Noname manuscript No.
(will be inserted by the editor)

A Framework for Leaking Secrets to Past Instructions

Jacob Fustos · Michael Bechtel · Heechul Yun

Received: date / Accepted: date

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 works on a single hardware thread and does not require SMT. We show that contention on the floating point division unit on commodity out-of-order processors can be used to create a high-performance ($\sim$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 and can be used in a Meltdown attack.

Keywords Spectre · Micro-architectural Attack · Covert-channel

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 [20]—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 [20], Meltdown [22] and many other subsequently discovered transient execution attacks [19,23,21,16,1,17,33,38,42,28,24,31,10,29, 39] 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 [18,43,14] or by reverting them when the transient instructions are squashed [27]. 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.

Jacob Fustos · Michael Bechtel · Heechul Yun
University of Kansas

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 [7] is the first to demonstrate such in the context of a Spectre-based attack. By generating contention on issue ports within simultaneous multi-threading (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 is a contention based covert channel and allows the attacker to both transmit and receive secret data before transient execution has completed, thus bypassing most defense mechanisms that attempt to revert or hide micro-architectural changes caused by the attack. Unlike SmotherSpectre, however, SpectreRewind is executed from a single hardware thread and does not require SMT. 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 *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, thus evading stateful defenses. However, the fact that the sender and the receiver must execute on the same hardware thread also limits its use in cross-core attack scenarios.

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 ($\sim$100KB/s), low-noise ($<$0.01%) covert channels that are comparable to commonly used cache-based covert channels. We also show the feasibility of our covert channel within a Chrome browser's JavaScript sandbox. Lastly, we demonstrate a complete cross-domain Meltdown attack, which is modified to use a SpectreRewind covert channel.

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

- We find that execution of speculative instructions can cause secret dependent delay to *logically prior* non-speculative (bound-to-retire) instructions due to resource contention between them and that can be exploited to create a new class of covert channels. To the best of our knowledge, we are the first to make the discovery.
- We present a covert channel that exploits contention on *non-pipelined functional unit*, namely a floating point division unit, between the speculative and non-speculative instructions [1]. We experimentally evaluate the performance of the covert channel on a number of commodity out-of-order processors from Intel, AMD, and ARM.
- We further demonstrate the feasibility of creating the covert channel within JavaScript sandbox, and successful application of the covert channel in a cross-domain transient execution attack, namely the Meltdown attack.

Compared to our prior workshop version [12], this journal extension further investigates the feasibility and effectiveness of the proposed approach by (1) exploring other micro-architectural resources that can also be used to leak secret to earlier instructions; (2) experimentally validating the existence of the SpectreRewind covert channel on more recent hardware architectures, (3) improving performance of the channel on AMD processors, (4) presenting a complete end-to-end Meltdown attack using our covert channel, and (5) comparing our work with recent related work.

The remainder of the paper is organized as follows. Section 2 describes the necessary background. Section 3 defines the threat model. Section 4 introduces SpectreRewind and Section 5 presents a concrete SpectreRewind covert channel, which exploit contention in the floating-point division unit in a out-of-order processor pipeline. Section 6 demonstrates the feasibility of a SpectreRewind channel in a JavaScript sandbox and Section 7 shows its application in a cross-domain attack. We discuss limitations of our approach in Section 8. We review related work in Section 9 and conclude in Section 10.

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

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

[1] Our PoC code is available at `http://github.com/CSL-KU/SpectreRewind-POC`

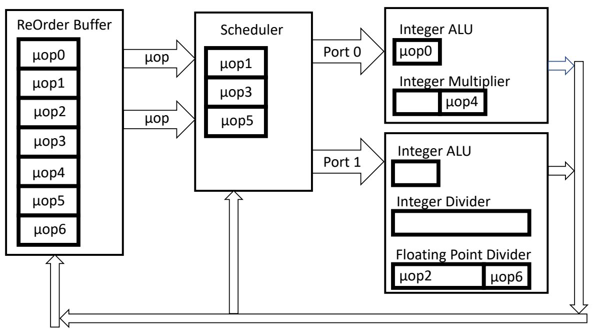


Fig. 1: Simplified out-of-order processor design. The ReOrder Buffer holds and retires µops in logical program order, while µops are issued to the execution units in out-of-order.

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 misspeculation 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 [20,19,20,16,23,21] 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 [19,22,1,17,33,38,42] corresponds to the exception that caused the fault. Micro-architectural Data Sampling (MDS) [24,28,31] 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) [37], 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 [44], Prime+Probe [36]), recently researchers have investigated contention based channels among the hardware threads within a single physical core [11,7,15]. 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 [14, 18, 43, 27].

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.

