Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks

Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida,
Vrije Universiteit Amsterdam

https://www.usenix.org/conference/usenixsecurity21/presentation/ragab

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the
30th USENIX Security Symposium
is sponsored by USENIX.

Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks

Hany Ragab* hany.ragab@vu.nl
Enrico Barberis* e.barberis@vu.nl
Herbert Bos herbertb@cs.vu.nl
Cristiano Giuffrida giuffrida@cs.vu.nl

Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Equal contribution joint first authors

Abstract

Since the discovery of the Spectre and Meltdown vulnerabilities, transient execution attacks have increasingly gained momentum. However, while the community has investigated several variants to trigger attacks during transient execution, much less attention has been devoted to the analysis of the root causes of transient execution itself. Most attack variants simply build on well-known root causes, such as branch misprediction and aborts of Intel TSX—which are no longer supported on many recent processors.

In this paper, we tackle the problem from a new perspective, closely examining the different root causes of transient execution rather than focusing on new attacks based on known transient windows. Our analysis specifically focuses on the class of transient execution based on *machine clears* (MC), reverse engineering previously unexplored root causes such as Floating Point MC, Self-Modifying Code MC, Memory Ordering MC, and Memory Disambiguation MC. We show these events not only originate new transient execution windows that widen the horizon for known attacks, but also yield entirely new attack primitives to inject transient values (*Floating Point Value Injection or FPVI*) and executing stale code (*Speculative Code Store Bypass or SCSB*). We present an end-to-end FPVI exploit on the latest Mozilla SpiderMonkey JavaScript engine with all the mitigations enabled, disclosing arbitrary memory in the browser through attacker-controlled and transiently-injected floating-point results. We also propose mitigations for both attack primitives and evaluate their performance impact. Finally, as a by-product of our analysis, we present a new root cause-based classification of all known transient execution paths.

1 Introduction

Since the public disclosure of the Meltdown and Spectre vulnerabilities in 2018, researchers have investigated ways to use transient execution windows for crafting several new attack variants [6–11, 28, 40, 43, 48, 59, 63, 65, 67, 68, 71, 74–76, 78, 79, 82]. Building mostly on well-known causes of transient execution, such as branch mispredictions or aborting Intel TSX transactions, such variants violate many security boundaries, allowing attackers to obtain access to unauthorized data, divert control flow, or inject values in transiently executed code. Nonetheless, little effort has been made to systematically investigate the root causes of what Intel refers to as *bad speculation* [36]—the conditions that cause the CPU to discard already issued micro-operations (μOps) and render them *transient*. As a result, our understanding of these root causes, as well as their security implications, is still limited. In this paper, we systematically examine such causes with a focus on a major, largely unexplored class of bad speculation.

In particular, Intel [36] identifies two general causes of bad speculation (and transient execution)—branch misprediction and *machine clears*. Upon detecting branch mispredictions, the CPU needs to squash all the μOps executed in the the mispredicted branches. Such occurrences, in a wide variety of forms and manifestations, have been extensively examined in the existing literature on transient execution attacks [10, 11, 31, 40, 41, 43, 48]. The same cannot be said for machine clears, a class of bad speculation that relies on a full flush of the processor pipeline to restart from the last retired instruction. In this paper, we therefore focus on the systematic exploration of machine clears, their behavior, and security implications.

Specifically, by reverse engineering this undocumented class of bad speculation, we closely examine four previously unexplored sources of machine clears (and thus transient execution), related to floating points, self-modifying code, memory ordering, and memory disambiguation. We show attackers can exploit such causes to originate new transient execution windows with unique characteristics. For instance, attackers can exploit Floating Point Machine Clear events to embed attacks in transient execution windows that require no training or special (in many cases disabled) features such as Intel TSX. Besides providing a general framework to run existing and future attacks in a variety of transient execution windows with different constraints and leakage rates, our analysis also uncovered new attack primitives based on machine clears.

In particular, we show that Self-Modifying Code Machine Clear events allow attackers to transiently execute stale code, while Floating Point Machine Clear events allow them to inject transient values in victim code. We term these primitives *Speculative Code Store Bypass* (*SCSB*) and *Floating Point Value Injection* (*FPVI*), respectively. The former is loosely related to Speculative Store Bypass (SSB) [55, 82], but allows attackers to transiently reference stale code rather than data. The latter is loosely related to Load Value Injection (LVI) [71], but allows attackers to inject transient values by controlling operands of floating-point operations rather than triggering victim page faults or microcode assists. We also discuss possible gadgets for exploitation in applications such as JIT engines. For instance, we found that developers following instructions detailed in the Intel optimization manual [36] ad litteram could easily introduce SCSB gadgets in their applications. In addition, we show that attackers controlling JIT'ed code can easily inject FPVI gadgets and craft arbitrary memory reads, even from JavaScript in a modern browser using NaN-boxing such as Mozilla Firefox [53].

