SpectreRewind: Leaking Secrets to Past Instructions

Jacob Fustos, Michael Bechtel, Heechul Yun
University of Kansas

ABSTRACT

Transient execution attacks use microarchitectural covert channels to leak secrets that should not have been accessible during logical program execution. Commonly used micro-architectural covert channels are those that leave lasting footprints in the micro-architectural state, for example, a cache state change, from which the secret is recovered after the transient execution is completed.

In this paper, we present SpectreRewind, a new approach to create and exploit contention-based covert channels for transient execution attacks. In our approach, a covert channel is established by issuing the necessary instructions logically *before* the transiently executed victim code. Unlike prior contention based covert channels, which require simultaneous multi-threading (SMT), SpectreRewind supports covert channels based on a single hardware thread, making it viable on systems where the attacker cannot utilize SMT. We show that contention on the floating point division unit on commodity processors can be used to create a high-performance (~100 KB/s), low-noise covert channel for transient execution attacks instead of commonly used flush+reload based cache covert channels. We also show that the proposed covert channel works in the JavaScript sandbox environment of a Chrome browser.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Browser security**.

KEYWORDS

Spectre; Micro-architecture; Side-channel Attack

ACM Reference Format:
Jacob Fustos, Michael Bechtel, Heechul Yun. 2020. SpectreRewind: Leaking Secrets to Past Instructions. In *4th Workshop on Attacks and Solutions in Hardware Security (ASHES'20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3411504.3421216

1 INTRODUCTION

Modern out-of-order microprocessors support speculative execution to improve performance. In speculative execution, instructions can be executed speculatively before knowing whether they are in the correct program execution path. If the speculation was wrong, the instructions that were executed incorrectly—known as transient instructions [18]—are squashed and the processor then simply retries to fetch and execute the correct instruction stream. Unfortunately, these transient instructions can potentially bypass both software and hardware defenses to access secret data. The disclosure of Spectre [18], Meltdown [20] and many other subsequently discovered transient execution attacks [1, 9, 14, 15, 17, 19, 21, 22, 27, 29, 34–37, 40] have shown the danger of these transient instructions. Namely, the secrets they have access to can be encoded and transmitted into microarchitectural covert channels and subsequently recovered by normal, non-speculative instructions, thus allowing the secrets to be visible to the attacker.

All known transient execution attacks share the same three basic steps: (1) the attacker initiates speculative execution where the secret is read improperly from memory or registers; (2) the secret dependent transient instructions then encode and transmit the secret to a micro-architectural covert channel; (3) finally, the secret is recovered from the covert channel by normal (non-transient) receiver instructions. Commonly used covert channels, such as cache, are stateful as they leave lasting footprints in the micro-architectural state, from which the secret is recovered after the transient execution is completed. Many hardware defense proposals aim to prevent such stateful covert channels either by hiding the changes into additional hardware buffers [12, 16, 41] or by reverting them when the transient instructions are squashed [25]. Such a mitigation strategy is attractive from a performance standpoint, as the transient instructions are allowed to execute normally, retaining many of the performance benefits of speculative execution.

These types of defenses are effective at blocking transient execution attacks that utilize stateful covert channels. Unfortunately, these techniques cannot be used to block attacks that both transmit into and read from covert channels before transient instructions have been squashed. SmotherSpectre [6] is the first to demonstrate such in the context of a Spectre-based attack. By generating contention on issue ports within SMT processors, SmotherSpectre is able to create a covert channel that can transmit a secret between the SMT threads. Such contention cannot be buffered or reverted, as instructions have already waited to use the issue ports, affecting their execution time.

In this paper, we present SpectreRewind, a new approach to create and utilize contention-based covert channels in transient execution attacks. Like SmotherSpectre, SpectreRewind allows the attacker to both transmit and receive secret data before transient execution has completed, allowing the attacker to bypass most defense mechanisms that attempt to revert or hide micro-architectural changes caused by the attack. However, unlike SmotherSpectre, SpectreRewind does not require the attacker to utilize SMT, instead the attack can be executed from a single hardware thread. While traditional transient execution attacks locate the instructions that will read from the covert channel logically *after* the instruction that triggers the transient execution (e.g., a branch), SpectreRewind takes the opposite approach and locates these instructions logically

ASHES'20, November 13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8090-4/20/11…$15.00
https://doi.org/10.1145/3411504.3421216

before the triggering instruction. This structure allows the transmitting and receiving instructions to execute concurrently on a modern out-of-order core and communicate the secret even before the transient execution completes.

We identify that non-pipelined functional units can be exploited to create SpectreRewind covert channels. In particular, we show that contention on the floating point division unit in commodity Intel, AMD, and ARM processors can create high bandwidth (~100KB/s), low-noise (<0.01%) covert channels that are comparable to commonly used cache-based covert channels. We also show that the feasibility of our covert channel within a Chrome browser's JavaScript sandbox.

In summary, we make the following **contributions**:

- We introduce SpectreRewind, a new approach to create and exploit contention based covert channels in transient execution attacks within the same hardware thread.
- We show that contention on non-pipelined floating point division unit can create a high-bandwidth, low-noise covert channel on commodity out-of-order processors.
- We demonstrate that the floating point division unit based covert channel works in a JavaScript sandbox of Google Chrome browser.

2 BACKGROUND

In this section, we provide necessary background on out-of-order cores, transient execution attacks, and simultaneous multithreading (SMT) hardware.

