Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker

Zirui Neil Zhao Adam Morrison† Christopher W. Fletcher Josep Torrellas
{ziruiz6, cwfletch, torrella}@illinois.edu †mad@cs.tau.ac.il
University of Illinois Urbana-Champaign †*Tel Aviv University*

Abstract

Microarchitectural side channels are a pressing security threat. These channels are created when programs modulate hardware resources in a secret data-dependent fashion. They are broadly classified as being either stateful or stateless (also known as contention-based), depending on whether they leave behind a trace for attackers to later observe. Common wisdom suggests that stateful channels are significantly easier to monitor than stateless ones, and hence have received the most attention.

In this paper, we present a novel stateless attack that shows this common wisdom is not always true. Our attack, called *Binoculars*, exploits unexplored interactions between in-flight page walk operations and other memory operations. Unlike other stateless channels, Binoculars creates significant timing perturbations—up to 20,000 cycles stemming from a single dynamic instruction—making it easy to monitor. We show how these perturbations are address dependent, enabling Binoculars to leak more virtual address bits in victim memory operations than any prior channel. Binoculars needs no shared memory between the attacker and the victim.

Using Binoculars, we design both covert- and side-channel attacks. Our covert channel achieves a high capacity of 1116 KB/s on a Cascade Lake-X machine. We then design a side-channel attack that steals keys from OpenSSL's side-channel resistant ECDSA by learning the ECDSA nonce k. Binoculars' ability to significantly amplify subtle behaviors, e.g., orderings of stores, is crucial for this attack to succeed because the nonce changes after each run. Finally, we fully break kernel ASLR.

1 Introduction

Microarchitectural side-channel attacks are a serious security threat to modern computer systems. In a side-channel attack, the attacker exploits hardware resources that are shared with the victim. Specifically, when a program's execution *modulates* (i.e., changes the utilization of) hardware resources (i.e., *channels*) as a function of its secret data, an attacker can measure these modulations and from them infer the secret. These attacks have caused a trust crisis in commercial shared-hardware multi-tenant cloud environments, as well as in daily-used personal applications like web browsers.

The ways in which hardware channels can be modulated to pass information can be broken down along two axes. To start, channel modulations are *stateful* if they leave a persistent state change (e.g., the eviction of a line from the cache) [1, 2, 7, 22, 23, 32, 33, 42, 49, 53, 71, 74], or alternatively *stateless* if they create only temporary contention on a resource (e.g., on an execution unit) [4, 6, 9, 24, 27, 44, 68, 69, 72]. Orthogonally, channels can be modulated *directly* by the execution of the victim instruction's micro-ops, or *indirectly* by operations that occur outside the purview of the instruction's micro-ops.[1] The large majority of channel modulations are direct (e.g., all of the above works). An example is a cache attack due to the execution of a victim memory instruction that evicts a line from the cache. On the other hand, there are a handful of channels involving indirect operations [16, 20, 28, 29, 62, 67, 75]. Examples are "implicit" memory operations due to hardware prefetchers or page walkers that occur beyond the purview of micro-ops.

Historically, stateless channels have been considered more difficult to exploit than stateful channels. Indeed, it is relatively easy to monitor a stateful channel because its effect persists after the victim instruction modulating it has retired. Further, the contention effects of stateless channels are typically small, which exacerbates measurement noise.

Our key observation is that all known stateless channels are due to *direct* operations. Accordingly, we perform the first investigation of *stateless-indirect* channels by exploiting interactions between in-flight operations stemming from the hardware page walker and other sources (e.g., program memory operations). We show how stateless-indirect channels are possible and enable powerful new attacks. We call our channel and attack framework *Binoculars*.

We find that because indirect memory operations are issued outside the purview of normal processor structures (e.g., the reorder buffer) they "live by different rules" and exhibit novel interactions with other memory operations. Based on these interactions, we construct novel attack primitives.

First, we show that shared resource contention between page walker (indirect) loads and regular (direct) memory operations can cause *significant* delays in thread execution time (e.g., up to 20,000 cycles) stemming from a single dynamic instruction. This magnitude of delay dwarfs the one created by any other microarchitectural side channel by at least two orders of magnitude. It enables Binoculars to create new low-noise attacks that are relatively easy to perform and observe despite our channel being stateless.

[1] Indirect modulations have been called "implicit" modulations by prior work [75].

Second, we show how the contention depends on the addresses of the memory operations involved. We show that this address dependence does not only apply to high-order address bits (e.g., the page number) or lower-order address bits (e.g., the bits that map the address to a cache set) but also to *intra cache line address bits* and *across address spaces*. In fact, we show that Binoculars can leak more bits of a victim's (virtual) memory address than any prior channel across address spaces.

Using the above attack primitives, we perform end-to-end attacks on security-critical programs. To start, we design and optimize a covert channel using Binoculars' underlying stateless-indirect channel that can achieve a high capacity of 1116 KB/s on a Cascade Lake-X machine. We then design a side-channel attack that steals keys from OpenSSL's side-channel resistant ECDSA by learning the ECDSA nonce k. Here, the nonce is computed by an implementation of the Montgomery ladder algorithm that is hardened against timing side channels. Binoculars is able to amplify subtle nonce-dependent behaviors occurring during execution into large timing delays that can be measured with low noise. This is critical for the attack to succeed, since each run of ECDSA uses a different nonce k. Finally, we fully break kernel ASLR (KASLR).

Contributions. This paper makes the following contributions:
• We investigate and demonstrate the first stateless-indirect channel. It is based on implicit loads issued by the page walker. The resulting attack framework, *Binoculars*, has a high signal-to-noise ratio and leaks a wide range of virtual address bits.
• We design and implement two Binoculars attack primitives. One leaks the byte offset of a store within the page. The other leaks the full virtual page number of a TLB-missing request.
• We demonstrate end-to-end attacks on real hardware, which include extracting the nonce k in ECDSA with a single victim run and fully breaking KASLR.

2 Background

Microarchitectural Side Channels. In a microarchitectural side-channel attack, an attacker learns secret information from a victim program by monitoring some microarchitectural resource—e.g., a cache, branch predictor or execution port—that the attacker shares with the victim, and which the victim uses in a secret-dependent way. Channels can be classified along two axes [3]: stateful vs. stateless and direct vs. indirect.

A *stateful* channel occurs when the victim's use creates persistent changes in the shared resource, which can be monitored by the attacker afterwards. An example is a cache side channel [49, 54, 71]. A *stateless* channel occurs when the changes in the resource are temporary. To see the change, the attacker has to physically contend for the resource during the victim's use. An example is port contention [4, 9, 27].

A *direct* channel is created directly by a victim program instruction. An example is a load that causes a cache line fill [71] or an eviction [49]. An *indirect* channel is created by operations that occur outside the victim program instructions. An example is cache state changes caused by a program-invisible hardware prefetcher.

Out-of-Order Execution. Dynamically-scheduled processors execute data-independent instructions in parallel [65], out of program order. Instructions are dispatched to reservation stations (RS) in program order, where they await execution. An instruction becomes ready to execute once its input operands have been computed. In each cycle, a hardware scheduler picks a subset of ready instructions and issues them to execution units. After they execute, their outputs become available to dependent instructions. Instruction retirement, where the instruction finally frees up its pipeline resources, is done in program order. In-order retirement is implemented by queuing instructions into a FIFO queue called a reorder buffer (ROB) [38] in program order, and retiring an instruction once it reaches the ROB head.

Page Tables in x86. The hardware performs virtual to physical address translation by first partitioning the virtual address into a page number and an offset, and then mapping the virtual page number to a physical page number using a *page table* data structure created by the operating system (OS). In x86-64, the page table is a 4-level radix tree that supports multiple page sizes. We focus on the basic case of 4 KB pages. A page table search is called a *page walk* and is done by a hardware unit called the *page walker* on a TLB miss.

Figure 1a shows the page table structure and the page walk process. Address translation uses four levels of page tables, which we refer to as PL_4, PL_3, PL_2, and PL_1. The root level, PL_4, is pointed to by the CR3 register. Each page in the page tables contains an array of 512 8-byte *page table entries* (PTEs). The virtual page number is decomposed into four 9-bit *PL indexes*, each of which selects a PTE from its corresponding level of the tree. Each PTE holds the physical page number of the next level of the tree or, at the lowest level, the final translation. Overall, to perform a page walk, the page walker issues four loads in total.

Because the 4-level page table supports only a 48-bit virtual address space, the 64-bit virtual addresses in x86-64 must be *canonical*—meaning that bits 64–48 are equal to bit 47. The address space is divided into two equal halves [19]. The lower canonical half is user space, while the upper canonical half is used by the OS kernel. An unprivileged user can only allocate pages in the lower canonical half.

To speed up the virtual address translation process, x86 processors cache address translations in two levels of translation lookaside buffers (TLBs). The first level TLBs (iTLB and dTLB) cache instruction and data translations, respectively. The second level TLB (sTLB) is larger and caches both instruction and data translations. A page walk is triggered if a translation request misses in all levels of TLBs. To minimize the TLB-miss penalty, the page walker loads check the cache hierarchy and so they can benefit from cached PTEs.

False Dependences in the L1D Cache. On Intel processors, the L1D cache is virtually indexed and physically tagged. It uses part of the virtual address (VA) bits (e.g., bits 11-6) as the index to find the cache set and uses the physical address (PA) tag to select the cache line within the set. This design enables the L1D cache to be accessed in parallel with the TLB translation.

The L1D cache uses the 12 least significant bits (i.e., the offset part) of the VA to detect potential dependences between multiple reads and writes that are issued to it, before their translations finish. If a read and a write target addresses with the same offsets (i.e., their 12 least significant bits are the same), then a dependence is possible. When a potential dependence is detected, one of the requests is squashed and will retry. The L1D cache thus conservatively prevents simultaneously reading and writing of addresses that have the same 12 least significant bits (i.e., they are *4K-aliasing*) even though there might be no dependence between the requests (i.e., the dependence may be a *false dependence*) [19, 44, 72]. Depending on the implementation, the dependence check can be done at a word granularity [44]—i.e., the read and the write addresses only need to share bits 11–2 to be counted as potentially dependent. In this paper, we say that two addresses have *4K-aliasing* if they have the same bits 11–2. These addresses are subject to false dependences.

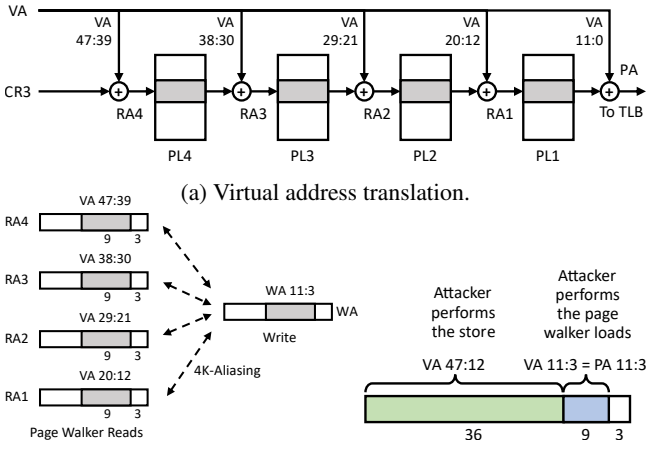
3 Threat Model

We consider an attacker who is an unprivileged user on a hypertheaded multi-core x86 machine. The attacker's goal is to learn some of the bits of the address operand of specific memory load/store instructions in some victim, through local hardware resource utilization changes modulated by the victim. The victim may be another process (belonging to a different user) or the OS kernel. Either way, the victim does not share virtual or physical memory with the attacker processes. We assume that the attacker knows the contents of the victim's executable.