Moreover, since existing mitigations cannot hinder our primitives, we implement new mitigations to eliminate the uncovered attack surface. As shown in our evaluation, SCSB can be efficiently mitigated with serializing instructions in the practical cases of interest such as JavaScript engines. FPVI can be mitigated using transient execution barriers between the source of injection and its transmit gadgets, either inserted by the programmer or by the compiler. We implemented the general compiler-based mitigation in LLVM [45], measuring an overhead of 32% / 53% on SPECfp 2006/2017 (geomean).

While not limited to Intel, our analysis does build on a variety of performance counters available in different generations of Intel CPUs but not on other architectures. Nevertheless, we show that our insights into the root causes of bad speculation also generalize to other architectures, by successfully repeating our transient execution leakage experiments (we originally designed for Intel) on AMD.

Finally, armed with a deeper understanding of the root causes of transient execution, we also present a new classification for the resulting paths. Existing classifications [10] are entirely attack-centric, focusing on classes of Spectre- or Meltdown-like attacks and blending together (common) causes of transient execution with their uses (i.e., what attackers can do with them). Such classifications cannot easily accommodate the new transient execution paths presented in this paper. For this reason, we propose a new orthogonal, root cause-centric classification, much closer to the sources of bad speculation identified by the chip vendors themselves.

Summarizing, we make the following contributions:

- We systematically explore the root causes of transient execution and closely examine the major, largely unexplored machine clear-based class. To this end, we present the reverse engineering and security analysis of causes such as Floating Point Machine Clear (FP MC), Self-Modifying Code Machine Clear (SMC MC), Memory Ordering Machine Clear (MO MC), and Memory Disambiguation Machine Clear (MD MC).
- We present two novel machine clear-based transient execution attack primitives (FPVI and SCSB) and an end-to-end FPVI exploit disclosing arbitrary memory in Firefox.
- We propose and evaluate possible mitigations.
- We propose a new root-cause based classification of all the known transient execution paths.

Code, exploit demo, and additional information are available at https://www.vusec.net/projects/fpvi-scsb.

2 Background

2.1 IEEE-754 Denormal Numbers

Modern processors implement a Floating Point Unit (FPU). To represent floating point numbers, the IEEE-754 standard [1] distinguishes three components (Fig. 1). First, the most significant bit serves as the sign bit s. The next w bits contain a biased exponent e. For instance, for double precision 64-bit floating point numbers, e is 11 bits long and the bias is $2^{11-1}-1 = 1023$ (i.e., $2^{w-1}-1$). The value e is computed by adding the real exponent e_{real} to the bias, which ensures that e is a positive number even if the real exponent is negative. The remaining bits are used for the mantissa m. The combination of these components represents the value. As an example, suppose the sign bit $s = 1$, the biased exponent $e = 10000000011_2 = 1027_{10}$ so that $e_{real} = 1027 - 1023 = 4$, and the mantissa $m = 0111000...0$. In that case,the corresponding decimal value is $-1 \cdot 2^4 \cdot (1 + 0.0111_2)$. We refer to such numbers as *normal* numbers, as they are in the normalized form with an implicit leading 1 present in the mantissa.

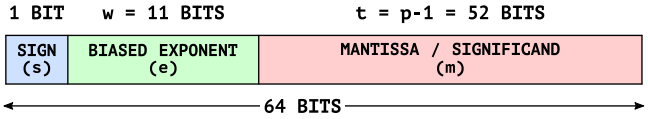


Figure 1: Fields of a 64 bits IEEE 754 double

If $e = 0$ and $m \neq 0$, the CPU treats the value as a *denormal* value. In this case, the leading 1 becomes a leading 0 instead, while the exponent becomes the minimal value of $2 - 2^{w-1}$, so that, with $s = 1$, $e = 0$ and $m = 0111000...0$, the resulting value for a double precision number is $-1 \cdot 2^{-1022} \cdot (0 + 0.0111_2)$. This additional representation is intended to allow a gradual underflow when values get closer to 0. In 64-bits floating point numbers, the smallest unbiased exponent is -1022, making it impossible to represent a number with an exponent of -1023 or lower. In contrast, a denormal number can represent this number by appending enough leading zeroes to the

mantissa until the minimal exponent is obtained. The denormal representation trades precision for the ability to represent a larger set of numbers. Generating such a representation is called *denormalization* or *gradual underflow*.