2.1 Out-of-order Processors

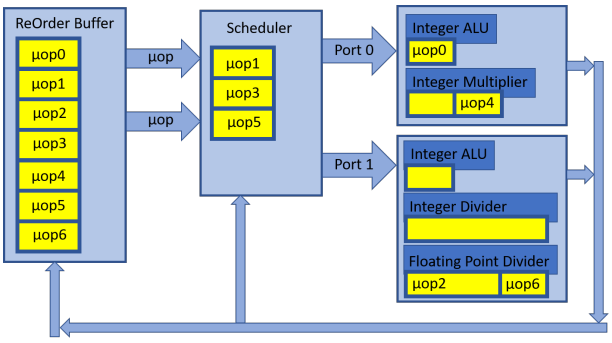


Figure 1: Simplified out-of-order processor design. The ReOrder Buffer holds and retires μops in logical program order, while μops are issued to the execution units in out-of-order.

Modern high performance microprocessors implement out-of-order execution to maximize instruction level parallelism and performance.

Figure 1 shows a simplified example of an out-of-order processor. In this example, instructions are translated into micro-operations (μops) and placed into the ReOrder Buffer (ROB) in logical program order. They are then passed to the scheduler which issues them to a proper functional unit when their operands and the necessary resources are available. In this example, the functional units are clustered into two execution units. Each execution unit contains a single issue port, which can only issue a single μop to one of the enclosed functional units every clock cycle. Once issued, the functional units run independent of each other. When an μop is executed by a functional unit, the scheduler is notified so that it can forward the results to any following dependent μops. The μop then waits in the ROB until it reaches the head where it may be retired. It is only now that the changes made by the μop become architecturally visible, giving the illusion—from the architecture's point of view—that the instructions are executed in-order.

To further reduce branch related stalls, modern processors implement speculative execution, which uses various branch predictors to predict future instructions (those in the predicted execution paths) and speculatively execute them even before the correct execution paths are known. If the prediction turns out to be incorrect, these speculatively executed instructions are squashed and the processor resumes executing the correct instructions. The instructions that were executed and later squashed are known as transient instructions.

2.2 Transient Execution Attacks

Transient execution attacks exploit the side-effects of executing transient instructions. While transient instructions do not retire—and do not become architecturally visible—they still can alter microarchitectural states through which secret can be leaked.

Known transient execution attacks can be largely grouped into two categories: Spectre and Meltdown types. Spectre type attacks utilize control and data-flow mis-speculation to force a victim to access secrets from their own address space and leak them into the covert channel where they can be accessed by the attacker. Each Spectre variant [14, 17, 18, 18, 19, 21] is distinguished by the microarchitectural component that is responsible for causing the mis-speculation namely—Branch History Buffer (BHB), Branch Target Buffer (BTB), Memory Disambiguator, and Return Stack Buffer (RSB).

Meltdown style attacks take advantage of "bugs" in deferred exception/fault handling in some (mainly Intel) processors. Each Meltdown variant [1, 15, 17, 20, 29, 34, 40] corresponds to the exception that caused the fault. Microarchitectural Data Sampling (MDS) [22, 27, 36] are also considered Meltdown-type attacks. These attacks target speculative loads that have incorrectly loaded data from internal buffers—Store Buffer, Load Port, Line Fill Buffer—and leak the data into covert channels before realizing the fault. The data that was incorrectly loaded could have come from other SMT threads on the same processor executing at any privilege level.

2.3 Simultaneous Multithreading (SMT)

To improve hardware utilization, manufacturers often employ a technique called Simultaneous Multithreading (SMT) [33], where a single physical core is allowed to execute multiple hardware threads simultaneously. These hardware threads share much of the core's hardware structures, such as functional units, to improve their utilization. However, the fact that these hardware resources are shared between the threads mean that they can interfere with each other, which in turn can be used to create covert/side channels among the threads in the physical core.

2.4 Contention-based Covert/Side Channels

While most existing transient execution attacks rely on stateful covert channels, such as cache based ones (Flush+Reload [42], Prime+Probe [32]), recently researchers have investigated contention based channels among the hardware threads within a single physical core [6, 10, 13]. These contention-based channels exploit the fact that use of the shared hardware resources (ports, functional units) from one thread will affect the performance of the other thread that tries to use the same shared hardware resources. As such, by monitoring the performance variation from one thread, one can infer information about the other thread.

In this work, we show that contention-based channels can be created within a single hardware thread without requiring SMT.

3 THREAT MODEL

We assume an attacker who aims to use transient execution to leak sensitive information from a victim in the same hardware thread. We assume that the attacker has the ability to control some non-privileged code that executes logically before and after a transient execution, which accesses the victim's secret, in program order. We assume that the attacker would like to construct code so that the transient execution transmits the secret over a covert channel. We assume that stateful covert channels, such as cache based channels, are not available to the attacker because the platform either does not provide necessary means to control cache state (e.g., `CLFLUSH`) or implements hardware level defense mechanisms that prevent stateful covert channels [12, 16, 25, 41].

4 SPECTREREWIND

SpectreRewind is an approach to create and utilize contention-based covert channels in transient execution attacks within the same hardware thread. It allows the attacker to both transmit into and receive from a covert channel *before* the transient execution phase of an attack is completed.

In the case of a traditional transient execution attack approach, the attacker will use a covert channel that causes a lasting state change in the micro-architecture, and read from the covert channel from μops that occur logically after the transient execution. Secret data can be read from the channel by measuring the timing differences of these μop. Therefore, hardware defenses (e.g., [16, 41]) that remove the secret from the covert channel after transient execution will be able to stop these attacks by disrupting the transmission of the secret.

In the case of the SpectreRewind approach, however, transient instructions will contend for resources with the μops that come logically before the transient instructions. Because the covert channel will be read from before transient execution completes, the aforementioned hardware defense mechanisms which attempt to remove the secret from the covert channel after transient execution finishes will be ineffective. In our approach, the attacker measures the entire execution time of the attack to detect the timing differences.

SpectreRewind assumes that older transient μops can contend with younger μops that began before the transient μops on certain micro-architectural resources. In the following, we will discuss the kinds of micro-architectural resources that can be used to create covert channels in SpectreRewind.