We assume a system configuration similar to that in prior cross-hyperthread side-channel attacks [4, 9, 28, 44, 72]. The system has Hyper-Threading enabled, and the attacker can interact with the OS scheduler to run its attack process on a hyperthread that shares the same physical core with the victim hyperthread. For attacks that rely on observing delays in the victim's execution, we do not assume a cooperative victim that times and reports its own execution.

4 The Binoculars Attack

The CacheBleed [72] and MemJam [44] attacks have shown that existing processors are vulnerable to false dependences between writes issued by a thread and reads issued by another thread (Section 2). In this paper, we show, for the first time, an attack that exploits false dependences between writes issued by a thread and reads issued by the hardware during a page walk triggered by a second thread. We call the new attack *Binoculars*.



(a) Virtual address translation.

(b) 4K-aliasing addresses.

(c) Virtual address bits leaked.

Figure 1: Overview of the Binoculars attack.

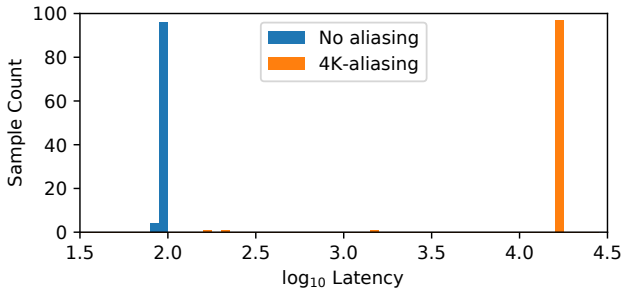


Figure 2: Distribution of the latency of a TLB-missing access while the sibling hyperthread keeps issuing writes whose address may or may not alias with the page walker loads. The data is measured on an Intel Skylake-X.

Compared to the prior false dependence attacks, we will see in this section that Binoculars is both easier to setup and leaks new bits. Specifically, if the attacker is the writer, Binoculars can leak the virtual page number of the victim access that triggers the page walk. On the other hand, if the attacker is the thread that triggers the page walk, Binoculars can leak page offset bits 11-3 of the address written by the victim. Note, the victim and attacker do not share an address space.

To demonstrate Binoculars, we run experiments on Intel Xeon W-2245 (Cascade Lake-X), Intel i7-7820X (Skylake-X), and Intel Xeon E3-1246 v3 (Haswell-EP) platforms. One hyperthread reads from an address that causes a miss in both TLB levels and hence triggers a page walk. Recall that, during the page walk, the hardware issues up to four loads to the data cache hierarchy, corresponding to the requested entries in the four page table levels. In Figure 1a, the four page levels are called PL_4, PL_3, PL_2, and PL_1, and the actual addresses read are RA_4, RA_3, RA_2, and RA_1.

The other sibling hyperthread keeps writing to an address *WA*. We perform two experiments: one where the *WA* has a false dependence with one of the RA_i, and one where it

does not. Following past work [44, 72], a false dependence is obtained with 4K-aliasing — in our case, when bits 11-3 of the two addresses are the same because we issue 8-byte loads. We repeat each experiment 100 times, measuring the time taken by the reader hyperthread to complete its TLB-missing access.

Figure 2 shows the histogram of measured read latencies in the two experiments running on a Skylake-X. To make the histogram readable, we plot the X axis in logarithmic scale. From the figure, we see that when the page walker loads and the store are not 4K-aliasing, the page read takes about 100 cycles (including the time for reading the timestamp). However, when there is 4K-aliasing, the latency goes up to ≈20,000 cycles (or $10^{4.3}$ in the figure). The page walker load is stalled and delayed for a long time. This very obvious difference in latency is exploited by Binoculars to leak address bits.

In the rest of this section, we discuss the two directions of the Binoculars attack: when the attacker triggers the page walk (Section 4.1) and when it performs the repeated writes (Section 4.2).

4.1 Leaking the Page Offset of a Store Address

In this attack, the attacker triggers the page walk and the victim performs repeated writes to the same address. The attacker keeps changing the address of the page that triggers the page walk and measures the latency of an access to the page. When the attacker observes a high access latency, it can deduce that the page offset of a page walk read and of the write have a false dependence. Since, in this attack, the information flows from victim stores to attacker page walker loads, we call this primitive the **store→load channel**.

To understand the attack, consider Figure 1b, which shows the addresses of the four loads issued during a page walk. Given a TLB-missing access to a virtual address VA, the hardware first reads address RA_4, whose address bits 11-3 are equal to VA bits 47-39. After that, the page walker reads address RA_3, whose 11-3 address bits are equal to VA bits 38-30. Then, the page walker reads RA_2 and RA_1. Bits 2-0 of RA_4, RA_3, RA_2, and RA_1 are 000 because these loads read 8-byte page table entries, which are aligned to 8-byte boundaries. If any of these four addresses has a false dependence with the address *WA* written by the victim, a long latency access ensues. The false dependence occurs when two addresses have the same bits 11-3 because we issue 8-byte loads—i.e., the two addresses have 4K aliasing (Section 2). Hence, this attack can learn bits 11-3 of the victim store address, which are the page offset bits with sub-cacheline granularity (Figure 1c).

Figure 3 shows simplified programs that demonstrate the store→load channel. The demonstration extracts bits 11-3 of a store address in a victim program. As shown in Figure 3a, the victim program allocates a page and keeps writing to it at a fixed page offset (i.e., `0x528`), which is a secret. As shown in Figure 3b, the attacker program first allocates 512 continuous

```
1  const u32 secret_offset = 0x528;
2  char *page = mmap(NULL, PAGE_SIZE, ...);
3  while (true) {
4    page[secret_offset] = 0xff;
5  }
```

(a) Victim program.

```
1  const u32 npages = 512;
2  u32 latencies[npages];
3  const u64 size = PAGE_SIZE * npages;
4  char *base_page = mmap(NULL, size, ...);
5  for (u32 i = 0; i < npages; i++) {
6    char *page = base_page + i * PAGE_SIZE;
7    invalidate_tlb(page);
8    u64 t_start = read_timestamp();
9    maccess(page);
10   u64 t_end = read_timestamp();
11   u32 PL1_index = ((u64)page & 0x1ff000) >> 12;
12   latencies[PL1_index] = t_end - t_start;
13 }
```

(b) Attacker program.

Figure 3: Simplified programs that demonstrate the store→load channel.

Figure 4: Demonstration of the store→load channel on a Skylake-X.

pages (Line 4). The page table entries (PTEs) of these 512 pages fill a 4KB PL_1 page, since each PTE is 8 bytes. Then, for each page, the attacker flushes the page's translation from TLBs and issues an access to the page, which triggers a page walk. The page walk of one of the pages will issue a load to a RA_1 address whose bits 11-3 match bits 11-3 in the victim's *WA* address. Because of this 4K aliasing, the latency of the access to this particular page will be higher.

We run both programs on two different hyperthreads of the same physical core. For each page of the attacker, we measure the page access latency 100 times and use the average value. The page walker read of each of the 512 pages tests a different 8-byte aligned address offset within a 4KB page. Figure 4 shows the average latency measured at each PL_1 offset on a Skylake-X. As the plot shows, the page access latencies are very low at most PL_1 offsets. When the PL_1 offset gets close to the victim's secret offset, $0x528$, the access latency starts to increase, and it reaches its peak value when the PL_1 offset exactly matches the secret offset. We obtain similar results on a Haswell-EP and a Cascade Lake-X.

The reason why the peak is not sharper is that page walker loads can also be stalled by stores that access the same L1 cache set. In this case, the two addresses only need to share bits 11-6. As we will see in Section 5, this second type of false dependence is harder to induce.

To maximize the number of different offsets monitored by a single TLB-missing access, the attacker can carefully allocate a page at an address that has a different PL offset at each level. In this case, as shown in Figure 1b, the attacker can theoretically monitor up to four different offsets, using the page walker loads from PL_4, PL_3, PL_2, and PL_1. However, in the current implementation of x86-64, an unprivileged user can only allocate pages in the lower half of the 64-bit VA space (Section 2), which means that the attacker does not have full control of the PL_4 index (i.e., bits 47-39 of the VA) and cannot use it to monitor arbitrary store offsets.

In addition, since we assume an unprivileged attacker (Section 3), the attacker cannot use privileged instructions to flush the TLB. Instead, to evict a target translation from the TLB, she has to build an eviction set of pages. Note that the hash function used in Skylake-X to map a page to a set in the TLB uses the PL_1 index and part of the PL_2 index (i.e., bits 26-12 of the VA) [28]. As a result, to build an eviction set, the attacker uses pages with different PL_3 indexes but the same PL_2 and PL_1 indexes. Consequently, the attacker cannot typically use PL_3 indexes to monitor store offsets, and has to limit herself to monitoring two offsets (using PL_2 and PL_1 indexes) with each TLB-missing access.

4.2 Leaking the Virtual Page Number of the Address of an Access

In this attack, the attacker repeatedly stores to a given offset in a page and the victim suffers a TLB miss that triggers a page walk. The attacker keeps changing the page offset of the store address and observes the victim's performance. When the victim's access latency is high, one of the page walk loads is 4K-aliasing with the attacker's store. The attacker can then learn the *PL* index of a level of the page table entry. By continuing to change the page offset of the store address, the attacker can recover the *PL* indexes of all the different page levels. As a result, as we will see later, the attacker will be able to learn the full virtual page number (VPN) of the victim's TLB-missing memory accesses (i.e., bits 47-12). Because the information in this attack flows from victim page walker loads to attacker stores, we call this primitive the **load→store channel**.

The process of the attack is shown in Figure 1b. A page walk issues, in the worst case, loads to addresses RA_4, RA_3, RA_2, and RA_1. Each of these addresses includes, in bits 11-3, a portion of the VA of the page accessed. When one of these addresses has the same bits 11-3 bits as the attacker's store address (*WA*)—i.e., it 4K-alias with *WA*—the victim's access suffers a long latency. Based on the observed latency, the attacker can deduce the four sets of 11-3 bits in some order.

```
1  const u64 addr = 0x5d21ca821000ull;
2  char *page = mmap(addr, PAGE_SIZE, ...);
3  while (true) {
4    wait_for_attacker();
5    invalidate_tlb(page);
6    u64 t_start = read_timestamp();
7    maccess(page);
8    u64 t_end = read_timestamp();
9    u64 t_diff = t_end - t_start;
10   // signal the attacker process; pass t_diff
11   signal_attacker(t_diff);
12 }
```

(a) Victim program.

```
1  const u32 nindexes = 512;
2  u32 latencies[nindexes];
3  char *page = mmap(NULL, PAGE_SIZE, ...);
4  for (u32 idx = 0; idx < nindexes; idx++) {
5    u32 offset = idx << 3;
6    signal_victim(); // signal the victim process
7    while (wait_for_victim()) {
8      page[offset] = 0xff;
9    }
10   latencies[idx] = get_victim_latency();
11 }
```

(b) Attacker program.