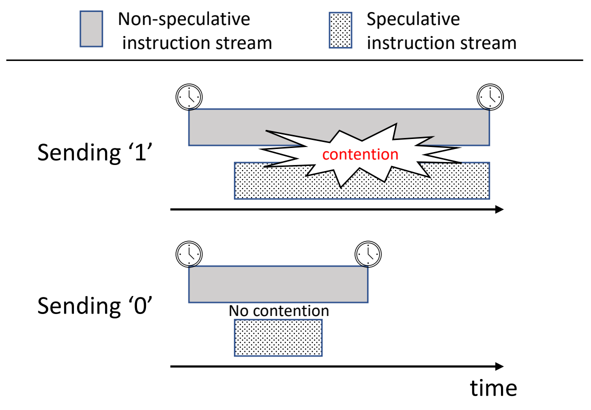

Non-speculative instruction stream
Speculative instruction stream
Sending '1'
contention
Sending '0'
No contention
time

Fig. 2: The SpectreRewind approach: exploit contention between secret-dependent speculative instructions and non-speculative instructions to create a covert channel.

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., [43, 18]) 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. Figure 2 illustrates the basic concept of the SpectreRewind approach.

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.

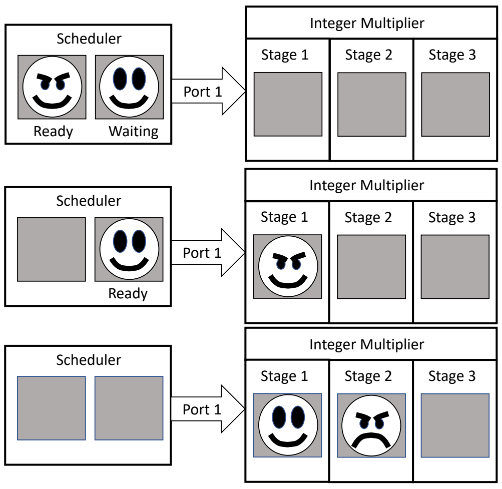
4.1 Non-Pipelined Functional Unit

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 [7] 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.

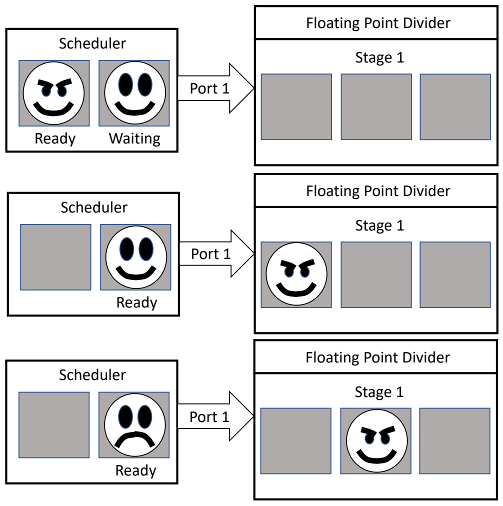
Figure 3a shows a situation where the attacker becomes ready the cycle before the victim on a pipelined functional unit. The attacker is issued into the multiplier, but still cannot create contention on the victim, as the victim is issued on the next cycle that it becomes ready, just as if the attacker was not there.

Figure 3b, on the other hand, shows the same situation but on a non-pipelined functional unit in which the first stage 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 [26]. In Section 5, we will develop a floating point division unit based covert channel.



(a) Waiting victim, Pipelined functional unit



(b) Waiting victim, Not fully pipelined functional unit

Fig. 3: 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.

4.2 Other Micro-architectural Resources

While we mainly focus on non-pipelined functional unit in this work, the SpectreRewind approach can be applied to other micro-architectural resources.

Recently, Behnia et al., also observe that mis-speculated young instructions can delay non-speculative old instructions in out-of-order processors and showed miss-status-holding-registers (MSHRs) and reservation station (RS) also can be used to create covert-channels [6]. Note, however, that any micro-architectural resource that can impact the execution of older instructions can be used to create covert channels. Furthermore, it is even possible that younger, secret dependent speculative instructions can make older non-speculative instructions run *faster*.