4.1 Not Fully Pipelined Functional Unit

Since we aim to contend with instructions that are logically older than us, we will not be able to cause port contention or contention on pipelined functional units as in [6] because younger instructions cannot delay the older instructions. However, we find that it is still possible to cause contention on certain functional units that contain at least one *non-pipelined* stage.

Figure 2 shows visual examples of this problem. In Figure 2a, we see an example of an attacker μop trying to cause slowdown on a victim μop that is trying to use a shared integer multiplier. Unfortunately, because both the attacker and victim are ready to issue, the scheduler will choose the older victim, preventing any contention.

Figure 2b shows a situation where the attacker becomes ready the cycle before the victim. The attacker is issued into the multiplier, but still cannot create contention on the victim, as the victim is issued on the next cycle that it becomes ready, just as if the attacker was not there.

Finally, Figure 2c shows an attack on a non-pipelined shared functional unit (stage 1 takes 3 clock cycles to complete). As the victim is not initially ready, the attacker is scheduled on the unit. As the unit is not pipelined, the victim cannot be issued on the unit until the attacker completes, which effects the execution time of the victim, making a covert channel possible. Thus, for our attack we will only focus on functional units that have at least one stage that is not fully pipelined. Note that it is well known that floating point division is difficult to pipeline because for division each step depends on the previous step [24]. In the following, we will develop a floating point division unit based covert channel.

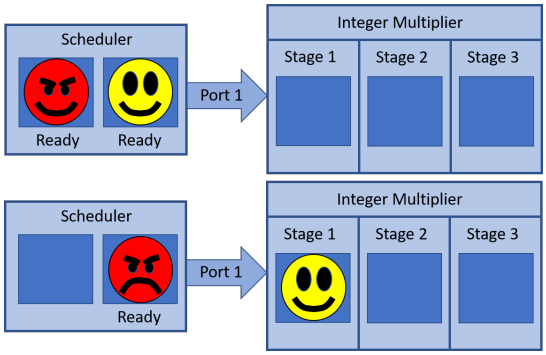
5 FLOATING POINT DIVISION UNIT COVERT CHANNEL

In this section, we utilize our SpectreRewind approach to create a covert channel on real commodity hardware that can transmit data from transient execution without using stateful covert channels, or SMT co-scheduled processes.

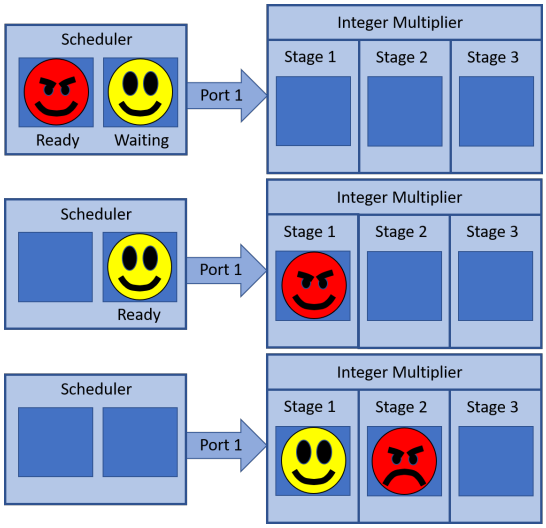
Our covert channel utilizes contention on a non-pipelined functional unit, namely the *floating point division unit* (see Figure 1), to transmit data from transient instructions to non-transient instructions, which will retire and become architecturally visible. The floating point division unit was chosen as it is not fully pipelined in all Intel, AMD, and ARM microarchitectures we tested. Table 1 shows the tested microarchitectures and their latency (Column 4) and throughput (Column 5) characteristics of the `DIVSD` (for x86-64 [2]) [1] , and `FDIV` (for ARM [4, 5]) instructions.

Note that in all tested x86-64 microarchitectures, the throughput of the `DIVSD` instruction is 4 or 8 cycles, meaning that while an `DIVSD` instruction is being executed, a pending `DIVSD` instruction has to wait 4 or 8 cycles before entering the floating point division unit. This delay makes the floating point division unit an ideal candidate for us to create a covert channel.

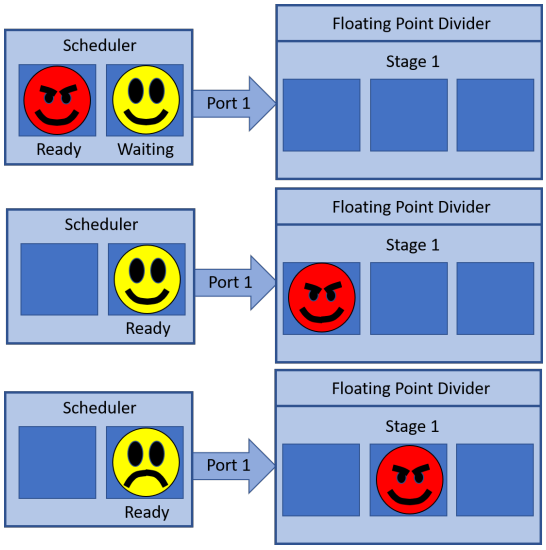
[1] As defined in [2], *latency* refers to the clock cycles needed from the time the μop is issued to the time the result become available to dependent μops, while *throughput* refers to the clock cycles needed from the time the μop is issued until to the time the functional unit becomes available again.