2.2 x86 Cache Coherence

On multicore processors, L1 and L2 caches are usually per core, while the L3 is shared among all cores. Much complexity arises due to the *cache coherence problem*, when the same memory location is cached by multiple cores. To ensure the correctness, the memory must behave as a single coherent source of data, even if data is spread across a cache hierarchy. Informally, a memory system is coherent if any read of a data item returns the most recently written value [30]. To obtain coherence, memory operations that affect shared data must propagate to other cores' private caches. This propagation is the responsibility of the *cache coherence protocol*, such as MESIF on Intel processors [37] and MOESI on AMD [3]. Cache controllers implementing these protocols snoop and exchange coherence requests on the shared bus. For example, when a core writes in a shared memory location, it signals all other cores to invalidate their now stale local copy.

The cache coherence policy also maintains the illusion of a unified memory model for backward compatibility. The original Intel 8086, released in 1978, operated in real-address mode to implement what is essentially a pure Von Neumann architecture with no separation between data and code. Modern processors have more sophisticated memory architectures with separate L1 caches for code (L1d) and data (L1i). The need for backward compatibility with the simple 8086 memory model, has led modern CPUs to a split-cache Harvard architecture whereby the cache coherence protocol ensures that the (L1) data and instruction caches are always coherent.

2.3 Memory Ordering

A Memory Consistency Model, or Memory Ordering Model, is a contract between the microarchitecture and the programmer regarding the permissible order of memory operations in parallel programs. Memory consistency is often confused with memory coherence, but where coherence hides the complex memory hierarchy in multicore systems, consistency deals with the *order* of read/write operations.

Consider the program in Figure 2. In the simplest consistency model, 'Sequential Consistency' (SC), all cores see all operations in the same order as specified by the program. In other words, A0 always execute before A1, and B0 always before B1. Thus, a valid memory order would be `A0-B0-A1-B1`, while `A1-B1-A0-B0` would be invalid.

Intel and AMD CPUs implement Total-Store-Order (TSO), which is equivalent to SC, apart from one case: a store followed by a load on a different address may be reordered. This allows cores to use a private store buffer to hide the latency of

X & Y are initially 0

PROCESSOR A	PROCESSOR B
(A0) [X] = 1	(B0) [Y] = 1
(A1) r1 = [Y]	(B1) r2 = [X]

Figure 2: Memory Ordering Example

store operations. In the example of Figure 2, the store operations `A0` and `B0` write their values initially only in their private store buffers. The subsequent loads, `A1` and `B1`, will now read the stale value 0, until the stores are globally visible. Thus, the order `A1-B1-A0-B0 (r1=0, r2=0)` is also valid.

2.4 Memory Disambiguation

Loads must normally be executed only after all the preceding stores to the same memory locations. However, modern processors rely on speculative optimizations based on *memory disambiguation* to allow loads to be executed before the addresses of all preceding stores are computed. In particular, if a load is *predicted* not to alias a preceding store, the CPU *hoists* the load and speculatively executes it before the preceding store address is known. Otherwise, the load is stalled until the preceding store is completed. In case of a *no-alias* (or *hoist*) misprediction, the load reads a stale value and the CPU needs to re-issue it after flushing the pipeline [36, 37].

3 Threat Model

We consider unprivileged attackers who aim to disclose confidential information, such as private keys, passwords, or randomized pointers. We assume an x86-64 based victim machine running the latest microcode and operating system version, with all state-of-the-art mitigations against transient execution attacks enabled. We also consider a victim system with no exploitable vulnerabilities apart from the ones described hereafter. Finally, we assume attackers can run (only) unprivileged code on the victim (e.g., in JavaScript, user processes, or VMs), but seek to leak data across security boundaries.

4 Machine Clears

The Intel Architectures Optimization Reference Manual [36] refers to the root cause of discarding issued μOp as *Bad Speculation*. Bad speculation consists of two subcategories:

- *Branch Mispredict*. A misprediction of the direction or target of a branch by the branch predictor will squash all μOps executed within a mispeculated branch.
- *Machine Clear* (or *Nuke*). A machine clear condition will flush the entire processor pipeline and restart the execution from the last retired instruction.