Figure 5: Simplified programs that demonstrate the load→store channel.

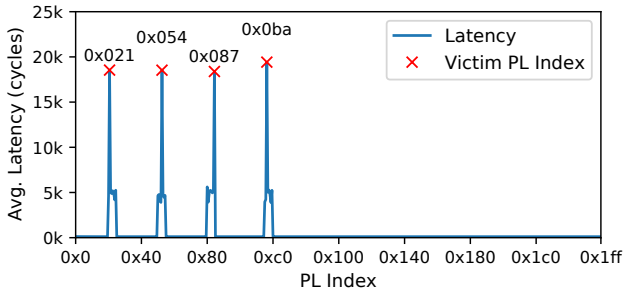


Figure 6: Demonstration of the load→store channel on a Skylake-X.

With some additional experiments that will be detailed later, the attacker can put together the whole VPN of the victim access (Figure 1c).

Figure 5 shows simplified programs that demonstrate the load→store channel. To measure the latency of the victim's access, the code unrealistically assumes that attacker and victim processes can communicate via shared memory to synchronize, and that the victim measures its own latency and reports it to the attacker process. This setting is for demonstration only. A realistic setting will be shown in Section 8, where the attacker only relies on the end-to-end execution time of the victim.

The victim program (Figure 5a) first allocates a page at virtual address `0x5d21ca821000`, which corresponds to indexes to PL_4, PL_3, PL_2, and PL_1 equal to `0x0ba`, `0x087`, `0x054`, and

```
1  // Attacker program, port contention version
2  const u32 nindexes = 512;
3  u32 latencies[nindexes];
4  char *page = mmap(NULL, PAGE_SIZE, ...);
5  for (u32 idx = 0; idx < nindexes; idx++) {
6    u32 offset = idx << 3;
7    u64 t_start = read_timestamp();
8    for (u32 i = 0; i < 10000; i++) {
9      page[offset] = 0xff;
10   }
11   u64 t_end = read_timestamp();
12   latencies[idx] = t_end - t_start;
13 }
```

Figure 7: Port-contention version of the load→store channel attack program.

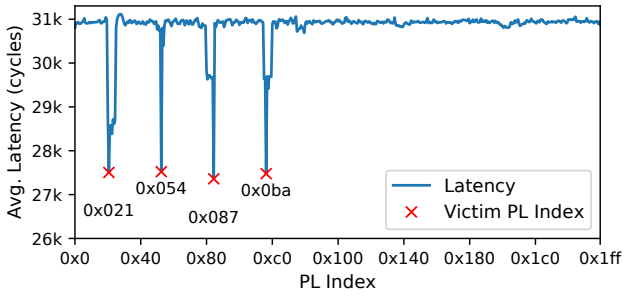


Figure 8: Demonstration of the load→store channel with the port-contention version on a Skylake-X.

0x021, respectively. Then, the victim program enters a loop where, in each iteration, the victim: (i) waits for the attacker to signal it, (ii) invalidates the translation of the page from the TLBs, (iii) accesses the page and measures the access latency, and (iv) signals the attacker, passing the access latency. The attacker program (Figure 5b) first allocates buffers for latency results and a page to write to. Then, it enters a loop that iterates over all the possible 512 indexes of page table entries (PTEs) in a page. For each of the resulting PL address offsets, the attacker: (i) signals the victim process, (ii) keeps writing to an address at the PL offset in the page until the victim sends it a signal, and (iv) receives the latency of the victim access and saves it.

We run both programs on two hyperthreads of a physical core. For each PL index, we measure the latency 100 times and save the average value. Figure 6 shows the resulting average latency for each PL index on a Skylake-X processor. Looking at the figure, we see there are four clear latency spikes. They are at indexes 0x021, 0x054, 0x087, and 0x0ba. These four spikes correspond to the four 9-bit PL indexes of the victim page. We obtain similar results on a Haswell-EP and a Cascade Lake-X.

From these latency results alone, we cannot determine which spike corresponds to which page table level. The full VPN is one of the permutations of these four indexes. There are multiple strategies to identify the correct permutation. For example, if we know which memory region the victim accesses (e.g., heap or stack), we can identify the possible PL_4 or even PL_3 indexes, since these memory regions usually have unique ranges of high-order VA bits. If the victim also happens to access neighboring pages (i.e., pages that differ in PL_1 indexes), the attacker should observe nearby spikes, and these spikes correspond to PL_1 indexes. After determining the PL_4, PL_3, and PL_1 indexes, we know which one is the PL_2 index. Last, as will be shown in Section 8, if the memory access is to a global variable, the PL_1 index can be derived from the variable's offset in the segment.

We can easily redesign the attack so that attacker and victim do not need to synchronize, and the victim does not need to measure the latency of its own accesses. Instead, the attacker measures the latency of its stores. The idea is that, when the victim's page walker load is stalled for a long time due to 4K aliasing, the victim's pipeline is blocked, and shared resources are freed-up for the attacker. As a result, the attacker sees lower latency for its own stores because of less port contention. Consequently, in Figure 7, we change the code from Figure 5 so that, in each iteration, the attacker measures the latency of issuing 10,000 stores—and neither synchronizes nor receives any latency measurement from the victim. This is a more realistic design. Figure 8 shows the average latency of those 10,000 stores at different indexes on a Skylake-X. It is clear that latencies drop at the victim's PL indexes.

4.3 Extensions

Cross Virtual Machine Attack. Binoculars also works if attacker and victim are in two different virtual machines that share the same physical core. Because in a virtualized environment, a TLB-missing access also triggers page walker loads, which are subject to contention with stores from the sibling thread. To verify this, we repeat our experiments in a virtualized environment with QEMU-KVM (4.2.1) [56] on Skylake-X. The attacker and victim programs run in two different virtual machines that share the same physical core and run Ubuntu 20.04 LTS (5.4.0-105-generic). Our experiments show results that are similar to the ones in a non-virtualized environment, and thus demonstrate Binoculars can be used for cross virtual machine attacks.

Other Paging Schemes. The previous discussion is mainly focused on a 4-level paging design, which issues four page walker loads to translate a VPN. If other paging schemes (e.g., huge page, 5-level paging) are used, Binoculars will still work with different attacker capabilities.

For the store→load channel, the attacker has no incentive to use huge pages: doing so would reduce the number of page walker loads and correspondingly the number of page offsets that can be observed in a single page walk. Using 5-level paging, on the other hand, can boost the attack as one page walk can monitor five store offsets. For the load→store channel, using huge pages reduces the number of low-order VPN bits that an attacker can extract. But it is still possible to attack (kernel) ASLR, as its entropy usually resides in high-

Table 1: Comparing the characteristics of different side-channel attacks.

Attack	Timing Difference*	Virtual Address Bits Leaked	Leakage Granularity	Cross Address Space?	Cross Core?
PortSmash [4], SMoTherSpectre [9]	10^{-1} cycles	n/a	μOps	Yes	No
TLBleed [28]	10^{1} cycles	Bits 26-12†	Memory page	Yes	No
CacheBleed [72], MemJam [44]	10^{1} cycles	Bits 11-2	Sub-cacheline	Yes	No
Prime+Probe [49]	10^{2} cycles	n/a	Cache set	Yes	Yes
Flush+Reload [71]	10^{2} cycles	Offset in segment	Cacheline	No	Yes
AVX2-Based [61]	10^{2} cycles	n/a	AVX2 instruction	Yes	No‡
Binoculars store→load channel	10^{4} cycles	Bits 11-3	Sub-cacheline	Yes	No
Binoculars load→store channel	10^{4} cycles	Bits 47-12	Memory page	Yes	No

* Magnitude of the maximum timing difference that can be caused by a *single dynamic instruction or operation* that the attack uses.
† Applicable to an Intel Skylake-X platform [28]. Actual bits may vary on different microarchitectures.
‡ Our threat model (Section 3) only considers local hardware resource utilization changes caused by the victim. Since AVX2-based attacks exploit power management mechanisms that only affect each individual physical core, we do not consider it as a cross-core channel under our threat model.

order VPN bits (see Section 8). Using 5-level paging, the attacker will observe five latency spikes associated with each level of translation, instead of four spikes shown in Figure 6 and 8, which makes finding the correct permutation of spikes and recover the full VPN harder.

Other CPUs. We believe the root cause of the Binoculars attack is related to Intel's optimization of page walker loads (see Section 5 and Appendix A). Therefore, Binoculars is likely exclusive to Intel processors. For example, the same experiments on an AMD-EPYC-7502 processor show that it does not exhibit the Binoculars channel.

4.4 Discussion

Table 1 compares the characteristics of Binoculars and existing side-channel attacks. From left to right, we compare: (i) the maximum timing difference that can be induced by a *single dynamic instruction or operation* that the attack uses, (ii) the virtual address bits leaked, (iii) the granularity of the leakage, and whether the attack is effective (iv) across address spaces or (v) across cores.

The second column shows the first advantage of *Binoculars*: its contention effect is very strong. *A single dynamic instruction* can create timing differences that are several orders of magnitude higher than in any other conventional side-channel attack. For example, with port contention, a dynamic instruction can cause a latency increase equal to a fraction of a cycle on average [4, 9]. Therefore, an attacker requires thousands of dynamic instructions to magnify the effects, or tens of thousands of replays to denoise the channel [63]. In CacheBleed [72] and MemJam [44], a dynamic instruction exploiting a false dependence causes a 10-cycle average latency increase. In side-channel attacks that rely on timing differences between cached and uncached accesses such as TLBleed, Prime+Probe, and Flush+Reload, the differences range from tens of cycles [28] to a few hundred cycles [49,71]. AVX2-based side channel that is exploited in NetSpectre [61] can also cause a timing difference of a few hundred cycles. Instead, a Binoculars access can trigger a stall of up to 20,000 cycles. This property makes Binoculars more resilient to noise, which means that it can recover secrets with fewer runs and with a higher confidence. Also, because of the long duration of the stall, it is possible to observe the contention even if the attacker cannot measure the time very precisely—e.g., due to lacking a high-resolution timer.

The third column of Table 1 shows a second advantage of *Binoculars*: it leaks a wide range of virtual address (VA) bits. The column lists the VA bits leaked by each attack. For example, TLBleed [28] can recover the bits that are used by TLB hash functions (i.e., bits 26-12 on a Skylake-X for the sTLB). Hence, TLBleed can observe only a victim's memory accesses at a page granularity, and cannot extract the full VPN. CacheBleed [72] and MemJam [44] recover low-order intra-cacheline bits (i.e., bits 11-2), but miss out on high-order bits. In Flush+Reload [71], because of ASLR, a shared memory segment can be allocated at different VA in different processes. As a result, Flush+Reload can only recover VA bits that are not subject to ASLR, namely the offset to the base of the segment—at a cache line granularity. With Binoculars, the attacker can learn VA bits 11-3 with the store→load channel and bits 47-12 (i.e., the full VPN) with the load→store channel.