(a) Ready victim, Pipelined functional unit



(b) Waiting victim, Pipelined functional unit



(c) Waiting victim, Not fully pipelined functional unit

Figure 2: Multiple attempts by attacker to delay the execution of the victim, causing measurable timing differences. If the attacker is younger than the victim, an age-ordered scheduler will prevent most contention.

```
1   double recv, div;
2   double send1, send2, send3, send4;
3   int message; // secret
4
5   start = rdtscp(); // start timer
6
7   // begin receiver (12 dependent FP divisions)
8   recv /= div;
9   recv /= div;
10  ...
11  recv /= div;
12  // end of receiver
13
14  if (recv == 1) { // begin speculative execution
15      m_bit = bit(message, k); // access secret
16      if (m_bit) { // secret dependent branch
17          // begin sender (independent FP divisions)
18          for (int x = 0; x < 100; x++) {
19              send1 /= div;
20              send2 /= div;
21              send3 /= div;
22              send4 /= div;
23          }
24          // end of sender
25      }
26  }
27
28  end = rdtscp(); // end timer
```

Figure 3: Pseudo code of our floating point division unit contention based covert channel in a Spectre like transient execution attack.

Figure 3 shows the code used to form the ideal covert channel. (1) A timer is started (Line 5); (2) A chain of *dependent* floating point division instructions begins execution (Line 8). Because the instructions are dependent, each instruction suffers the full round-trip latency of the floating point division unit (see Table 1). This chain of division instructions acts as a *receiver*; (3) The result of the receiver instruction chain is compared in the if statement (Line 14). Note that we train the if statement to be true so that the body will execute speculatively while the result of the receiver chain is being calculated; (4) A single bit of the (secret) message to transmit is accessed (Line 15) and the inner if statement branches depending on the value of the secret bit (Line 16); (5) The inner if statement is trained to be false. Thus, if the secret bit was '1', the processor backtracks and begins to speculatively execute a set of *independent* floating point division instructions (Line 18-23), which act as a *sender*. The "sender" instructions are independent with each other so as to be issued concurrently and maximally contend with the "receiver" instructions on the floating point division unit of the processor. (6) When the "receiver" instructions are completed, the processor will realize the mis-speculation (*recv* in Line 14 was 0) and squash the speculative instructions from the "sender". We then stop the timer (Line 28) and measure the time difference.

Note that if the secret bit was '1', the observed time difference will be longer, due to the contention in the floating point division unit with the mis-speculated "sender" instructions, compared to

the case when the secret bit was '0' where there was no contention. This secret-dependent timing difference creates a covert channel.

5.1 Covert Channel Properties

We experimentally evaluate the characteristics the covert channel on a number of commodity Intel, AMD, and ARM systems, as listed in Table 1.

Each system runs Linux (Ubuntu 18.04 or 16.04). For x86 platforms from Intel and AMD, we use `rdtscp` instructions for cycle accurate timing measurements. For ARM, we use an additional thread based software counter instead due to the architectural limitation. We repeatedly send 0 and 1 values over the covert channel, each for 1,000,000 times, and measure the timing results. To minimize noise, we use Linux's performance governor disable Turbo-boost (for X86 platforms) to improve reliability of the measurements.

Figure 4 shows the results. The X-axis shows the number of cycles taken to transmit, while the Y axis displays the probability a measurement has to take that many cycles. Note first that on all tested platforms, we see clear timing differences between '0' and '1' values. As explained in Section 4.1, not fully pipelined floating point division units in these platforms allow the mis-speculated division instructions to contend with the logically prior "receiver" instructions, resulting in clearly measurable timing differences.

Another interesting observation is that the two AMD processors and the ARM Cortex-A57 show discreet timing characteristics—large proportion of the samples are concentrated on a few small measured cycles—whereas Intel processors show more varied timing behaviors, especially the Skylake processors. These differences are likely due to the way the floating point division unit is implemented in each of these vendors.

In addition to `DIVSD`, we also evaluated other instructions that utilize the same floating point division unit to determine if they could be used for creating covert channels as well. To this end, we evaluated division and square root instructions from the AVX (`VDIVSD`, `VDIVSS`, `VSQRTSD`, `VSQRTSS`), SSE (`SQRTSS`, `DIVSS`), and SSE2 (`SQRTSD`) instructions on both the Intel i5-6500 and AMD Ryzen 5 2600 machines, and found that they all can be used to create covert channels. Finally, we also evaluated floating point multiplication instructions but were not able to observe any noticeable timing difference, suggesting that the floating point multiplication units in these platforms are well pipelined, and thus cannot be used to create covert channels.

5.2 Performance Analysis

Next, we analyze the performance of the covert channel in terms of transfer rate and error rate. The measured transfer rates of our tested platforms are calculated by simply dividing the total bits sent (1 million bits of 0 and 1 million bits of 1) with the time it took to send them. The error rate of each system is calculated as follows. We first sort each million timing samples of 0 and 1. We then find 99 percentile value of the '0' samples and 1 percentile value of the '0' samples. If the former (99 percentile of '0' samples) is smaller than the latter (1 percentile of '1' samples), we pick the average of the two value as the threshold to determine 0 or 1. If the 99 percentile of 0 is bigger than the 1 percentile of 1, we set the average of the median values of 0 and 1 samples as the threshold value. We then apply the threshold against the collected samples to determine if it correctly classifies the sample against its known correct value.

The results are shown in Table 1 (see the 'Transfer Rate' and 'Error Rate' columns). First, notice that the proposed covert channel supports very high transfer rates on all tested platforms, ranging from 63 to 105 KB/s. Furthermore, the error rates are also very low, especially on Intel processors, as we observe less than 0.5% error rates. AMD processors show higher error rates, of up to 5.5% on low end Ryzen3 APU.

5.3 Sensitivity Analysis

An interesting aspect of our covert channel is that the size (duration) of the speculation window can be controlled by adjusting the number of dependent division instructions used in the "receiver" part of the covert channel—i.e., Line 8-11 in Figure 3. This is because speculatively executed sender instructions are squashed after the receiver instruction change is completed. As such, the longer the receiver instruction chain is, the longer the sender instructions can contend on the floating point division unit. To understand the effect of the length of the receiver to the effectiveness of the covert channel, we measure the characteristics of the covert channel as a function of the number divisions in the receiver chain.