Table 1: Machine clear performance counters

Name	Description
MACHINE_CLEARS.COUNT	Number of machine clears of any type
MACHINE_CLEARS.SMC	Number of machine clears caused by a self/cross-modifying code
MACHINE_CLEARS.DISAMBIGUATION	Number of machine clears caused by a memory disambiguation unit misprediction
MACHINE_CLEARS.MEMORY_ORDERING	Number of machine clears caused by a memory ordering principle violation
MACHINE_CLEARS.FP_ASSIST	Number of machine clears caused by an assisted floating point operation
MACHINE_CLEARS.PAGE_FAULT	Number of machine clears caused by a page fault
MACHINE_CLEARS.MASKMOV	Number of machine clears caused by an AVX `maskmov` on an illegal address with a mask set to 0

Bad speculation not only causes performance degradation but also security concerns [6–8,31,41,43,47,55,59,63,71,74–76,79,82]. In contrast to branch misprediction, extensively studied by security researchers [10, 11, 31, 40, 41, 43, 48], machine clears have undergone little scrutiny. In this paper, we perform the first deep analysis of machine clears and the corresponding root causes of transient execution.

Since machine clears are hardly documented, we examined all the performance counters for every Intel architecture and found a number relevant counters (Table 1). Some counters are present only in specific architectures. For example, the page fault counter is available only on Goldmont Plus. However, thanks to the generic counter `MACHINE_CLEARS.COUNT` it is always possible to count the overall number of machine clears, regardless of the architecture. In the remainder of this work, we will reverse engineer and analyze the causes of machine clears by means of these six counters.

As a general observation, we note that the Floating Point Assist and Page Fault counters immediately suggest that machine clears are also related to microcode assists and faults/exceptions. In particular, further analysis shows that:

- *Microcode assists trigger machine clears*. The hardware occasionally needs to resort to microcode to handle complex events. Doing so requires flushing all the pending instructions with a machine clear before handling a *microcode assist*. Indeed, in our experiments, where the `OTHER_ASSISTS.ANY` counter increased, we also observed a matching increase in `MACHINE_CLEARS.COUNT`.
- *Machine clears do not necessarily trigger microcode assists*. Not all machine clears are microcode assisted, as some machine clear causes are handled directly in silicon. Indeed, in our experiments, we observed that SMC, MD, and MO machine clears cause an increase of `MACHINE_CLEARS.COUNT`, but leave `OTHER_ASSISTS.ANY` unaltered.
- *An exception triggers a machine clear*. When a fault or exception is detected, the subsequent μOps must be flushed from the pipeline with a machine clear, as the execution should resume at the exception handler. Indeed, in our experiments, we observed an increase of `MACHINE_CLEARS.COUNT` at each faulty instruction.

In this paper, we focus on the machine clear causes mentioned by the Intel documentation (Table 1), acknowledging that undocumented causes may still exist (much like undocumented x86 instructions [16]). We now first examine the four most relevant causes of machine clears (Self-Modifying Code MC, Floating Point MC, Memory Ordering MC, and Memory Disambiguation MC), then briefly discuss the other cases.

5 Self-Modifying Code Machine Clear

In a Von Neumann architecture, stores may write instructions as data and modify program code as it is being executed, as long as the code pages are writable. This is commonly referred to as Self-modifying Code (SMC).

Self-modifying code is problematic for the Instruction Fetch Unit (IFU), which maintains high execution throughput by aggressively prefetching the instructions it expects to execute next and feeding them to the decode units. The CPU speculatively fetches and decodes the instructions and feeds them to the execution units, well ahead of retirement.

In case of a misprediction, the CPU flushes the speculatively processed instructions and resumes execution at the correct target. The IFU's aggressive prefetching ensures that the first-level instruction cache (L1i) is constantly filled with instructions which are either currently in (possibly speculative) execution or about to be executed. As a result, a store instruction targeting code cached in L1i requires drastic measures—as the associated cache lines should now be invalidated. Moreover, the target instructions do not even need to be part of the actual execution: since a prefetch is sufficient to bring them into L1i, any write to prefetched instructions also invalidates the prefetch queue. In other words, the problem occurs when the code is already in L1i or the store is sufficiently close to the target code to ensure the target is prefetched in L1i. This behavior leads to a temporary desynchronization between the code and data views of the CPU, transiently breaking the architectural memory model (where L1d/L1i coherence ensures consistent code/data views).