The fourth column of Table 1 shows the leakage granularity of each attack. Binoculars provides sub-cacheline resolution with the store→load channel and page-level granularity with the load→store channel. The fifth and sixth columns show whether the attack works across address spaces and across physical cores. Since Binoculars requires no shared memory, it can attack a victim running in a different address space. However, it cannot attack a victim on a different core.

Finally, Binoculars has a third, although not unique, advantage: it does not require complex state preparation. For example, most cache-based side-channel attacks require the victim data to be present at a given level of the memory hierarchy before the attack. For that, the attacker has to carefully manipulate cache state, which is slow, sometimes complex, and requires fine-grained synchronization with the victim process. Binoculars only needs page walkers to trigger loads. Contention occurs regardless of what level in the cache hierarchy the page walker loads read from.

5 Root Cause Analysis

The resource contention observed by a single Binoculars access can reach up to 20,000 cycles on a Skylake-X processor. The magnitude of this contention has no precedent in existing contention-based attacks. For this reason, we explored its root cause, relying on (sometimes undocumented) performance counters and Intel's patents, similar to prior work [35, 45, 60].

We find that the contention is the result of an optimization in the Intel processor hardware that issues page walker loads as "stuffed" loads. These loads bypass the RSs and the ROB to avoid their scheduling latency [26]. We hypothesize that this optimization has an unexpected side effect: when the implicit stuffed loads conflict with explicit data stores in the L1D cache, the scheduler cannot detect such conflicts. Consequently, the scheduler simply allows the explicit data stores from one hyperthread to run "at full speed", without realizing that these stores are repeatedly squashing the stuffed loads from the other hyperthread. Hence, the page walker suffers from resource starvation and eventually triggers a mechanism that aborts and restarts the page walk—presumably with a higher priority.

Our further experiments lead us to conclude that both set conflicts (i.e., page walker loads and data stores trying to access the same L1D set) and false dependences (i.e., page walker loads and data stores 4K-aliasing with each other) can cause stuffed loads to be squashed by data stores in the L1D cache. Further, the impact of these two events depends on the writer thread's behavior. Specifically, when the writer thread stores at a high frequency, set conflicts occur frequently and dominate false dependences. Instead, when the store frequency is reduced, false dependences begin to dominate set conflicts. Appendix A summarizes our findings.

6 Binoculars Covert Channel

A covert channel is a communication channel that allows two cooperating parties to bypass system policies to communicate with each other. Most covert channels are synchronous, where the transmission process is divided into time epochs for synchronization. In every epoch, the sender encodes one or several bits of information by changing microarchitectural states, while the receiver decodes the information by observing the changes. Depending on the mechanism used by the covert channel, in every epoch, after the receiver decodes the transmission, it may need to precondition the channel for the next epoch [10]. To keep the sender and the receiver well-synchronized, an epoch has to be long enough to cover the encoding and decoding operations, and the potential preconditioning operations.

A robust metric to measure a covert channel's transmission capability (i.e., its raw throughput and bit-error rate) is the *channel capacity* [43]. This metric measures the highest rate of reliable information transmission that a communication channel supports. It is computed by $r \times (1 - H(p))$, where r is the raw throughput of the channel, p is the probability of a bit error, and H is the binary entropy function. Using this formula, we can see that a high-capacity covert channel requires a high raw throughput (which is determined by the length of an epoch and the number of bits it can transmit per epoch) and a low bit-error rate (which is determined by the noise in the channel). This metric is also used in some prior work [47, 50, 55].

A straw man Binoculars covert channel works as follows. Before the transmission, sender and receiver agree on a page offset. To send a bit 1, the sender keeps writing to the agreed offset until the end of the epoch. To send a bit 0, the sender does nothing and waits for the next epoch. To decode the information, the receiver issues and times a TLB-missing memory access to a target page, whose page walk includes a load that 4K-aliases with the sender write. If the receiver measures a high access latency, the sender is sending a bit 1; otherwise, it is sending a bit 0. After that, to precondition the channel, the receiver accesses a TLB eviction set to evict the target page from the TLB.

Unfortunately, this straw man scheme does not achieve a high channel capacity. Since the page walker loads can be stalled for up to 20,000 cycles (Section 4), an epoch has to be longer than that, which drastically limits the channel capacity. Fortunately, in practice, such large stall times are unnecessary to build a low error-rate covert channel. Therefore, we carefully tune the number of stores that are executed by the sender. We want to make sure that these stores can create reliably-high timing differences on the receiver side while keeping the epochs short.

To further improve the channel capacity, on the receiver side, we build a large TLB eviction set using methods similar to ones in [28, 64]. At every epoch, the receiver chooses the target page from that set. Moreover, the chosen target page is different from the target page used in the previous epoch. With this design, we can ensure that the read access to the target page not only decodes the information, but also evicts the translation of the page that will be used as the target in the next epoch. This design eliminates the need of preconditioning the channel through explicit eviction of TLB entries. Hence, we can support an even shorter epoch.

Finally, we also make sure that all the pages in the TLB eviction set are mapped to the same physical page. As a result, accessing them one after another does not evict their data from the caches, which removes any noise due to cache misses.

We evaluate the average capacity of the Binoculars covert channel on Intel Haswell-EP, Skylake-X, and Cascade Lake-X platforms. In each platform, we run the sender and the receiver for 100 times to transmit a 1MB-long randomly-generated message. Table 2 lists the channel capacity of Binoculars on each platform, as well as the capacities of prior covert channels. From the table, we see that on a Haswell-EP, Binoculars attains a moderate channel capacity of 177 KB/s with a 1.2% bit error rate. On a Skylake-X, the channel capacity increases

Table 2: Comparison of covert channels with average capacity higher than 100 KB/s.

Attack	Channel Capacity	Cross Address Space?	Cross Core?
Streamline [58]	1733 KB/s	No	Yes
Lord of the Ring(s) [50]	518 KB/s	Yes	Yes
Take-a-Way [41]	505 KB/s	Yes	No
Flush+Flush [32]	463 KB/s	No	Yes
L1 Prime+Probe [54]	400 KB/s	Yes	No
Flush+Reload [32,71]	298 KB/s	No	Yes
Binoculars (Cascade Lake-X)	1116 KB/s	Yes	No
Binoculars (Skylake-X)	622 KB/s	Yes	No
Binoculars (Haswell-EP)	177 KB/s	Yes	No

to 622 KB/s and the bit error rate decreases to 0.9%. Finally, on a Cascade Lake-X, which is one of the latest Intel server microarchitectures, the capacity reaches 1116 KB/s with a low bit error rate of 0.6%.

The main reason for the large channel capacity variations across different platforms is the processor performance. In newer generations, processors can execute the same number of stores in a shorter period of time, and these stores can also cause more contention effects. As a result, a newer processor requires fewer stores to cause enough contention and a shorter epoch to execute them. For example, on a Haswell-EP, we need to execute 380 stores on the sender side every epoch. On a Cascade Lake-X, the number is reduced to only 80 stores, which only require a 420-cycle epoch.

Table 2 also lists the characteristics of existing covert channels with a channel capacity greater than 100 KB/s. Among all these channels, Binoculars has the second highest channel capacity[2] (on Skylake-X and Cascade Lake-X), and is only behind Streamline [58]. Although Streamline has a higher channel capacity and supports cross-core communication, Binoculars does not require shared memory thus works across address spaces.

7 Attacking Montgomery Ladder and ECDSA

We use Binoculars to obtain the private key used by OpenSSL's ECDSA implementation. Our attack targets the Montgomery ladder, a widely used optimization for computing scalar multiplication on elliptic curves [39]. OpenSSL's ECDSA implementation uses the Montgomery ladder to calculate the point $k \times G$ during signing, where the scalar k is a nonce (i.e., an ephemeral key). Our goal is to learn the nonce k, which together with the signature can be used to derive the private key used for signing [37,46].

Figure 9 shows the Montgomery ladder implementation used in OpenSSL 1.0.1e. The code iterates over the bits of k. In each iteration, it performs an elliptic curve point addition and doubling by calling the functions `gf2m_Madd` and `gf2m_Mdouble`, respectively. The current bit, k_i, determines

[2]Parallel to our work, TLB;DR [64] built a more performant covert channel with an average channel capacity of 1375 KB/s.

[3]The code is from function `ec_GF2m_montgomery_point_multiply` at `crypto/ec/ec2_mult.c:268` [48].

```
1  for (; i >= 0; i--) {
2    word = scalar->d[i];
3    while (mask) {
4      if (word & mask) { // checks ki
5        // compute (x1,z1)=(x1/z1)+(x2/z2)
6        if (!gf2m_Madd(group,&point->X,x1,z1,x2,z2,ctx))
7          goto err;
8        // compute (x2,z2)=2*(x2/z2)
9        if (!gf2m_Mdouble(group, x2, z2, ctx))
10         goto err;
11     } else {
12       if (!gf2m_Madd(group,&point->X,x2,z2,x1,z1,ctx))
13         goto err;
14       if (!gf2m_Mdouble(group, x1, z1, ctx))
15         goto err;
16     }
17     mask >>= 1;
18   }
19   mask = BN_TBIT;
20 }
```

Figure 9: Montgomery ladder implementation used in OpenSSL 1.0.1e[3].

the big number variables written to in each step. If $k_i = 1$, (x_1, z_1) is added to and (x_2, z_2) is doubled; if $k_i = 0$, the order is reversed. In the following, we denote function calls to `gf2m_Madd` and `gf2m_Mdouble` under the $k_i = 1$ direction as `Madd1` (Line 6) and `Mdouble1` (Line 9), and the calls under the $k_i = 0$ direction as `Madd0` (Line 12) and `Mdouble0` (Line 14).

While this implementation is data-oblivious to the sequence of operations and end-to-end timing, it nevertheless has a secret-dependent order of stores to (x_1, z_1) and (x_2, z_2). Since stores to these two pairs of variables have different page offsets, *Binoculars can identify the store order by monitoring stores to these offsets, and thereby recover the k_i values.*

Challenge. The nonce k in ECDSA changes at every run and never repeats. An attacker only has *one chance* to capture a k and cannot rely on repeated runs to de-noise the channel. As a result, the attacker needs a side channel with an extremely high signal-to-noise ratio to exfiltrate k with a single victim run. There exist partial key recovery techniques for ECDSA [8,21,25,46] that allow an attacker to reconstruct the private key from multiple signatures and part of corresponding ks. However, most of them assume that the known parts of ks are error free [25], or at least that they have a low bit-error rate (e.g., less than 2% [8,21]).

7.1 Attack Method

We assume the attacker can obtain a signature from the victim (e.g., by making a network request) and use Binoculars to monitor the victim's signing execution. The attacker's process does not need to share any physical memory with the victim.