Table 2 shows the results. The first column shows the number of division instructions in the receiver chain. The second and third columns show the median cycles observed when sending '0' and '1' values over the covert channel, respectively. The fourth column is the cycle difference between 0 and 1 samples. Finally, the fifth and the last columns show the transfer and error rates of the channel.

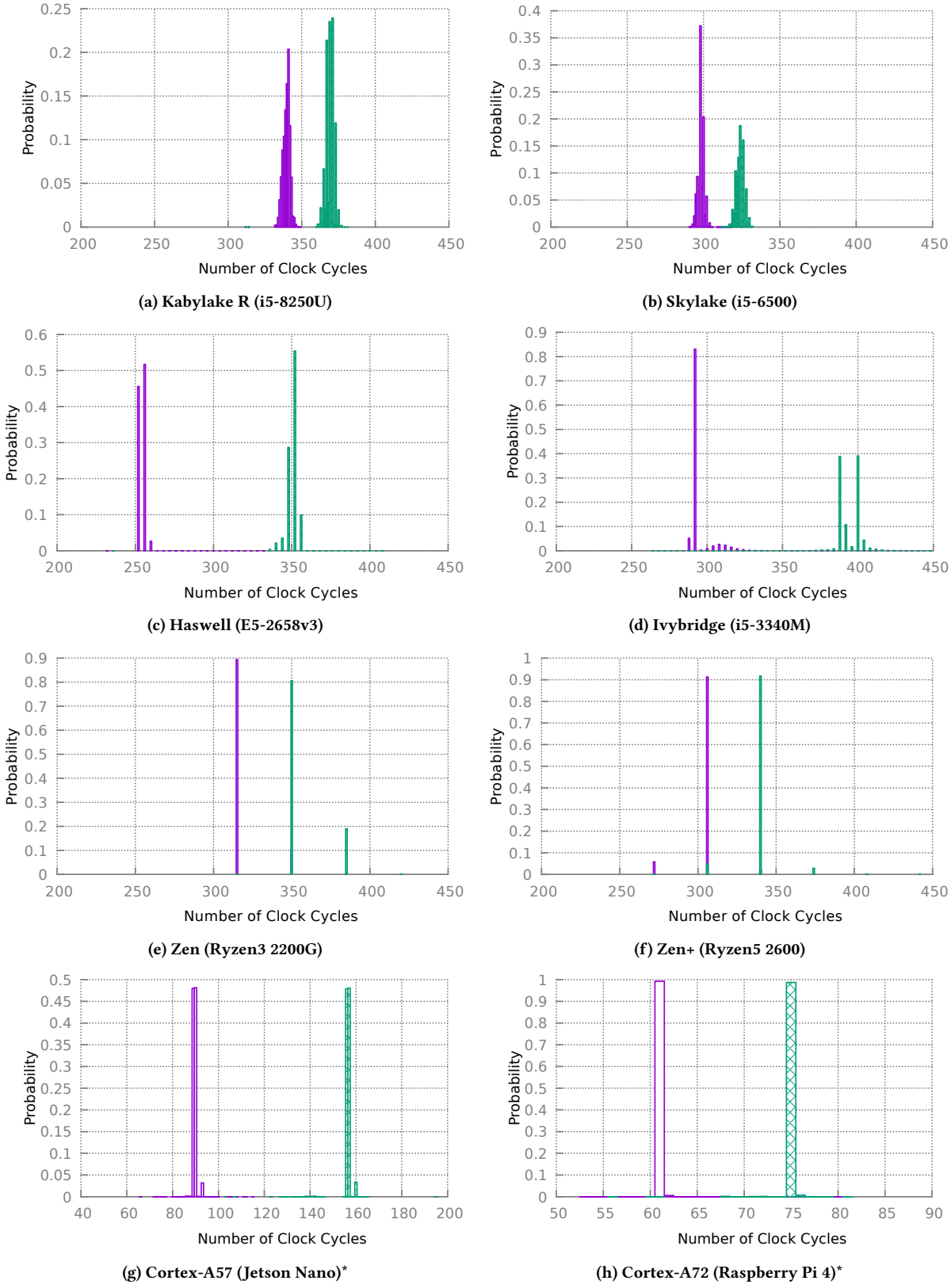
Note first that the transfer rate is inversely proportional to the number of divisions in the receiver, which is expected as the more divisions are used, the longer time is needed to execute them before squashing the speculation. As such, from the transfer rate perspective, using a smaller number of divisions in the receiver may be desirable. However, when the number of divisions is too small, as in the case of 3 divisions, the covert channel becomes ineffective as the error rate is too high. This is because the speculation window is not long enough for the sender instructions to be able to effectively contend with the receiver instructions on the floating point division unit.

The error rate dramatically decreases as we increase the number of divisions in the receiver. At 9 or more divisions, the covert channel shows very low error rate while showing gradually decreasing transfer rates. For this platform, we can see using 12 divisions in the receiver chain is a "sweet spot" in the sense that it offers high enough performance and low noise. While different platforms may have different sweet spots, we nevertheless used the same 12 divisions in all platforms, unless noted otherwise, as it performed reasonably well in all of them.

6 SPECTREREWIND IN JAVASCRIPT

In this section, we show that SpecreRewind attack can work in a JavaScript sandbox environment.

Similarly to the original Spectre attack PoC in JavaScript [18], we developed a PoC that implements our floating point division unit covert channel in JavaScript, and successfully executing it on Google Chrome version 62.0.3202.75, which allows a website



(a) Kabylake R (i5-8250U)

(b) Skylake (i5-6500)

(c) Haswell (E5-2658v3)

(d) Ivybridge (i5-3340M)

(e) Zen (Ryzen3 2200G)

(f) Zen+ (Ryzen5 2600)

(g) Cortex-A57 (Jetson Nano)*

(h) Cortex-A72 (Raspberry Pi 4)*

Figure 4: Floating point division unit based covert channel (Figure 3) timing characteristics; 1,000,000 timing measurement samples of transmitting 0 (purple) and 1 (green). (*) For ARM platforms, we use an additional thread based software counter for time measurement due to the lack of high-precision clock source (such as `rdtsc` in x86) available at the user level.