Consider, for example, a set of non-speculative and data dependent instructions that perform linked-list traversal, which may miss the cache at every list item access. Due to the inherent data dependency, the hardware prefetcher might not be able to prefetch necessary data in the list on its own. In this situation, if secret dependent speculative instructions are executed concurrently, they can act as a "prefetcher" for the data that will be needed by the non-speculative instructions, thereby reducing their execution time. This "prefetching" by speculative instructions can be performed efficiently because the speculative instructions can issue multiple memory requests simultaneously, while the non-speculative ones, the linked-list traversal, can only generates one memory access at a time. This creates a covert-channel because the attacker can measure the execution time of the linked-list traversal, which will take longer if the secret was zero (no "prefetching"), or shorter if the secret was one (due to "prefetching"). In other words, the secret is transferred to the receiver in the form of execution time difference without leaving any trace. As such, we would like to stress that it is not necessary to delay the older instructions to create a covert channel, but any measurable execution timing changes—be it *faster* or *slower*—by secret-dependent speculative instructions can be used to create a 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]) [2] , 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

[2] 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.

```
1   double recv, div;
2   double send1, send2, send3, send4;
3   int message; // secret
4
5   start = rdtscp(); // start timer
6
7   // begin receiver (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
```

Fig. 4: Pseudo code of our floating point division unit contention based covert channel in a Spectre like transient execution attack.

the floating point division unit an ideal candidate for us to create a covert channel.

Figure 4 shows the code used to form the 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 of 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 to disable Turbo-boost (for X86 platforms) and improve the reliability of the measurements.

Figure 5 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 discrete timing characteristics—a 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

CPU	ISA	Microarch.	Latency (cycles)	Throughput (cycles)	Transfer Rate (KB/s)	Error Rate (%)
Intel Core i7-1185GRE	x86-64	Tiger Lake	13–15	4	113.7	0.02
Intel Core i7-1065G7E	x86-64	Ice Lake	13–15	4	102.2	0.23
Intel Core i5-8250U	x86-64	Kaby Lake R	13–15	4	53.1	0.02
Intel Core i5-6500	x86-64	Skylake	13–15	4	115.1	0.01
Intel Core i5-6200U	x86-64	Skylake	13–15	4	74.9	0.04
Intel Xeon E5-2658 v3	x86-64	Haswell	10–20	8	64.1	0.01
Intel Core i7-9530K	x86-64	Haswell-E	10–20	8	103.0	0.00
Intel Core i5-3340M	x86-64	Ivybridge	10–20	8	75.6	0.16
AMD Ryzen 3 2200G	x86-64	Zen	8–13	4	130.2	0.13
AMD Ryzen 5 2600	x86-64	Zen+	8–13	4	131.8	0.19
NVIDIA Jetson Nano	ARMv8	Cortex A57	7-32	5-30	126.0	0.02
Raspberry Pi 4	ARMv8	Cortex A72	6-18	4-16	98.1	0.01

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.

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.

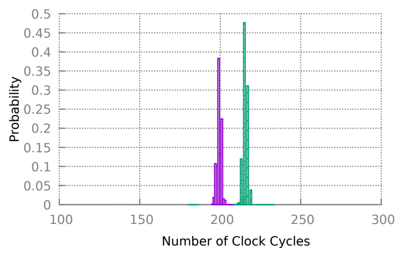
5.2 Performance Analysis

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.

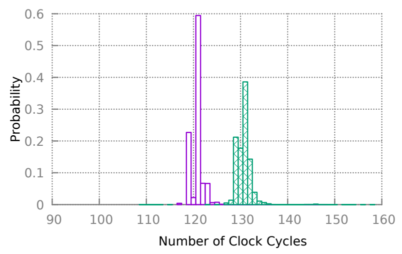
The results are shown in Table 1 (see the 'Transfer Rate' and 'Error Rate' columns). First, notice that the proposed covert channel supports very high transfer rates on all tested platforms, ranging from 53 to 130 KB/s. One of the major factors that affect the transfer rate is the branch predictor of the processing core. Note that, to send a single bit, either 0 or 1, we need to *mistrain* the core's branch predictor so that the desired secret dependent speculative instructions are executed. The number of needed training runs varies depending on the architecture and it impacts the transfer rates. On Tiger Lake and Ice Lake, which are two more recent architectures from Intel, at least nine training runs were needed while seven runs were sufficient on all other architectures we tested. As such, the transfer rates of Tiger Lake and Ice Lake are about 20% slower than they would be had they needed seven training runs as in other architectures. On the other hand, the Jetson Nano platform's ARM Cortex-A57 core needs only two training runs to mistrain its branch predictor, which is the smallest among the all platforms we have tested. As a result, despite being the slowest architecture in terms of raw processing power, the Cortex-A57 achieves one of the highest transfer rates (over 120KB/s). AMD processors also require only four training runs to mistrain their branch predictors, which results in relatively higher transfer rates.

Second, the measured error rates are very low on all tested platforms across multiple generations of Intel, AMD, and ARM processors. In our earlier work [12], we reported that AMD processors suffer relatively high error rates (>5%) than Intel's. However, we found that this was because AMD's floating point units have special performance optimization for certain division operands (namely divide by 1, which we originally used) that significantly alter the execution timing of our dependent division instruction chain irrespective of contention from the speculative instructions. After carefully choosing the division operand such that the timing is only affected by the secret dependent execution of of speculative instructions, the error rates improved on par with what we observe on other architectures.