Our attack infers the value of k_i using the store→load channel to monitor the order of stores to (x_1, z_1) and (x_2, z_2). To

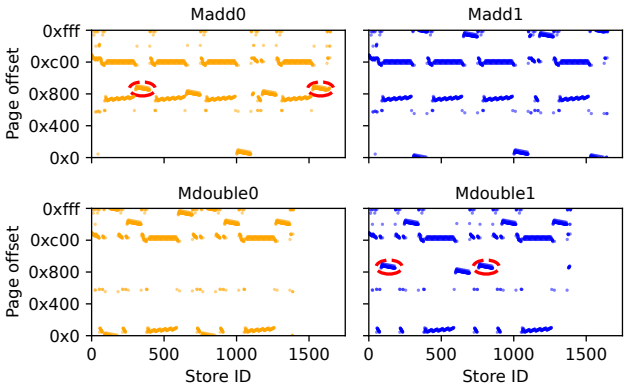


Figure 10: Page offsets of stores in `Madd0`, `Mdouble0`, `Madd1`, and `Mdouble1`. Red circles highlight stores to variable x_2. The traces are collected with Intel Pin [57] on a Skylake-X.

avoid monitoring four variables, we only monitor stores to a single variable (e.g., x_2). We infer the value of k_i based on whether the stores happen in the first half of a Montgomery ladder iteration (in which `Madd0` executes) or in the second half of an iteration (in which `Mdouble1` executes).

Crucially, we design a realistic end-to-end attack that does not assume synchronization between victim and attacker, such as knowing when Montgomery ladder iterations begin and end. Our attack method contains the following three steps:

Step 1: Identify Target Store Page Offsets. Finding store page offsets to monitor can be done offline. Page offsets depend only on memory allocation details, which are fixed for a given environment. (In particular, they are independent of ASLR, which only randomizes high-order bits of addresses.) This means that running the same OpenSSL as the victim in the same environment is sufficient for the attacker to find suitable offsets for using in a later online attack.

To minimize noise, we prefer offsets that are exclusively used by one of the four variables (e.g., x_2). Figure 10 shows traces of page offset stored to by `Madd0`, `Mdouble0`, `Madd1`, and `Mdouble1`, obtained using Intel's Pin tool [57]. The X axis is the order of stores inside the function and the Y axis is the page offset of each store. Stores to x_2 are highlighted with red circles. Since x_2 is a big number that occupies multiple double words, offsets of stores to x_2 form a continuous descent "slope" with a step of 8 bytes in the figure, instead of a single point. We find that x_2's offsets are good candidates for low-noise monitoring, as only `Madd0` and `Mdouble1` store to these offsets, and only when writing to x_2.

Step 2: Monitor Victim Stores. The attacker process is co-located on the same physical core as the victim, but on a sibling hyperthread. While the victim is signing, the attacker process keeps recording the latency of TLB-missing loads that monitor stores to the chosen page offsets, as well as the timestamp of each measurement. The latency trace is then saved for the next step.

Figure 10 shows that for a given offset of x_2, only a few

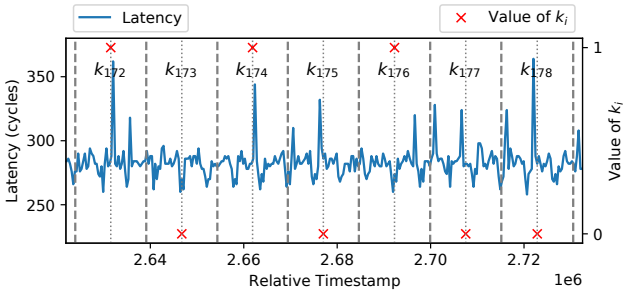


Figure 11: A snippet of measured raw latencies. Grey vertical dashed lines indicate start of Montgomery ladder iterations. Grey vertical doted lines are the halves of iterations. Red crosses are the ground truth of k. Timestamps are relative to the first Montgomery ladder iteration. Measured on a Skylake-X.

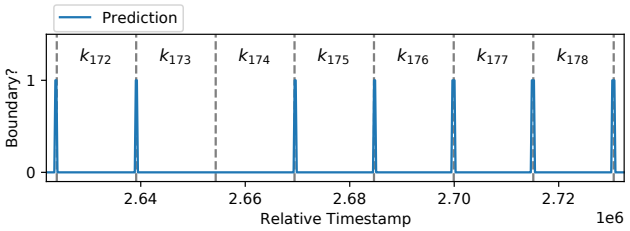


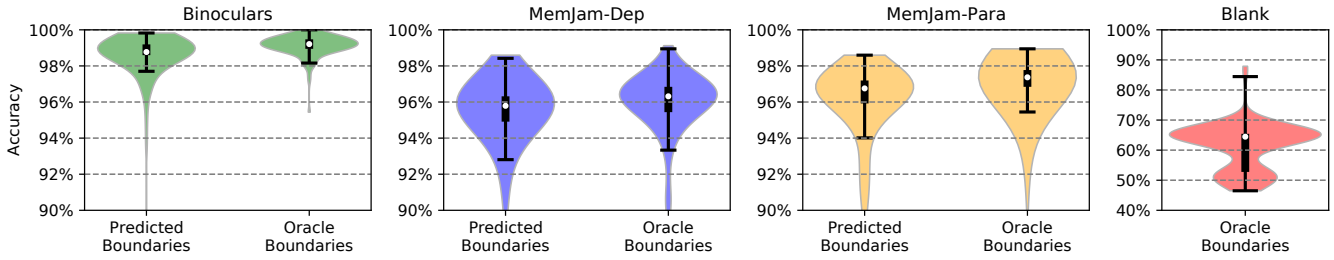
Figure 12: Montgomery ladder iteration boundaries predicted by the classifier on the trace in Figure 11. Timestamps are relative to the first Montgomery ladder iteration.

stores are executed in a short period during signing. To obtain a detectable signal in the store→load channel from these stores, we time four dependent TLB-missing loads instead of one. Since a single TLB-missing load can monitor two unique offsets (Section 4.1), these four loads can monitor eight neighboring offsets of x_2 to increase the chance of contention. Also, because these four loads are dependent, their contention effects are built up and observable.

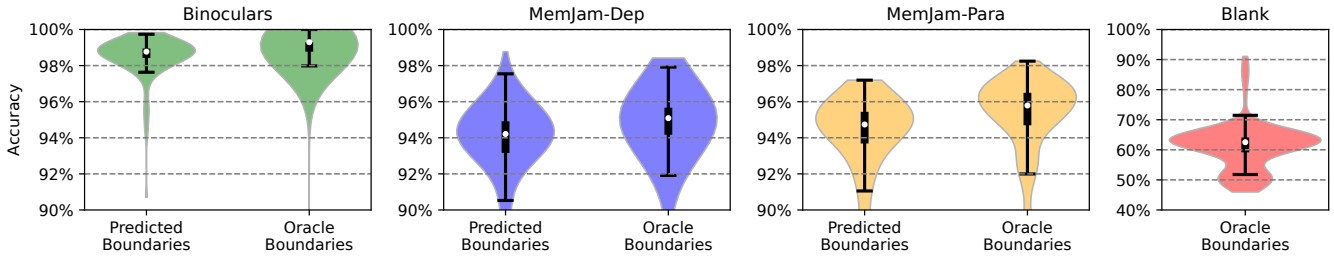
Step 3: Process Signal. To recover the k_i values from the latency trace collected in step 2, we need to (1) identify when the victim stores to x_2; and (2) find boundaries of Montgomery ladder iterations, so that we can know whether the stores are performed in the first or second half of an iteration.

Figure 11 shows a snippet of a latency trace collected on a Skylake-X, while the attacker monitors stores to x_2. We thus expect to see latency increases when the victim executes `Madd0` or `Mdouble1`. Red crosses show the k_i values. Iteration boundaries and halves are marked by vertical grey dashed lines and thin dotted lines, respectively. The lines are plotted by instrumenting the victim, *but in a real attack, they must be recovered by the attacker from the trace.* On average, the probing latency is around 300 cycles on the test machine.

Using the vertical lines as references, we see that when k_i is 0, there are two high latencies events in the first half of the Montgomery ladder iteration (bits k_{175}, k_{177}, and k_{178}). And when k_i is 1, high latency events occur in the second half of



(a) Skylake-X.



(b) Cascade Lake-X.

Figure 13: Violin plots of each approach's accuracy distribution on a Skylake-X and a Cascade Lake-X. Each distribution contains 100 predictions. BLANK with predicted boundaries are omitted since the boundary predictor fails to output any meaningful data for it. All plots except BLANK share the same Y axis range starting from 90%.

an iteration (bits k_{172}, k_{174}, and k_{176}).

Not all contention effects are obvious, however (e.g., bit k_{173}). We therefore use a supervised machine learning model, random forest classifier [51, 52], to predict iteration boundaries and k_i values from the latency trace. The following details how the model is used.

Preprocess. Because the attacker process keeps measuring latencies without any synchronization, the trace forms an unevenly spaced time series. We transfer the data into an equally spaced time series by resampling the raw data at a fixed period with linear interpolation. Since most machine learning models work best when the data under classification roughly follows a standard normal distribution, we normalize the resampled latencies by subtracting the mean and then dividing by the standard deviation.

Predict iteration boundaries. To recover Montgomery ladder iteration boundaries from the latency trace, we use a binary random forest classifier. Our classifier takes as input a vector of 160 normalized latencies and predicts whether the center of the vector is an iteration boundary.

Figure 12 shows the classifier's outputs for the trace snippet in Figure 11. The blue line is the classifier output on each timestamp and the grey vertical dashed lines are the ground truth of iteration boundaries. While the classifier manages to recover most boundaries, it sometimes misses a boundary (e.g., between k_{173} and k_{174}). We overcome this problem by, ironically, exploiting the Montgomery ladder's constant-time property. Because it executes the same sequence of operations regardless of k_i, the iteration length is relatively constant. We can therefore estimate the average iteration period from the predictions and use it to fix missing boundaries and remove false positives.

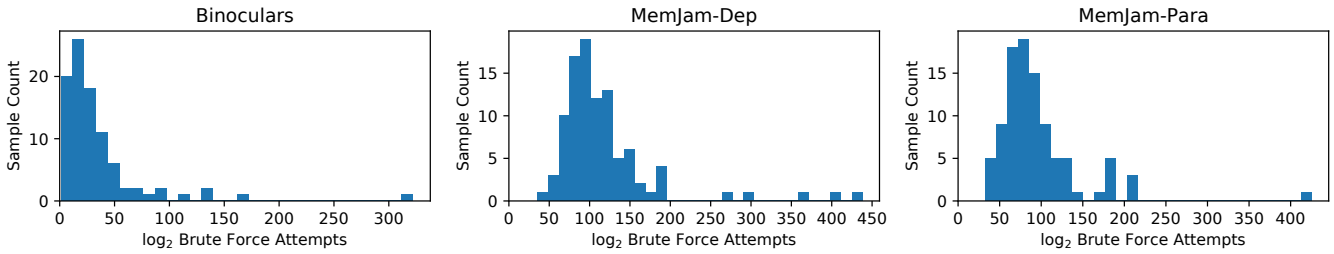
Predict k_i. Finally, we train another random forest classifier that takes as input a vector of normalized latencies from an iteration i, and predicts the value of k_i. For each prediction, the classifier also outputs a confidence score.