CPU	ISA	Microarch.	Latency (cycles)	Throughput (cycles)	Transfer Rate (KB/s)	Error Rate (%)
Intel Core i5-8250U	x86-64	Kabylake R	13–15	4	53.1	0.02
Intel Core i5-6500	x86-64	Skylake	13–15	4	115.1	0.01
Intel Xeon E5-2658 v3	x86-64	Haswell	10–20	8	64.1	0.01
Intel Core i5-3340M	x86-64	Ivybridge	10–20	8	75.6	0.16
AMD Ryzen 3 2200G	x86-64	Zen	8–13	4	83.1	5.50
AMD Ryzen 5 2600	x86-64	Zen+	8–13	4	84.8	3.30
NVIDIA Jetson Nano	ARMv8	Cortex A57	7-32	5-30	87.7	0.02
Raspberry Pi 4	ARMv8	Cortex A72	6-18	4-16	80.7	0.16

Table 1: Evaluation platforms; latency and throughput for `DIVSD` (for x86-64 [2]) and `FDIV` (for ARM [4, 5]) instructions; measured performance (transfer and error rates) of each platform's floating point division unit covert channel.

#divs	'0' (cycles)	'1' (cycles)	Diff. (cycles)	Transfer (KB/s)	Error (%)
3	169	169	0	155.0	49.98
6	204	212	8	140.6	0.62
9	242	258	16	126.2	0.03
12	276	299	23	115.1	0.01
15	312	345	33	105.0	0.01
24	418	472	54	84.7	0.01
48	705	814	109	55.5	0.01
72	991	1107	116	41.3	0.01

Table 2: Sensitivity to #of divisions (`DIVSD`) used in the "receiver" part of the covert channel on Intel i5-6500.

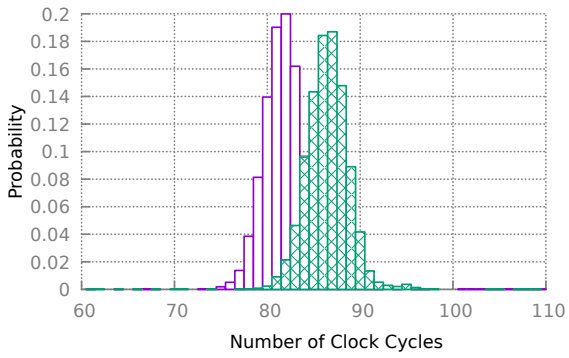


Figure 5: Timing characteristics of division floating point unit covert channel execution in Google Chrome JavaScript sandbox

to read private memory from the process in which it runs. For a high resolution timer, as in [18], we also followed the approach described by Schwarz et al. [28], which utilize Web Workers along with `SharedArrayBuffer`. This allows for the creation of a separate thread, that continuously increments a value in shared memory that the original thread can use to time code execution. The main difference in our PoC is that we do not rely on any cache state manipulation techniques unlike Kocher et al. [18].

Figure 6 shows a snippet of the final code along with the generated assembly, produced by the JavaScript JIT compiler, of the JavaScript version side-by-side with the natively compiled C version's assembly code. While the number of instructions the JavaScript version is bigger than that of the natively compiled version, we find that the majority of these extra instructions happen in the section of code that is responsible for accessing the message and branching on bit values. Moreover, the all important division operations are compiled neatly down to a few floating point division instructions in both versions. We find that the resolution of the SharedArrayBuffer based timer is, though not as good as the native timers, sufficient for data transmission. We have however increased the number of receiver code divisions from 12 to 24 to improve signal over the lower resolution timer. Figure 5 shows the probability distribution of the transmission of the JavaScript based covert channel, which show distinguishable timing differences depending on the value of the secret bit it accesses during the transient execution.

Note that our current JavaScript PoC may not work in recent Chrome browsers which implement Spectre prevention mechanisms because they also appear to block speculative execution of the secret dependent division instructions needed by SpectreRewind. Circumventing the Spectre defense mechanisms in recent versions of JavaScript sandbox environments is future work.

7 DISCUSSION

In this section, we discuss the benefits and shortcomings of SpectreRewind, and its mitigation options.

7.1 Benefits and Limitations

SpectreRewind utilizes a new type of contention-based covert channel, which is available in a wide range of micro-architectures while providing high bandwidth and low noise characteristics. As such, we believe that our covert channel can be used as an alternative covert channel to cache-based ones for transient execution attacks. Our covert channel may be preferable to Flush+Reload in environments where instructions to flush cache lines (e.g. `CLFLUSH` in x86) are not available (e.g., most ARM platforms, browser sandboxes). We are currently developing a set of PoCs that demonstrate the potential use of the new covert channels in a subset of Spectre, Meltdown, and MDS attacks, which are mounted from the same hardware thread. For example, in our preliminary experiment, we