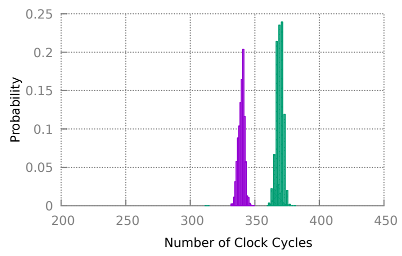
In addition, it is worth mentioning that we observe small but consistently higher error rates when sending 0 than sending 1. In other words, more zeros are falsely recognized as ones than the opposite although both cases are rare in most platforms. For example, we observe 49 errors out of 1,000,000 samples of sending zero while observe 2,142 errors out of the same one million samples of sending one. We believe that this is because, in our covert channel, speculative execution of the sender instructions (sender) reliably increase the execution time of the timed non-speculative instructions (receiver) in the pipeline due to increased contention while not executing the speculative instructions (i.e., sending zero) does not guarantee the absence of contention



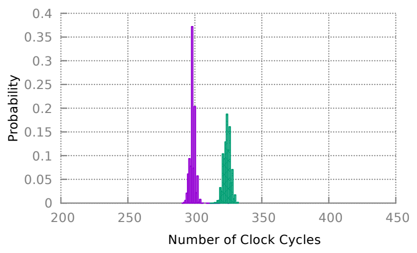
(a) TigerLake (i7-1185GRE)



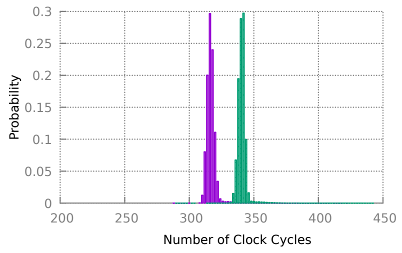
(b) Ice Lake (i7-1065G7)



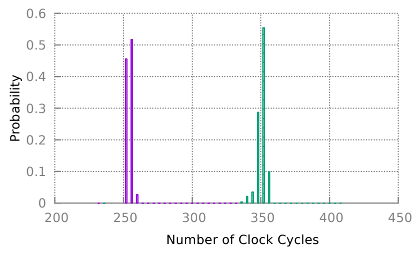
(c) Kabylake R (i5-8250U)



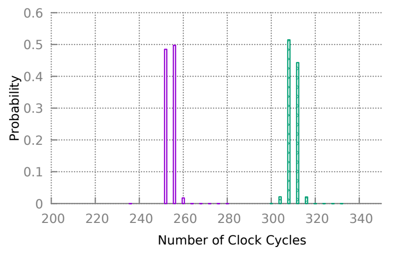
(d) Skylake (i5-6500)



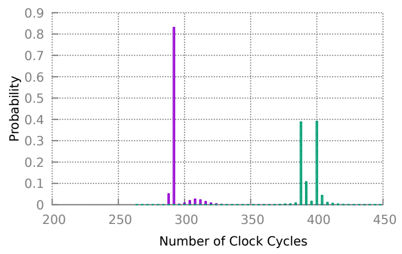
(e) Skylake (i5-6200U)



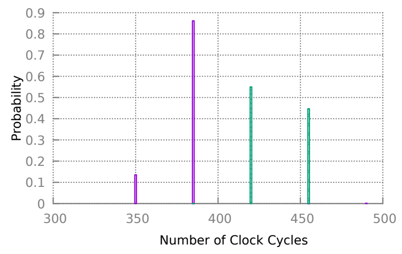
(f) Haswell (E5-2658v3)



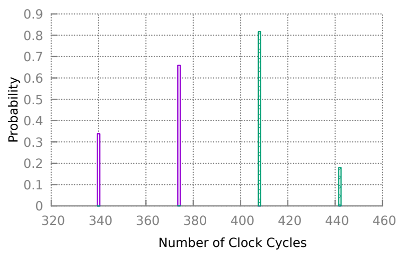
(g) Haswell-E (i7-9530K)



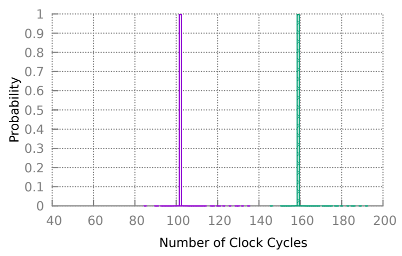
(h) Ivybridge (i5-3340M)



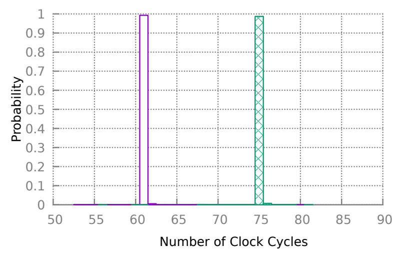
(i) Zen (Ryzen3 2200G)



(j) Zen+ (Ryzen5 2600)



(k) Cortex-A57 (Jetson Nano)*



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

Fig. 5: Floating point division unit based covert channel (Figure 4) 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.

and the receiver can be delayed by other sources. This slight bias toward one in our covert channel is different from the cache covert channel's behavior reported in [22], which shows a significant bias toward zero, that required many retries to filter out ”incorrect” zeros. Nevertheless, we would like to stress that the overall error rates of our floating point division unit covert channel is very low.

