186-4 Digital Signature Standard (DSS). https://csrc.nist.gov/publications/detail/fips/186/4/final.

[46] OpenSSL 2018. OpenSSL, Cryptography and SSL/TLS Toolkit. http://www.openssl.org.

[47] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*. 1–20. https://doi.org/10.1007/11605805_1

[48] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *International Symposium on Computer Architecture (ISCA)*. 118–131. https://doi.org/10.1109/ISCA45697.2020.00021

[49] Arash Pashrashid, Ali Hajiabadi, and Trevor E Carlson. 2022. Fast, Robust and Accurate Detection of Cache-based Spectre Attack Phases. In *International Conference on Computer-Aided Design (ICCAD)*. https://doi.org/10.1145/3508352.3549330

[50] Colin Percival. 2005. Cache missing for fun and profit. In *Technical BSD Conference (BSDCan)*. 1–13.

[51] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Čapkun. 2021. Frontal attack: leaking control-flow in SGX via the CPU frontend. *USENIX Security Symposium (USENIX Security)*, 663–680.

[52] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *Symposium on Microarchitecture (MICRO)*. 775–787. https://doi.org/10.1109/MICRO.2018.00068

[53] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium (USENIX Security)*. 431–446.

[54] James Reinders. 2005. VTune performance analyzer essentials. *Intel Press* (2005).

[55] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. 2021. I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *International Symposium on Computer Architecture (ISCA)*.

361–374. https://doi.org/10.1109/ISCA52012.2021.00036

[56] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. 2020. Reverse Engineering the Stream Prefetcher for Profit. In *European Symposium on Security and Privacy Workshops (EuroS&PW)*. 682–687.

[57] Tobias Schneider and Amir Moradi. 2015. Leakage assessment methodology. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. 495–513. https://doi.org/10.1007/978-3-662-48324-4_25

[58] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Conference on Computer and Communications Security (CCS)*. 753–768. https://doi.org/10.1145/3319535.3354252

[59] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. 2018. Unveiling Hardware-Based Data Prefetcher, a Hidden Source of Information Leakage. In *Conference on Computer and Communications Security (CCS)*. 131–145. https://doi.org/10.1145/3243734.3243736

[60] Alan Jay Smith. 1978. Sequential program prefetching in memory hierarchies. *Computer* (1978), 7–21. https://doi.org/10.1109/C-M.1978.218016

[61] Wei Song and Peng Liu. 2019. Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the LLC. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 427–442.

[62] SPEC2006 2006. SPEC CPU2006. https://www.spec.org/cpu2006/.

[63] SPEC2017 2017. SPEC CPU2017. https://www.spec.org/cpu2017/.

[64] System Call Frequency 2013. SystemTap Beginners Guide: Tracking Most Frequently Used System Calls. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/systemtap_beginners_guide/topsyssect.

[65] Timing-constant RSA 2021. Timing-constant RSA. https://github.com/ARMmbed/mbedtls/blob/3b9bea0757814c346d0848e9058afa1b499fcc19/library/bignum.c#L1528.

[66] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking data on Intel CPUs via cache evictions. (2021), 339–354. https://doi.org/10.1109/SP40001.2021.00064

[67] Jose Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Chris Fletcher, and David Kohlbrenner. 2022. Augury: Using data memory-dependent prefetchers to leak data at rest. In *Symposium on Security and Privacy (S&P)*. 1518–1518. https://doi.org/10.1109/sp46214.2022.9833570

[68] Pepe Vila, Boris Köpf, and José F. Morales. 2019. Theory and Practice of Finding Eviction Sets. In *Symposium on Security and Privacy (S&P)*. 39–54. https://doi.org/10.1109/SP.2019.00042

[69] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. 2019. PAPP: Prefetcher-aware prime and probe side-channel attack. In *Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3316781.3317877

[70] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Conference on Computer and Communications Security (CCS)*. 2421–2434. https://doi.org/10.1145/3133956.3134038

[71] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium (USENIX Security)*. 719–732. https://doi.org/10.5555/2671225.2671271

[72] Jiyong Yu, Lucas Hsiung, Mohamad El'Hajj, and Christopher W Fletcher. 2019. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *Network and Distributed System Security Symposium (NDSS)*. 808–825. https://doi.org/10.14722/ndss.2019.23061

[73] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. 2016. Cloudradar: A real-time side-channel attack detection system in clouds. In *Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. 118–140. https://doi.org/10.1007/978-3-319-45719-2_6

Received 2022-07-07; accepted 2022-09-22