7.2 Results

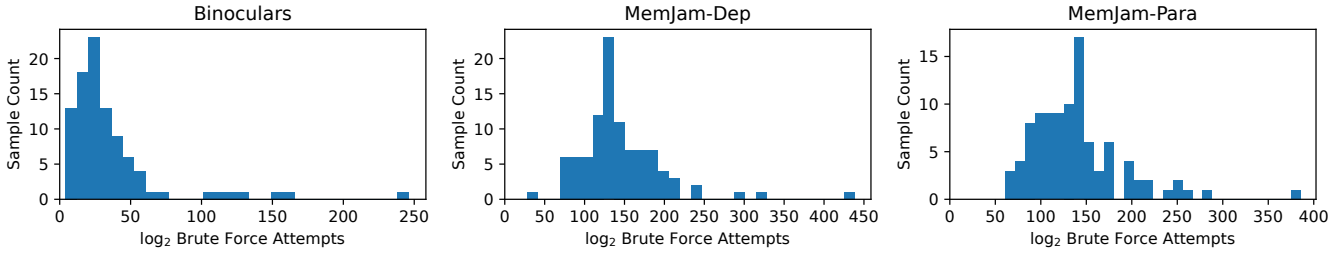
Setup. We evaluate our attack method on a Skylake-X and a Cascade Lake-X with OpenSSL 1.0.1e on Ubuntu 20.04 LTS (5.4.0-105-generic). We use OpenSSL 1.0.1e strictly as a demonstration benchmark; after version 1.0.1e, OpenSSL switched to an invulnerable branchless Montgomery ladder implementation. We configure cores to run in performance mode without fixing their frequency. Cores for experiments are isolated to minimize context switches. We use the default compilation flags to compile OpenSSL. The curve that we are targeting is `sect571r1`, which uses a 571-bit nonce. We use the binary random forest classifier machine learning model[4] from scikit-learn 1.0.2 [52] for signal processing.

We evaluate the attack's end-to-end accuracy with three other approaches for monitoring victim stores (Step 2) besides Binoculars, while keeping the rest of the steps same: (1) BLANK: the attack process only measures time. This approach serves as a sanity check to show that the signal we observe is not caused by any resource contention on reading the timestamp. (2) MEMJAM-PARA: this approach relies on false dependence in the L1D cache (Section 2) that is described in MemJam [44]. Similar to the setup in MemJam, we

[4] Model parameters: `n_estimators=400, min_samples_split=5, min_samples_leaf=1, max_features="log2", max_depth=30`



(a) Skylake-X.



(b) Cascade Lake-X.

Figure 14: Required $\log_2$ brute force attempts to recover the full nonce k with the improved method (Based on k_i predictions with *predicted boundaries*).

measure the latency of eight parallel loads that are 4K-aliasing with a target store offset. (3) MEMJAM-DEP: this approach replaces eight parallel loads in MEMJAM-PARA with four dependent loads to enhance the contention. (4) BINOCULARS (*this work*): this approach relies on the store→load channel.

For each approach, we collect 100 latency traces to train the first random forest classifier that predicts Montgomery ladder iteration boundaries. This step takes about 1 minute to collect and process the traces (about 500k training samples), and 10 minutes to train on a 16-core machine. Then, we use another 30 traces to train the second random forest classifier that predicts k_i. This step takes about 2 minutes to collect, process, and train on the same machine (about 17k training samples). Finally, we evaluate the end-to-end accuracy of each method on another 100 traces and discuss the security implications (i.e., nonce recovery). Note that we do not include results on a Haswell-EP, because none of these four approaches can achieve good accuracy with a single victim run on it, mainly due to its low performance and thus weak contention effects.

Accuracy. Figure 13 shows the accuracy distribution of each approach on different CPU platforms. For each approach, we show the results with iteration boundaries predicted by the boundary classifier and with oracle boundaries. The exception is BLANK, in which we only show results with oracle boundaries since the boundary classifier cannot output any meaningful prediction for its traces. For each distribution, the area represents the density of samples at a given accuracy. A wider area means the corresponding accuracy is more likely to occur. The thick short black bar represents the first to third quartile range. The thin long black bar represents the lower and upper bounds after filtering outliers with the quartile range. The white dot represents the median of the distribution.

BINOCULARS has the highest accuracies on both CPU platforms. On average, its accuracies are $98.5 \pm 0.3\%$ on a Skylake-X and $98.4 \pm 0.3\%$ on a Cascade Lake-X ($N = 100, P = 0.95$, same N and P value below), or with oracle boundaries, $99.1 \pm 0.1\%$ on a Skylake-X and $98.7 \pm 0.8\%$ on a Cascade Lake-X. Using oracle boundaries, sometimes BINOCULARS can even recover the full nonce without any error. Compared to BINOCULARS, other approaches have lower accuracy and higher deviation. MEMJAM-PARA, on average, can achieve $96.0 \pm 0.6\%$ on a Skylake-X and $94.1 \pm 0.5\%$ on a Cascade Lake-X. On average, MEMJAM-DEP's accuracies are $94.9 \pm 0.7\%$ on a Skylake-X and $93.8 \pm 0.6\%$ on a Cascade Lake-X. BLANK achieves accuracies that are only slightly better than random guessing (i.e., 50%), indicating—as expected—that just reading the timestamp cannot reveal k_i.

Nonce Recovery. To completely recover the full 571-bit-long nonce k, we need to find and correct erroneous bits in the predictions through brute forcing. Since we do not know how many bits are incorrect and where those bits are, we have to first guess the number of erroneous bits n_e, starting from 1, then try to flip all $C_{571}^{n_e}$ combinations. If no correct solutions are found, we will increment n_e and repeat the process.

To speed up the brute force search, we improve the method based on an observation that most erroneous bits have low confidence scores. In the improved method, we first sort all the predicted bits by their confidence scores in an ascending order. Then, we pick the top N_L low-confident bits and try to flip all $C_{N_L}^{n_e}$ combinations for a given n_e. Since N_L can be much smaller than the bit-length of k (i.e., 571), the search space is significantly reduced. Note that if the N_L low-confident bits

fail to cover some erroneous bits, the brute force will fail. In that case, we will increase N_L and retry.

Figure 14 shows histograms of $\log_2$ brute force attempts with the improved method. These histograms are based on k_i predictions with predicted Montgomery ladder iteration boundaries. On a Skylake-X (Figure 14a), BINOCULARS requires a median of $2^{23.4}$ brute force attempts to recover k, which is feasible. If we assume an acceptable brute force attempts threshold at 2^{40}, BINOCULARS can succeed on 78.5% of traces. However, MEMJAM-DEP and MEMJAM-PARA require many more brute force attempts. Their median attempts are $2^{101.6}$ and $2^{82.0}$ respectively. With the same brute force threshold, they can only recover 1.0% and 3.1% of traces.

On a Cascade Lake-X (Figure 14b), every approach requires slightly more brute force attempts. From left to right, BINOCULARS requires $2^{24.7}$ median brute force attempts, and recovers k on 77.9% of traces with a brute force threshold of 2^{40}. While MEMJAM-DEP and MEMJAM-PARA require $2^{130.8}$ and $2^{132.5}$ median brute force attempts. Under the same brute force threshold, they can only recover 1.0% and 0.0% of traces.

8 Compromising KASLR

Linux uses kernel address space layout randomization (KASLR) to increase the difficulty of exploiting memory safety vulnerabilities. Linux randomizes the base addresses of several kernel memory regions at boot time. Since the possible kernel's address range is 1 GB (30 bits) and base addresses are aligned to 2 MB boundaries (21 bits), Linux's KASLR has 9 bits of entropy (512 choices) [14]. The address bits randomized are bits 29-21, i.e., the PL_2 index.

Attack Method. Using the load→store channel, an attacker can recover the full virtual page number (VPN) of a TLB-missing victim (kernel) memory load. Assuming the offset of the accessed page within its kernel segment is known, the base address of the segment can be derived, breaking KASLR.

We implement this idea by attacking a system call that accesses a global variable, whose offset within the kernel image is known for a given kernel build. We choose the `SYS_time` system call (similar to prior work [41]) which accesses the global variable `tk_core`. The attack is similar to the setup in Figure 6, with the difference that the attacker measures the end-to-end execution time of the victim. The attacker runs two hyperthreads on the same physical core. The first hyperthread flushes the TLBs and measures the latency of calling `SYS_time`, while the second hyperthread keeps writing to each possible *PL* offset. Because the TLBs are flushed, the system call's read of `tk_core` will miss in TLBs and trigger a page walk. Consequently, the attacker will observe system call latency spikes at offsets that are 4K-aliasing with any PL index of `tk_core`. To minimize noise caused by irrelevant TLB-missing loads, the first thread makes an invalid system call to warm up the system call handler before calling `SYS_time`.

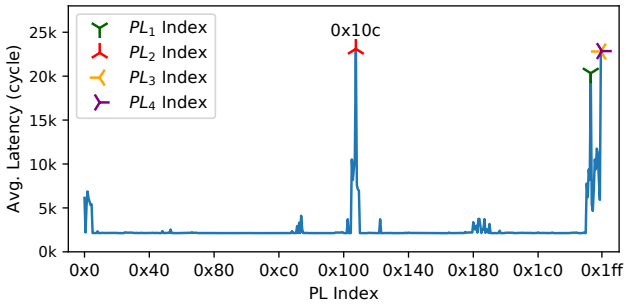


Figure 15: Average latency of calling `SYS_time` when varying store offsets on a Skylake-X.

Binoculars is fundamentally different from most prior KASLR attacks [13, 14, 31, 34, 36, 59] which rely on monitoring microarchitectural side effects of accessing a mapped or unmapped address. Such attacks can be defeated by software mitigations like FLARE [14], which creates fake mappings for all possible kernel addresses, or kernel page-table isolation (KPTI), which unmaps most kernel pages in user space [30]. In contrast, Binoculars directly observes the kernel's TLB-missed memory loads, which allows the attacker to break KASLR even if KPTI or FLARE is deployed. Compared to prior work that also monitors victim's memory accesses [41], Binoculars can completely break KASLR thanks to its wide address bits coverage, in contrast to reducing entropy in [41].

Results. We use the same hardware set as in Section 4, running Ubuntu 20.04 LTS (5.4.0-105-generic). On Haswell-EP and Skylake-X platforms, which are vulnerable to Meltdown, KPTI is enabled. We rely on `/proc/kallsyms` to collect the ground truth of the global variable `tk_core`'s address.

Figure 15 shows the average latency of calling `SYS_time`, measured at each offset on a Skylake-X. The global variable `tk_core` is located at `0xffffffffa19f4f40` in this boot, which corresponds to indexes to PL_4, PL_3, PL_2, and PL_1 equal to `0x1ff`, `0x1fe`, `0x10c`, and `0x1f4`, respectively, which are marked in the figure. While the latencies are huge at these indexes, we do not see peaks as sharp as the ones in Figure 6 due to TLB-missing accesses that are not to `tk_core`. We therefore run a peak detection algorithm.

To identify which peak corresponds to `tk_core`'s PL_2 index, we need to discard the PL_4, PL_3, and PL_1 indexes. We use the fact that the PL_4 and PL_3 indexes are the same constants in any Linux kernel image base address. Moreover, we can learn `tk_core`'s PL_1 index from its offset in the kernel image, which is known for a given kernel build (and readable in the image's symbol table).