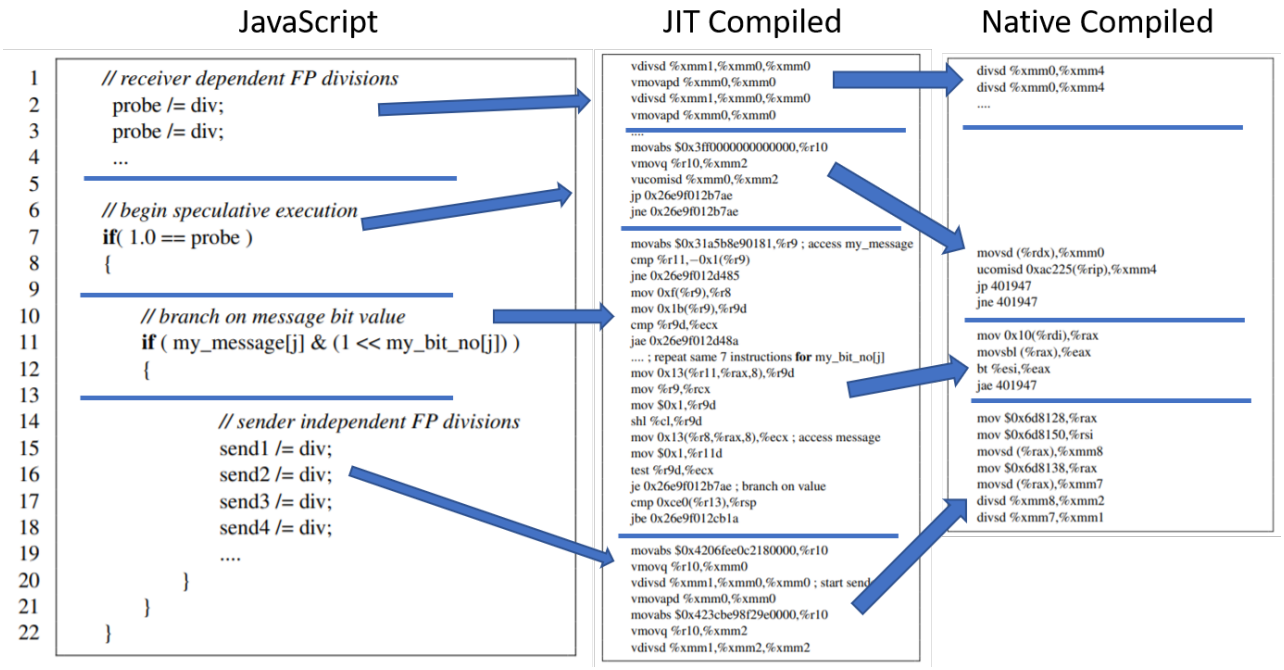


Figure 6: Excerpt from JavaScript covert channel code (Left), the assembly the JIT compiler created (Center), and the native generated assembly (Right)

were able to modify a publicly available Meltdown PoC to utilize SpectreRewind's covert channel.

One major downside of SpectreRewind is that it requires sender and receiver instructions be present simultaneously at the same hardware thread, which restricts its use in cross-process/core attack scenarios (e.g., [18]). Also, finding an exploitable gadget, which includes secret dependent division instructions, may be challenging to find in real application binaries. In addition, the sender and receiver instructions must be executed from the same protection domain—either both in kernel or both in user. This is because the CPU privilege mode change involves pipeline flush. Therefore, initiating the receiver instructions at the user-level while executing the sender instructions at the kernel (e.g., a system call) may not be feasible. Note, however, that speculative access to a memory location in a different protection domain (e.g., access to a kernel address in Meltdown-type attacks) is still possible because the involved instructions are still executed at the same protection domain.

7.2 Mitigation Strategies

As SpectreRewind requires out-of-order contention on not fully pipelined functional units in the processor, one mitigation strategy is to redesign the functional units to be fully pipelined. But such a redesign may not always be possible. Another alternative is to adopt a strict in-order scheduling policy such that younger instructions (sender) can never be issued before all older instructions are issued first, though it would incur high performance cost.

An effective mitigation strategy is to delay or prevent the execution of secret dependent instructions during the transient execution phase. SpectreGuard [11] is an example of such an approach, where secret data is marked as secret in the application process's page table and then is disallowed from being forwarded to dependent instructions until it reaches a point where it can be logically considered safe to forward. ConTExT [26] uses a similar approach, marking data as secret in the page table and delaying propagation of the value of the secret. Intel and NVIDIA also proposed similar mitigation solutions [7, 30]. NDA [39] and STT [44] are software transparent hardware solutions that selectively allow some (safe) instructions to be executed speculatively while preventing other (unsafe) instructions. All these techniques that prevent secret dependent speculative execution may mitigate SpectreRewind covert channels.

8 RELATED WORK

Most known transient execution attacks utilize stateful cache-based covert channels, which exploit the timing differences in accessing cached (hit) and non-cached (miss) memory addresses. Cache-based covert channels are powerful because secret dependent state changes in a cache can be long lasting (persistent), making secret recovery relatively easier for an attacker. Also, they generally offer high bandwidth and low noise compared to other covert channels in modern processors. For these reasons, there have been a flurry of research proposals to protect specifically against cache based covert channels [12, 16, 25, 41] as a mean to defend against transient execution attacks. For example, InvisiSpec [41] and SafeSpec [16] are both recently proposed hardware solutions that defer updating microarchitectural states of caches (and TLBs) until such changes are considered to be safe. Gonzalez et al [12] implemented

such a defense on an actual out-of-order open source RISC-V processor core. CleanupSpec [25] lets the microarchitectural changes from transient instructions occur but later undo those changes after recognizing mis-speculation. In contrast, SpectreRewind exploits a contention-based covert channel and thus bypasses all these defense mechanisms against stateful cache covert channels.