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 4. 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.

#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.

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.

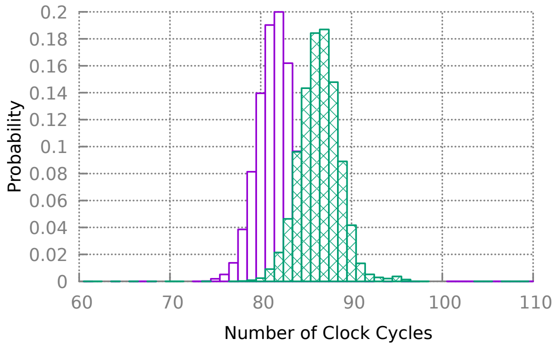
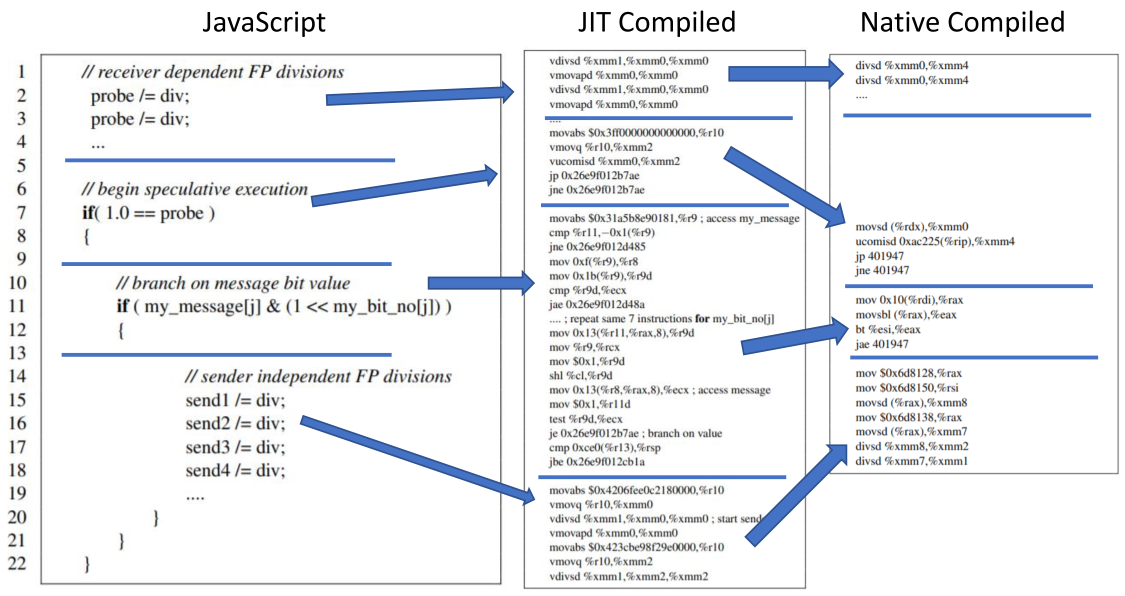


Fig. 6: Timing characteristics of division floating point unit covert channel execution in Google Chrome JavaScript sandbox

Similarly to the original Spectre attack PoC written in JavaScript [20], we developed a PoC that implements our floating point division unit covert channel in JavaScript, and successfully execute it on Google Chrome version 62.0.3202.75, which allows a website to read private memory from the process in which it runs. For a high resolution timer, as in [20], we also followed the approach described by Schwarz et al. [32], 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. [20].