To reverse engineer this behavior, we use the analysis code exemplified by Listing 1. The store at line 15 overwrites code already cached in L1i (lines 18-21), triggering a machine clear. The machine clear needs to update the L1i cache (and

related microarchitectural structures) by flushing any stale instruction(s), resuming execution at the last retired instruction, and then fetching the new instructions. To test for the presence of a transient execution path, our analysis code immediately executes lines 18-21 targeted by the store and jumps to a `spec_code` gadget (lines 29-32) which fills a number of cache lines in a (flushed) buffer. Architecturally, this gadget should never be executed, as the store instruction should nop out the branch at line 18. However, microarchitecturally, we do observe multiple cache hits in the (reload) buffer using FLUSH + RELOAD, which confirms the existence of a pre-SMC-handling transient execution path executing stale code and leaving observable traces in the cache. We observe that the scheduling of the store instruction heavily affects the length of the transient path. Indeed, we use different instructions (lines 5-8) to delay as much as possible the store retirement, and thus the SMC detection. This suggests that the root cause of the observed transient window might be the microarchitectural de-synchronization between the store buffer (new code) and the instruction queue (stale code), yielding transiently incoherent code/data views. We sampled machine clear performance counters to confirm the transient execution window is caused by the SMC machine clear and not by other events (e.g., memory disambiguation misprediction). Finally, we repeated our experiments on uncached code memory and could not observe any transient path. The counters revealed one machine clear triggered for each executed instruction, since the CPU has to pessimistically assume every fetched instruction has potentially been overwritten. Additionally, we have verified that SMC detection is performed on physical addresses rather than virtual ones.

Listing 1 Self-Modifying Code Machine Clear analysis code

```
1  smc_snippet:
2      push r11
3      lea  r11, [target] ; Load addr of target instr (line 17)
4
5      clflush [r11]   ; These instructions serve as a delay
6      %rep 10         ; for the store argument address. They
7      imul    r11, 1  ; ensure that the execution window of
8      %endrep         ; spec_code is as long as possible.
9
10 ;Code to write as data: 8 nops (overwriting lines 18-21)
11     mov     rax, 0x9090909090909090
12
13 ;Store at target addr. Also: the last retired instr
14 ;from which the execution will resume after the SMC MC
15     mov     QWORD [r11], rax
16
17 target:          ;Target instruction to be modified
18     jmp spec_code
19     nop
20     nop
21     nop
22
23 ;Architectural exit point of the function
24     pop r11
25     ret
26
27 ;Code executed speculatively (flushed after SMC MC).
28 spec_code:
29     mov rax, [rdi+0x0]   ; rdi: covert channel reload buffer
30     mov rax, [rdi+0x400]
31     mov rax, [rdi+0x800]
32     mov rax, [rdi+0xc00]
```

Cross-Modifying Code

Instead of modifying its own instructions, a thread running on one core may also write data into the currently executing code segment of a thread running on a different physical core. Such Cross-Modifying Code (XMC) may be synchronous (the executor thread waits for the writer thread to complete before executing the new code) or asynchronous, with no particular coordination between threads. To reverse engineer the behavior, we distributed our analysis code across cores and reproduced a signal on the reload buffer in both cases. This confirms a Cross-Modifying Code Machine Clear (XMC machine clear) behaves similarly to a SMC machine clear across cores, with a store on the writer core originating a transient execution window on the executor core.

6 Floating Point Machine Clear

On Intel, when the Floating Point Unit (FPU) is unable to produce results in the IEEE-754 [1] format directly, for instance in the case of *denormal* operands or results [4, 17], the CPU requires special handling to produce a correct result. According to an Intel patent [62], the denormalization is indeed implemented as a microcode assist or an exception handler since the corresponding hardware would be too complex. In our experiments, we observed microcode assists on all x87, SSE, and AVX instructions that perform mathematical operations on denormal floating-point (FP) numbers. Increments of the `FP_ASSIST.ANY`, or `MACHINE_CLEARS.COUNT` (or on older processors, `MACHINE_CLEARS.FP_ASSIST`) performance counters confirm such assists cause machine clears.