Contention-based covert channels are well studied in the context of Simultaneous multi-threading (SMT) processors. Wang and Lee showed various ways to create covert/side channels in SMT processors [38] and discussed possible mitigations. Acıiçmez and Seifert used the contention on the shared integer multiplication unit as a side channel [3] to break a cryptographic function in OpenSSL running concurrently on a separate hardware thread on the same core. CacheBleed [43] exploited L1 cache bank contention as a covert channel while MemJam [23] instead utilized false read-after-write dependencies to create a covert channel. Both CacheBleed and MemJam applied their respective covert channels to break constant time OpenSSL implementations. Covert Shotgun [10] systematically explored possible contention-based covert channels by exhaustively executing instructions on different SMT threads of the same physical core. PortSmash [8] utilized port contention to create a microarchitectural side-channel to leak the secret key from a vulnerable version of OpenSSL. SmotherSpectre [6] utilized a port contention based side channel to mount a transient execution attack, specifically the Branch Target Injection attack (BTI, a.k.a., Spectre variant 2 [18]). Using BTI allowed this attack to run attacker code to transiently access secret in the victim and then to execute secret dependent instructions, which can be monitored by the attacker's process on a different SMT thread of the same core. ABSynth [13] goes a step further by automatically discovering the best set resources, not just execution ports in most prior works, that can leak information with a blackbox analysis. SMT-cop [31] prevent these SMT based covert channels by providing spatial and temporal partitioning of the SMT resources. SMT based covert channels can also be prevented by simply disabling SMT. Our work differs from these prior works as we focus on contention based covert channels in the non-SMT context, specifically from the single hardware thread context, and in the context of transient execution attacks.

9 CONCLUSION AND FUTURE WORK

In this paper, we presented SpectreRewind, a new approach to create and exploit contention-based covert channels in transient execution attacks from a single hardware thread. We identified that speculatively executed young instructions can delay logically older non-speculative (bound-to-retire) instructions due to contention on non-pipelined functional units of modern out-of-order processors. Specifically, we showed that contention on non-pipelined floating point division units in commodity Intel, AMD, and ARM processors can create high-bandwidth, low-noise covert channels in same thread transient execution attacks. We also showed that the covert channel can be used in the JavaScript sandbox of a Chrome browser. As future work, we plan to develop end-to-end transient execution attacks leveraging the covert channel. Also, we will further investigate if other microarchitectural structures can be used to create contention based covert channels in transient execution attacks.

ACKNOWLEDGEMENTS

This research is supported in part by NSF grant CNS 1718880 and NSA Science of Security initiative contract no. H98230-18-D-0009.

REFERENCES

[1] 2018. Cache Speculation Side-channels. *ARM White paper* (2018).
[2] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 673–686.
[3] Onur Aciicmez and Jean-Pierre Seifert. 2007. Cheap Hardware Parallelism Implies Cheap Security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 80–91.
[4] ARM. 2015. Cortex-A72 Software Optimization Guide. https://static.docs.arm.com/uan0016/a/cortex_a72_software_optimization_guide_external.pdf. (2015).
[5] ARM. 2016. Cortex-A57 Software Optimization Guide. https://static.docs.arm.com/uan0015/b/Cortex_A57_Software_Optimization_Guide_external.pdf. (2016).
[6] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 785–800.
[7] Darrell D Boggs, Ross Segelken, Mike Cornaby, Nick Fortino, Shailender Chaudhry, Denis Khartikov, Alok Mooley, Nathan Tuck, and Gordon Vreugdenhil. 2019. Memory type which is cacheable yet inaccessible by speculative instructions. (Jan. 3 2019). US Patent App. 16/022,274.
[8] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *IEEE Symposium on Security and Privacy (SP)*.
[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*.
[10] Anders Fogh. 2016. https://cyber.wtf/2016/09/27/covertshotgun/. (2016).
[11] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. In *Design Automation Conference (DAC)*. 61–1.
[12] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanović. 2019. Replicating and Mitigating Spectre Attacks on an Open Source RISC-V Microarchitecture. In *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*.
[13] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *Network and Distributed Systems Security (NDSS)*.
[14] Jann Horn. 2018. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528. (2018).
[15] Intel. 2018. *Intel Analysis of Speculative Execution Side Channels (Rev. 4.0)*. Technical Report. https://software.intel.com/sites/default/files/managed/b9/f9/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf
[16] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *Design Automation Conference (DAC)*.
[17] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).
[18] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, MMike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society.
[19] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*.
[20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*.
[21] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2109–2122.
[22] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Berk Sunar, Frank Piessens, and Yuval Yarom. 2019. Fallout: Reading Kernel Writes From User Space.
[23] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming* (2019).

[24] Stuart F Oberman. 1999. Floating point division and square root algorithms and implementation in the AMD-K7/sup TM/microprocessor. In *IEEE Symposium on Computer Arithmetic (Cat. No. 99CB36336)*. IEEE, 106–115.

[25] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *International Symposium on Microarchitecture (MICRO)*. ACM, 73–86.

[26] Michael Schwarz, Moritz Lipp, Claudio Alberto Canella, Robert Schilling, Florian Kargl, and Daniel Gruß. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *Network and Distributed System Security (NDSS)*.

[27] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM Conference on Computer and Communications Security (CCS)*.

[28] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security*, Aggelos Kiayias (Ed.). Springer International Publishing, Cham, 247–267.

[29] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv preprint arXiv:1806.07480* (2018).

[30] K Sun, R Branco, and K Hu. 2019. A New Memory Type Against Speculative Side Channel Attacks. (2019).

[31] Daniel Townley and Dmitry Ponomarev. 2019. SMT-COP: Defeating Side-Channel Attacks on Execution Units in SMT Processors. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[32] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology* 23 (07 2010), 37–71.

[33] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous Multithreading: Maximizing On-chip Parallelism. In *International Symposium on Computer Architecture (ISCA)*. ACM, 392–403.

[34] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*. USENIX Association.

[35] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[36] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[37] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. https://cacheoutattack.com/. (2020).

[38] Z. Wang and R. B. Lee. 2006. Covert and Side Channels Due to Processor Architecture. In *Annual Computer Security Applications Conference (ACSAC)*. 473–482.

[39] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. 2019. NDA: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 572–586.

[40] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018).

[41] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *International Symposium on Microarchitecture (MICRO)*.

[42] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732.

[43] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* (2017).

[44] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. 2019. Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data. In *International Symposium on Microarchitecture (MICRO)*. 954–968.