Figure 7 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 6 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 specula-



```
// receiver dependent FP divisions
  probe /= div;
  probe /= div;
  ...

// begin speculative execution
if( 1.0 == probe )
{

    // branch on message bit value
    if ( my_message[j] & (1 << my_bit_no[j]) )
    {

        // sender independent FP divisions
        send1 /= div;
        send2 /= div;
        send3 /= div;
        send4 /= div;
        ....
    }
  }
}
```

```
vdivsd %xmm1,%xmm0,%xmm0
vmovapd %xmm0,%xmm0
vdivsd %xmm1,%xmm0,%xmm0
vmovapd %xmm0,%xmm0
....
movabs $0x3ff0000000000000,%r10
vmovq %r10,%xmm2
vucomisd %xmm0,%xmm2
jp 0x26e9f012b7ae
jne 0x26e9f012b7ae
movabs $0x31a5b8e90181,%r9 ; access my_message
cmp %r11,−0x1(%r9)
jne 0x26e9f012d485
mov 0xf(%r9),%r8
mov 0x1b(%r9),%r9d
cmp %r9d,%ecx
jae 0x26e9f012d48a
.... ; repeat same 7 instructions for my_bit_no[j]
mov 0x13(%r11,%rax,8),%r9d
mov %r9,%rcx
mov $0x1,%r9d
shl %cl,%r9d
mov 0x13(%r8,%rax,8),%ecx ; access message
mov $0x1,%r11d
test %r9d,%ecx
je 0x26e9f012b7ae ; branch on value
cmp 0xce0(%r13),%rsp
jbe 0x26e9f012cb1a
movabs $0x4206fee0c2180000,%r10
vmovq %r10,%xmm0
vdivsd %xmm1,%xmm0,%xmm0 ; start send
vmovapd %xmm0,%xmm0
movabs $0x423cbe98f29e0000,%r10
vmovq %r10,%xmm2
vdivsd %xmm1,%xmm2,%xmm2
```

```
divsd %xmm0,%xmm4
divsd %xmm0,%xmm4
....
movsd (%rdx),%xmm0
ucomisd 0xac225(%rip),%xmm4
jp 401947
jne 401947
mov 0x10(%rdi),%rax
movsbl (%rax),%eax
bt %esi,%eax
jae 401947
mov $0x6d8128,%rax
mov $0x6d8150,%rsi
movsd (%rax),%xmm8
mov $0x6d8138,%rax
movsd (%rax),%xmm7
divsd %xmm8,%xmm2
divsd %xmm7,%xmm1
```

Fig. 7: Excerpt from JavaScript covert channel code (Left), the assembly the JIT compiler created (Center), and the native generated assembly (Right)

tive 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 SpectreRewind in Meltdown

In this section, we demonstrate a complete cross-domain transient execution attack using the proposed SpectreRewind covert channel.

Our attack is based on the Meltdown attack [22], which allows transient instructions to access secrets belonging to other processes and security domains, including the OS and virtual machines. In a Meltdown attack, the attacker attempts to read from a privileged memory address, such as a kernel virtual address, from userspace. While architecturally such an access will generate an exception, a speculatively executed memory access can forward the secret data to a dependent load instruction, which encodes the secret into a cache state change, before it can be squashed.

In our modification, we simply surround the exception generating memory access with `DIVSD` sender and receiver instructions as shown in Figure 4. In more detail, we base our implementation on the original Meltdown open-source repository [3]. We modified a single function `libkdump_read(addr)` in `libkdump.c`, which reads a single byte from the given address (`addr`), to utilize our DIVSD based SpectreRewind covert channel. The rest of the code and other settings are unchanged. For the experiment, we used Intel i5-6500 processor and disabled the kernel page-table isolation (KPTI), the software based meltdown mitigation feature in Linux.

Note that because a Meltdown attack generates exceptions, it is necessary to suppress the exceptions. In the original PoC, either Intel's Transactional Synchronization Extension (TSX) or signal handling was used to suppress the exceptions. In our approach, however, an exception generating secret memory access can only occur in a mis-speculated transient execution, which will be squashed when the the receiver code has completed. Thus, we effectively suppress the exception without needing to use TSX or signal handling methods used in the original Meltdown PoC.

Method	# Reads	Success (%)
Original (SigHandle)	2197	97.78
Original (TSX)	217691	100.00
SpectreRewind	193399	99.97

Table 3: Performance of SpectreRewind covert channel (`DIVSD`) based Meltdown attack (Demo #3: Reliability test of the original Meltdown PoC repository) on Intel i5-6500.

Table 3 compares the performance of our modified Meltdown attack with the original ones, which utilize flush+reload based covert channels. Of the two original versions we evaluated, Original (SigHandle) suppresses exception by installing a signal handler while the Original (TSX) does so by utilizing TSX. We use the *reliability* PoC in the official Meltdown repository, which continuously

[3] `https://github.com/IAIK/meltdown`

reads a single byte from a kernel memory address and reports the number of reads and the success (i.e., correct reading) rate. In each configuration, we ran the reliability PoC for 60 seconds and measured the performance on the Intel Core i5-6500 (Skylake) processor, which supports TSX. As can be seen in the table, our SpectreRewind version of Meltdown performs significantly faster than the signal handler version of the original Meltdown in terms of the speed and the success rate, while it performs similarly compared to the TSX version of the original attack.