To measure the accuracy of identifying the PL_2 index, we reboot the system 10 times, and recover the index 100 times per boot (1000 recoveries in total). We achieve accuracies of 100.0%, 98.7%, and 92.6% on the Skylake-X, Haswell-EP, and Cascade Lake-X, respectively. These accuracies, however, are of a single attacker run. They can be trivially improved to 100% on all three platforms by repeating the runs and picking

the most frequent PL_2 index guess.

9 Potential Mitigations

The root cause of the Binoculars attack is the starvation of hardware page walker loads by concurrent stores due to false dependence and/or set conflicts in the L1D cache. A complete fix of Binoculars thus requires hardware-level changes.

Software-wise, two mitigations can be applied. A system can disable hyperthreading, or only allow mutally trusted programs to share a physical core (e.g., core scheduling [40]). However, this mitigation can under-utilize hardware resources and lead to system performance degradation [15]. Alternatively, potential victims can be rewritten with data-oblivious programming practices [5, 17, 73], so that they do not make secret-dependent memory accesses. But while data-oblivious code is invulnerable to Binoculars and many other side channels, it usually requires a non-trivial amount of effort to rewrite and verify the program, and incurs significant execution overhead.

10 Related Work

Attacks on Montgomery ladder. There have been several attacks on Montgomery ladder. Yarom et al. [70] extend Flush+Reload [71] to attack the same vulnerable implementation in Figure 9. Compared to Binoculars, their attack requires the attacker to share memory with the victim, and has lower average accuracy (95.7%).

Brumley et al. [11] attack a different part of the OpenSSL implementation to recover the logarithm of k, which is then used in a lattice attack to recover the private key. Unlike Binoculars, this attack requires thousands of signatures and its success rate is low.

Exploiting page walker loads. Page walker loads go through the cache hierarchy and so can be observed with cache-based side channels. There have been side-channel attacks exploiting page walker loads to build stateful-indirect channels [12, 29, 66, 67]. Using these channels, the attack's granularity is limited to a cache line (64 bytes), which means that it cannot distinguish accesses to neighboring pages, whose 8-byte PTEs share the cache line with the target page. Binoculars is also capable of performing such monitoring with the load→store channel and has a finer granularity.

11 Conclusion

In this paper, we investigated and demonstrated the first stateless-indirect channel by exploiting interactions between in-flight page walk loads on behalf of one thread and stores by another thread. We introduced a new side-channel attack called *Binoculars*. Unlike conventional stateless channels, Binoculars creates significant timing perturbations (e.g., up to 20,000 cycles)—making it easy to monitor. We showed that the perturbations are address dependent, and designed two Binoculars attack primitives to leak a wide range of virtual address bits in victim memory operations. Using these primitives, we demonstrated end-to-end attacks on real hardware, which include extracting the nonce k in ECDSA with a single victim run, and fully breaking kernel ASLR.

Disclosure. We disclosed the Binoculars side channel vulnerabilities to Intel in November 2021. Following our report, Intel considers our findings covered by their guidelines for mitigating timing side channels against cryptographic implementations [18].

Acknowledgments

This research was funded by an Intel Resilient Architectures and Robust Electronics (RARE) gift. We thank the anonymous reviewers for their valuable feedback. We also thank Jinyang Li for his suggestions on signal processing.

Availability

We open sourced Binoculars PoC at https://github.com/zzrcxb/binoculars.

A Appendix: Detailed Root Cause Analysis

In this appendix, we explore the root cause of the strong resource contention triggered by Binoculars (up to 20,000 cycles on a Skylake-X processor, Section 4).

Source of the Contention. We first confirm that the contention originates from the page walk triggered by TLB misses. To this end, we monitor various TLB- and page-walker-related performance counter sub-events (Table 3) during an experiment similar to the one in Figure 2. In the experiment, one hyperthread performs a TLB-missing page access while the sibling hyperthread keeps writing to an address that is, (or is not), 4K-aliasing with one of the page walker loads. Before the measurement, we first warm up the page by accessing it multiple times so that its data is cached; then, we invalidate its translation from all the TLB levels.

Table 4 shows performance counter values collected on an Intel Skylake-X for both the 4K-aliasing and no-aliasing cases. If the store is not aliasing with the page walker load, the page walker starts and completes one page walk to handle the TLB miss (`MISS_CAUSES_A_WALK` and `WALK_COMPLETED` events, respectively), which takes 42 core clock cycles (`WALK_DURATION`), and all its page walker loads plus the data access hit in the L1D cache. It takes 180 core clock cycles in total to complete the page walk, the data access, and then stop the performance counters.

In the 4K-aliasing case, however, the page walker starts two page walks but only finishes one. Also, the sub-event `WALK_DURATION` has a value that is close to `Unhalted Core Cycles`. These results indicate that the core spends most its cycles servicing the page walk, which takes a long time to complete. Also, the page walk seems to be aborted and restarted. The counters for page walker loads all indicate

Table 3: List of performance counter events.

Parent Event	Sub-event	Description
DTLB_LOAD_MISSES	MISS_CAUSES_A_WALK	Number of page walks (including incomplete walks)
DTLB_LOAD_MISSES	WALK_COMPLETED	Number of completed page walks
DTLB_LOAD_MISSES	WALK_DURATION†	Count of *core clock cycles* when the page walker is servicing page walks
PAGE_WALKER_LOADS	DTLB_L1*	Number of page walker loads that hit in L1D+Fill Buffer
PAGE_WALKER_LOADS	DTLB_L2*	Number of page walker loads that hit in L2
PAGE_WALKER_LOADS	DTLB_L3*	Number of page walker loads that hit in L3
PAGE_WALKER_LOADS	DTLB_MEMORY*	Number of page walker loads that read from main memory
MEM_LOAD_RETIRED	L1_HIT	Number of load instructions that hit in L1D (excluding page walker loads)
n/a	Unhalted Core Cycles†	Count of *core clock cycles* when the core is running

* Although these sub-events are only documented for Haswell-EP and Broadwell-EP, we find that they still exist and are functional on newer microarchitectures like Skylake-X and Cascade Lake-X.

† These sub-events count *core clock cycles*, which are subject to turbo-boost. The rest of the paper uses *reference clock cycles*, which are not.

Table 4: Performance counter values on a Skylake-X.

Sub-event	No Aliasing	4K-Aliasing
MISS_CAUSES_A_WALK	1	2
WALK_COMPLETED	1	1
WALK_DURATION (core clock cycles)	42	16452*
DTLB_L1	4	4
DTLB_L2	0	0
DTLB_L3	0	0
DTLB_MEMORY	0	0
L1_HIT	1	1
Unhalted Core Cycles (core clock cycles)	180	16584*

* These core clock cycles correspond to $\approx 20,000$ reference clock cycles.

there are no L1D misses, which means that the slow page walk is not caused by cache-missing loads. These observations confirm that the contention indeed comes from the page walker. We hypothesize that the contention is so strong that it leads to resource starvation of the page walker, which triggers a "watchdog" to abort the page walk and restart it with a higher priority over shared resources.

Cause of Starvation. To validate our starvation hypothesis, we rely on Intel's patents on virtual memory translation. According to one of Intel's patents, the page walker issues "stuffed" loads that bypass the RS and the ROB [26]. This mechanism is presented as an optimization to avoid any scheduling latency that the RS or the ROB may cause.

After the stuffed load is dispatched by the page walker, it is handled by the memory-order buffer (MOB). The MOB checks for potential conflicts with pending stores—-i.e., whether a store may be writing to the address read by the stuffed load. If a potential conflict is found, the page walker aborts the walk and retries when the conflict is resolved. Although this might sound like the root cause of the contention, our further experimentation finds that only stores from *the same thread* can cause conflicts, as the MOB is not shared by the two hyperthreads, which disproves this explanation.

If the MOB finds no conflicts, the stuffed load is issued to the L1D cache. In this step, the L1D cache may "squash" the stuffed load under certain circumstances. If the squash happens, the page walker will re-dispatch the stuffed load as soon as possible, and the re-dispatched stuffed load may get squashed by the L1D cache again. *This behavior can starve the stuffed load indefinitely*. As will be discussed later, we indeed find a performance counter sub-event that suggests that the L1D cache receives thousands of read requests from the stuffed load during a stalled page walk.

Magnitude of Starvation. Given that the L1D cache rejects data accesses for various reasons (Section 2), why can Binoculars stall a page walk for up to $20,000$ cycles while attacks like CacheBleed and MemJam only delay a data access for a few cycles? We hypothesize the answer is related to instruction scheduling differences.

In CacheBleed and MemJam, the conflicts are between explicit data loads and stores. Data loads and stores are processed by the ROB and the RS, and are scheduled by the same Out-of-Order (OoO) engine of the physical core. The OoO engine can therefore detect and mediate between the conflicts after a few failed L1D accesses. In Binoculars, however, the conflicts are between implicit stuffed page walker loads and explicit data stores. Because stuffed loads are managed outside of the RS and the ROB, we hypothesize that the OoO engine cannot detect such conflicts. Consequently, the OoO engine simply allows the explicit data stores from the other hyperthread to run "at full speed", without realizing that one hyperthread is trying to perform a page walk and failing, as its stuffed loads are getting squashed. The page walker thus suffers from resource starvation and eventually triggers a mechanism that aborts and restarts the page walk (presumably with a higher priority).

Cause of L1D Squashes. We find that both set conflicts and false dependences can cause stuffed loads to be squashed by the L1D cache, depending on the writing thread's behavior. Our analysis here is based on identifying *undocumented* performance counters for these events.

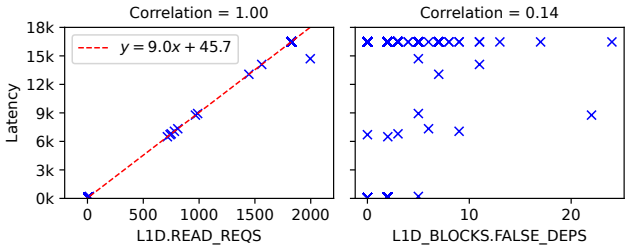
To identify relevant counters, we perform a brute force search over all possible counter sub-events, searching for the ones that are highly correlated with the access latency of TLB-missing loads. We perform the search by trying every combination of the two 1-byte-long fields, `EventSel` and `UMask`, which determine the sub-event in the performance counter configuration model-specific registers [19]. Our search finds two interesting undocumented sub-events: (1) `EventSel=0x51, UMask=0x20` and (2) `EventSel=0xbf,`

UMask=0x01. Based on the EventSels of these two sub-events and our reverse engineering, the first sub-event likely counts the number of L1D read requests, including both successful and squashed requests. The second sub-event likely counts the number of failed L1D read requests due to false dependences (Section 2). In the rest of our discussion, we will refer to these two sub-events as L1D.READ_REQS and L1D_BLOCKS.FALSE_DEPS respectively. We also find that the L1D.READ_REQS sub-event is present only on Haswell-EP but not on newer generations. Therefore, we will focus on results on Haswell-EP for the rest of the discussion.