8 Discussion

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

8.1 Benefits and Limitations

SpectreRewind is a new type of contention-based covert channel, which is available in a wide range of microarchitectures 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., many ARM platforms, browser sandboxes).

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-core attack scenarios (e.g., [20]). Also, finding an exploitable gadget, which includes secret dependent division instructions, could be challenging 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 a CPU privilege mode change involves a 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, as we have demonstrated in Section 7.

8.2 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 re-design 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 [13] 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 [30] 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 [34, 8]. NDA [41] and STT [46] 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.

9 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 [43, 18, 27, 14] as a mean to defend against transient execution attacks. For example, InvisiSpec [43] and SafeSpec [18] 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 [14] implemented such a defense on an actual out-of-order open source RISC-V processor core. CleanupSpec [27] 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 [40] 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 [45] exploited L1 cache bank contention as a covert channel while MemJam [25] 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 [11] systematically explored possible contention-based covert channels by exhaustively executing instructions on different SMT threads of the same physical core. PortSmash [9] utilized port contention to create a microarchitectural side-channel to leak the secret key from a vulnerable version of OpenSSL. SmotherSpectre [7] 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 [20]). 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 [15] 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 [35] 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.

Concurrent to our work, Behnia et al., also observe that mis-speculated young instructions can delay non-speculative old instructions in out-of-order processors [6]. They then propose a simple but clever way to convert the resulting secret dependent timing difference into memory access ordering change, which in turn results in persistent cache state change that can be observed externally by an attacker. This transformation makes their attack, which they call speculative interference attack, be viable not only in the same-thread scenarios but also in the cross-core attack scenarios. In this sense, they enhanced the applicability of a SpectreRewind based attack. However, because their attack still requires certain interference gadgets (similar to our SpectreRewind gadget in Figure 4), it is mainly applicable when the attacker can control victim's instruction streams (e.g., JavaScript sandbox or Linux kernel eBPF [20]) and finding exploitable gadgets in real-world applications will be challenging as we discussed in Section 8.

10 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 showed that the covert channel can be used in the JavaScript sandbox of a Chrome browser. We also presented a complete end-to-end Meltdown attack using our covert channel. 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. Cache speculation side-channels. ARM White paper (2018)
2. Abel, A., Reineke, J.: uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 673–686. ACM, New York, NY, USA (2019)
3. Aciicmez, O., Seifert, J.P.: Cheap hardware parallelism implies cheap security. In: Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 80–91 (2007)
4. ARM: Cortex-A72 Software Optimization Guide. `https://static.docs.arm.com/uan0016/a/cortex_a72_software_optimization_guide_external.pdf` (2015)
5. ARM: Cortex-A57 Software Optimization Guide. `https://static.docs.arm.com/uan0015/b/Cortex_A57_Software_Optimization_Guide_external.pdf` (2016)
6. Behnia, M., Sahu, P., Paccagnella, R., Yu, J., Zhao, Z., Zou, X., Unterluggauer, T., Torrellas, J., Rozas, C., Morrison, A., Mckeen, F., Liu, F., Gabor, R., Fletcher, C.W., Basak, A., Alameldeen, A.: Speculative Interference Attacks: Breaking Invisible Speculation Schemes. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2020)
7. Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A.: Smotherspectre: exploiting speculative execution through port contention. In: ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 785–800 (2019)

8. Boggs, D.D., Segelken, R., Cornaby, M., Fortino, N., Chaudhry, S., Khartikov, D., Mooley, A., Tuck, N., Vreugdenhil, G.: Memory type which is cacheable yet inaccessible by speculative instructions (2019). US Patent App. 16/022,274
9. Cabrera Aldaya, A., Bob Brumley, B., ul Hassan, S., Pereida García, C., Tuveri, N.: Port contention for fun and profit. In: IEEE Symposium on Security and Privacy (SP) (2019)
10. Canella, C., Bulck, J.V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A systematic evaluation of transient execution attacks and defenses. In: USENIX Security Symposium (2019)
11. Fogh., A.: https://cyber.wtf/2016/09/27/covertshotgun/ (2016)
12. Fustos, J., Bechtel, M., Yun, H.: Spectrerewind: Leaking secrets to past instructions. In: Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security, pp. 117–126 (2020)
13. Fustos, J., Farshchi, F., Yun, H.: SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. In: Design Automation Conference (DAC), pp. 61–1 (2019)
14. Gonzalez, A., Korpan, B., Zhao, J., Younis, E., Asanović, K.: Replicating and mitigating spectre attacks on an open source risc-v microarchitecture. In: Third Workshop on Computer Architecture Research with RISC-V (CARRV) (2019)
15. Gras, B., Giuffrida, C., Kurth, M., Bos, H., Razavi, K.: Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In: Network and Distributed Systems Security (NDSS) (2020)
16. Horn, J.: speculative execution, variant 4: speculative store bypass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528` (2018)
17. Intel: Intel Analysis of Speculative Execution Side Channels (Rev. 4.0). Tech. rep. (2018). URL `https://software.intel.com/sites/default/files/managed/b9/f9/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf`
18. Khasawneh, K.N., Koruyeh, E.M., Song, C., Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In: Design Automation Conference (DAC) (2019)
19. Kiriansky, V., Waldspurger, C.: Speculative buffer overflows: Attacks and defenses. arXiv preprint arXiv:1807.03757 (2018)
20. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: IEEE Symposium on Security and Privacy (SP). IEEE Computer Society (2019)
21. Koruyeh, E.M., Khasawneh, K.N., Song, C., Abu-Ghazaleh, N.: Spectre returns! speculation attacks using the return stack buffer. In: USENIX Workshop on Offensive Technologies (WOOT) (2018)
22. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: USENIX Security (2018)
23. Maisuradze, G., Rossow, C.: ret2spec: Speculative execution using return stack buffers. In: ACM Conference on Computer and Communications Security (CCS), pp. 2109–2122. ACM (2018)
24. Minkin, M., Moghimi, D., Lipp, M., Schwarz, M., Van Bulck, J., Genkin, D., Gruss, D., Sunar, B., Piessens, F., Yarom, Y.: Fallout: Reading kernel writes from user space. In: ACM SIGSAC Conference on Computer and Communications Security (2019)
25. Moghimi, A., Wichelmann, J., Eisenbarth, T., Sunar, B.: Memjam: A false dependency attack against constant-time crypto implementations. International Journal of Parallel Programming (2019)
26. Oberman, S.F.: 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), pp. 106–115. IEEE (1999)
27. Saileshwar, G., Qureshi, M.K.: Cleanupspec: An "undo" approach to safe speculation. In: International Symposium on Microarchitecture (MICRO), p. 73–86. ACM (2019)
28. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue in-flight data load. In: S&P (2019)
29. van Schaik, S., Minkin, M., Kwong, A., Genkin, D., Yarom, Y.: CacheOut: Leaking data on Intel CPUs via cache evictions. `https://cacheoutattack.com/` (2020)
30. Schwarz, M., Lipp, M., Canella, C., Schilling, R., Kargl, F., Gruß, D.: Context: A generic approach for mitigating spectre. In: Network and Distributed System Security (NDSS) (2020)
31. Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross-privilege-boundary data sampling. In: ACM Conference on Computer and Communications Security (CCS) (2019)
32. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In: A. Kiayias (ed.) Financial Cryptography and Data Security, pp. 247–267. Springer International Publishing, Cham (2017)
33. Stecklina, J., Prescher, T.: Lazyfp: Leaking fpu register state using microarchitectural side-channels. arXiv preprint arXiv:1806.07480 (2018)
34. Sun, K., Branco, R., Hu, K.: A new memory type against speculative side channel attacks. `https://github.com/IntelSTORMteam/Papers` (2019)
35. Townley, D., Ponomarev, D.: Smt-cop: Defeating side-channel attacks on execution units in smt processors. In: 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT) (2019)
36. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on aes, and countermeasures. J. Cryptology **23**, 37–71 (2010)
37. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: Maximizing on-chip parallelism. In: International Symposium on Computer Architecture (ISCA), pp. 392–403. ACM (1995)
38. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: USENIX Security Symposium. USENIX Association (2018)
39. Van Bulck, J., Moghimi, D., Schwarz, M., Lipp, M., Minkin, M., Genkin, D., Yuval, Y., Sunar, B., Gruss, D., Piessens, F.: LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: 41th IEEE Symposium on Security and Privacy (S&P'20) (2020)
40. Wang, Z., Lee, R.B.: Covert and side channels due to processor architecture. In: Annual Computer Security Applications Conference (ACSAC), pp. 473–482 (2006)
41. Weisse, O., Neal, I., Loughlin, K., Wenisch, T.F., Kasikci, B.: Nda: Preventing speculative execution attacks at their source. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 572–586 (2019)
42. Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T.F., Yarom, Y.: Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report (2018)
43. Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C.W., Torrellas, J.: InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In: International Symposium on Microarchitecture (MICRO) (2018)
44. Yarom, Y., Falkner, K.: Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 719–732. USENIX Association, San Diego, CA (2014)

45. Yarom, Y., Genkin, D., Heninger, N.: Cachebleed: a timing attack on openssl constant-time rsa. Journal of Cryptographic Engineering (2017)
46. Yu, J., Yan, M., Khyzha, A., Morrison, A., Torrellas, J., Fletcher, C.W.: Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In: International Symposium on Microarchitecture (MICRO), pp. 954–968 (2019)