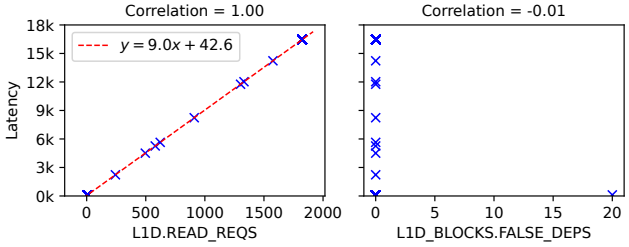
We perform three experiments to understand whether Binoculars L1D squashes are due to 4K-aliasing or L1D set conflicts. The experiments monitor these undocumented sub-events as one hyperthread performs a TLB-missing memory access, which triggers a page walk that reads from RA_4, RA_3, RA_2, and RA_1, while its sibling hyperthread keeps writing to an address WA. The experiments differ in the bits shared by RA and WA, and in the frequency of writing to WA: (1) BINOCULARS-4K: RA_1 and WA share bits 11-3 (i.e., 4K-aliasing); (2) BINOCULARS-SAMESET: RA_1 and WA share bits 11-6 but differ in bits 5-3 (i.e., they are mapped to the same L1D cache set); (3) BINOCULARS-4K-LOWFREQ: RA_1 and WA share bits 11-3, but the writer thread has a reduced write frequency, as it executes arithmetic instructions between writes. We ensure that the page walker loads, the data load, and the stores only access up to two unique cache lines in an L1D set, i.e., fewer than the associativity of the L1D cache. We repeat each experiment 1000 times.

Figure 16 shows the results on a Haswell-EP in reference clock cycles (the maximum stall on a Haswell-EP is around 16,000 cycles). The red dashed lines are fitted linear regression lines. In the BINOCULARS-4K experiment (Figure 16a), the access latency is strongly correlated to the number of L1D read requests, which confirms that the stuffed page walker loads are repeatedly squashed by the L1D cache and re-dispatched by the page walker. From the fitted line, on average, it takes 9 cycles to squash and retry a stuffed load. However, looking at the right plot of Figure 16a, the correlation between the access latency and L1D_BLOCKS.FALSE_DEPS is very low, which suggests false dependences are not the main cause of L1D squashes in this experiment.

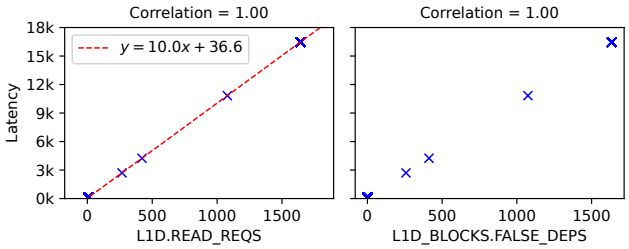
The BINOCULARS-SAMESET experiment (Figure 16b) still shows many high-latency events that are correlated to L1D.READ_REQS. Compared to BINOCULARS-4K, however, it has significantly fewer events that reach the maximum 16,000-cycle latency (131/1000 events in BINOCULARS-SAMESET versus 847/1000 events in BINOCULARS-4K). Also as expected, L1D_BLOCKS.FALSE_DEPS is always 0 because the page walker load is not 4K-aliasing with stores. This experiment shows that without false dependences, contention and even starvation can still occur as long as the RA_1 and the WA are mapped to the same L1D cache set (i.e., they suffer *set conflicts*). Hovewer, they occur less frequently than they would in BINOCULARS-4K. Recall that we also see a similar behavior in Figure 4.



(a) BINOCULARS-4K.



(b) BINOCULARS-SAMESET.



(c) BINOCULARS-4K-LOWFREQ.

Figure 16: Scatter plot between the access latency and the two undocumented sub-events. All plots share the same Y axes.

Finally, in the BINOCULARS-4K-LOWFREQ experiment (Figure 16c), the latency is still strongly correlated to L1D.READ_REQS and it can reach the maximum 16,000-cycle latency. But now it takes 10 cycles to squash and retry. Also, the latency becomes strongly correlated to L1D_BLOCKS.FALSE_DEPS, which means that false dependences in the L1D cache become the main reason of L1D squashes when the writer thread has a reduced frequency.

The results of the three experiments lead us to conclude that both set conflicts and false dependences can cause stuffed loads to be squashed by the L1D cache, depending on the writer thread's behavior. We believe that set conflicts only happen in an early stage of a read access, while false dependences occur in a later stage. This explains the one-cycle difference in the squash-and-retry latency. We believe that set conflicts require stricter timing requirements to trigger (e.g., that read and write requests arrive at the same cycle) compared to false dependences. Finally, we believe that set conflicts (when they occur) dominate false dependences—i.e., when a set conflict occurs, a false dependence will not happen.

The above explain the results we see. When stores are frequent (Figures 16a and 16b), set conflicts are more likely

to occur. This explains Figure 16a (set conflicts occur frequently and dominate false dependences when they do) and Figure 16b (in which set conflicts occur frequently and false dependences are impossible). This also explains that in Figure 16c set conflicts are less likely and thus false dependences dominate. Finally, this explains why starvation occurs less frequently in BINOCULARS-SAMESET than in BINOCULARS-4K: since a read request can "slip through" set conflicts due to the strict timing requirements, high latency is hard to build up in BINOCULARS-SAMESET. In BINOCULARS-4K, the "slipped-through" request will likely be squashed due to a false dependence in the next cycle, which makes the starvation more likely to happen.

References

[1] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18, 2007.

[2] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *ct-rsa*, 2008.

[3] Onur Aciicmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, pages 80–91. IEEE, 2007.

[4] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida Garcia, and N. Tuveri. Port contention for fun and profit. In *IEEE S&P*, pages 870–887, May 2019.

[5] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security*, 2016.

[6] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *IEEE S&P*, May 2015.

[7] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *IEEE S&P*, pages 591–604, 2015.

[8] Diego F Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. Ladderleak: Breaking ecdsa with less than one bit of nonce leakage. In *CCS*, pages 225–242, 2020.

[9] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *CCS*, pages 785–800, 2019.

[10] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. Casa: End-to-end quantitative security analysis of randomly mapped caches. In *MICRO*, pages 1110–1123. IEEE, 2020.

[11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *ESORICS*, pages 355–371. Springer, 2011.

[12] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, pages 1041–1056, 2017.

[13] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*, pages 769–784, 2019.

[14] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *AsiaCCS*, pages 481–493, 2020.

[15] Shawn Casey. How to determine the effectiveness of hyper-threading technology with an application. *Intel Technology Journal*, 6(1):11, 2011.

[16] Yun Chen, Lingfeng Pei, and Trevor E Carlson. Leaking control flow information via the hardware prefetcher. *arXiv preprint arXiv:2109.00474*, 2021.

[17] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *IEEE S&P'09*, 2009.

[18] Intel Corparation. Guidelines for mitigating timing side channels against cryptographic implementations. *Nov*, 2021.

[19] Intel Corparation. Intel 64 and ia-32 architectures software developer's manual. *Combined Volumes, Dec*, 2021.

[20] Patrick Cronin and Chengmo Yang. A fetching tale: Covert communication with the hardware prefetcher. In *HOST*, pages 101–110, 2019.

[21] Gabrielle De Micheli, Rémi Piau, and Cécile Pierrot. A tale of three signatures: practical attack of ecdsa with wnaf. In *International Conference on Cryptology in Africa*, pages 361–381. Springer, 2020.

[22] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. Prime+abort: A timer-free high-precision L3 cache attack using intel TSX. In *USENIX Security*, pages 51–67, 2017.

[23] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, 2016.

[24] Dmitry Evtyushkin and Dmitry V. Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, pages 843–857, 2016.

[25] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking openssl implementation of ecdsa with a few signatures. In *CCS*, pages 1505–1515, 2016.

[26] Andy Glew, Glenn Hinton, and Haitham Akkary. Method and apparatus for performing page table walks in a microprocessor capable of processing speculative instructions, 1997. US Patent 5,680,565.

[27] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

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

[29] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, February 2017.

[30] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *ESSoS*, 2017.

[31] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *CCS*, pages 368–379, 2016.

[32] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[33] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, pages 897–912, 2015.

[34] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE S&P*, pages 191–205, 2013.

[35] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: speculative load hazards boost rowhammer and cache attacks. In *USENIX Security*, pages 621–637, 2019.

[36] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *CCS*, pages 380–392, 2016.

[37] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[38] Mike Johnson. *Superscalar microprocessor design*. Prentice Hall series in innovative technology. Prentice Hall, 1991.

[39] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *International workshop on cryptographic hardware and embedded systems*, pages 291–302. Springer, 2002.

[40] Linux. Core Scheduling; The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/core-scheduling.html, 2022.

[41] Moritz Lipp, Vedad Hažić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of amd's cache way predictors. In *AsiaCCS*, pages 813–825, 2020.

[42] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE S&P*, 2015.

[43] David JC MacKay and David JC Mac Kay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[44] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 47(4):538–570, 2019.

[45] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *USENIX Security*, pages 1427–1444, 2020.

[46] Phong Q Nguyen and Igor E Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, codes and cryptography*, 30(2):201–217, 2003.

[47] Hamed Okhravi, Stanley Bak, and Samuel T King. Design, implementation and evaluation of covert channel attacks. In *HST*, pages 481–487, 2010.

[48] OpenSSL. Montgomery Ladder Implementation of OpenSSL 1.0.1e. https://github.com/openssl/openssl/blob/46ebd9e3bb623d3c15ef2203038956f3f7213620/crypto/ec/ec2_mult.c#L268, 2011.

[49] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20, 2006.

[50] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security 2021*, pages 645–662, 2021.

[51] Mahesh Pal. Random forest classifier for remote sensing classification. *International journal of remote sensing*, 26(1):217–222, 2005.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[53] Colin Percival. Cache missing for fun and profit. In *Proc. of the Technical BSD Conference (BSDCan)*, 2005.

[54] Colin Percival. Cache missing for fun and profit, 2005.

[55] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security*, 2016.

[56] QEMU. QEMU version 4.2.0 released. https://www.qemu.org/2019/12/13/qemu-4-2-0/, 2019.

[57] Vijay Janapa Reddi, Alex Settle, Daniel A Connors, and Robert S Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *WCAE*, pages 22–es, 2004.

[58] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *ASPLOS*, pages 1077–1090, 2021.

[59] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-leak forwarding: Leaking data on meltdown-resistant cpus. *arXiv preprint arXiv:1905.05725*, pages 15–20, 2019.

[60] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *CCS*, pages 753–768, 2019.

[61] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *ESORICS*, pages 279–299, 2019.

[62] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *CCS*, page 131–145, 2018.

[63] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. Microscope: Enabling microarchitectural replay attacks. In *ISCA*, pages 318–331. IEEE, 2019.

[64] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In *USENIX Security*, 2022.

[65] Robert M Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Res. and Dev.*, pages 25–33, 1967.

[66] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *USENIX Security*, pages 937–954, 2018.

[67] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches. In *EuroSec*, April 2017.

[68] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *ACSAC*, pages 473–482, 2006.

[69] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security*, pages 159–173, 2012.

[70] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptol. ePrint Arch.*, 2014:140, 2014.

[71] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[72] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[73] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *NDSS*, 2019.

[74] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *CCS*, pages 305–316, 2012.

[75] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *MICRO*, pages 28–41. IEEE, 2020.