Since a machine clear implies a pipeline flush, the assisted FP operation will be squashed together with subsequent μOps. To reverse engineer the behavior, we used analysis code exemplified by Algorithm 1. Our code relies on a FLUSH + RELOAD [81] covert channel to observe the result of a floating-point operation at the byte granularity. In our experiments, we observed two different hits in the reload buffer for each byte, for the transient and architectural result, respectively. The double-hit microarchitectural trace confirms that the transient (and wrong) value generated by the FPU is used in subsequent μOps—as also exemplified in Table 2. Later, the CPU detects the error and triggers a machine clear to flush the wrongly executed path. The microcode assist then corrects the result, while subsequent instructions are reissued.

While we could not find any documentation on floating-point assist handling (even in the patents), our experiments revealed the following important properties. First, we verified that many FP operations can trigger FP assists (i.e., add, sub, mul, div and sqrt) across different extensions (i.e., x87, SSE, and AVX). Second, the *transient result* is computed by "blindly" executing the operation as if both operands and result

Algorithm 1 Floating Point Machine Clear analysis (pseudo) code. *byte* is used to extract the *i*-th byte of *z*

```
1: for i ← 1, 8 do
2:     flush(reload_buf)
3:     z = x / y                          ▷ Any denormal FP operation
4:     reload_buf[byte(z, i) * 1024]
5:     reload(reload_buf)
6: end for
```

	Representation	Value	Type	Exp.
x	0x001**0deadbeef1337**	2.34e-308	N	-1022
y	0x40f**0000000000000**	65536 (2^{16})	N	16
z_{arch}	0x000**00010deadbeef**	3.57e-313	D	-1022
z_{tran}	0x3f1**0deadbeef1337**	6.43e-05	N	-14

Table 2: Architectural (z_{arch}) and transient (z_{tran}) results of dividing x and y of Algorithm 1. N: Normal, D: Denormal representations. The mantissa is in bold. A *normal* division by 2^{16} leaves the mantissa untouched and subtracts 16 from the exponent—the result of z_{tran} where the exponent overflowed from -1022 to -14

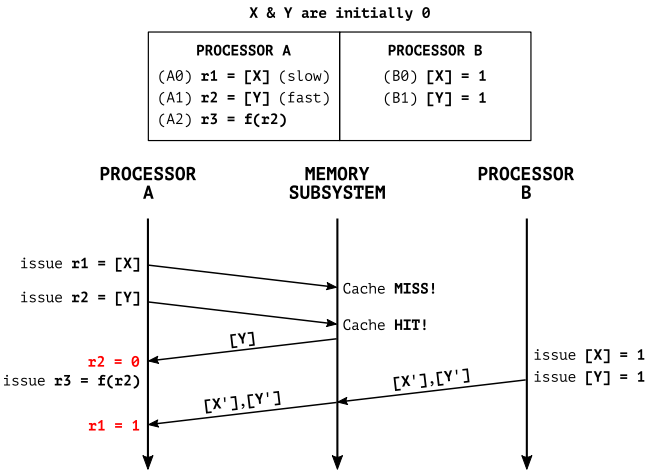


Figure 3: Transient execution due to invalid memory ordering

are normal numbers (see Table 2). Third, the detection of the wrong computation occurs later in time, creating a transient execution window. Finally, by performing multiple floating-point operations together with the assisted one, we were able to expand the size of the window, suggesting that detection is delayed if the FPU is busy handling multiple operations.

7 Memory Ordering Machine Clear

The CPU initiates a memory ordering (MO) machine clear when, upon receiving a snoop request, it is uncertain if memory ordering will be preserved [36]. Consider the program

Algorithm 2 Pseudo-code triggering a MO machine clear

```
Processor A
1: clflush(X)                          ▷ Make the load slow
2: unlock(lock)                        ▷ Synchronize loads and stores
3: r1 ← [X]
4: r2 ← [Y]
5: reload(r2)                          ▷ For FLUSH + RELOAD
Processor B
1: wait(lock)                          ▷ Synchronize loads and stores
2: 1 → [Y]
```

in Figure 3. Processor A loads X and Y, while processor B performs two stores to the same locations. If the load of X is slow due to a cache miss, the out-of-order CPU will issue the next load (and subsequent operations) ahead of schedule. Suppose that while the load of X is pending, processor B signals, through a snoop request, that the values of X and Y have changed. In this scenario, the memory ordering is `A1-B0-B1-A0`, which is not allowed according to the Total-Store-Order memory model. As two loads cannot be reordered, `r1=1 r2=0` is an illegal result. Thus, processor A has no choice other than to flush its pipeline and re-issue the load of Y in the correct order. This MO machine clear is needed for every inconsistent speculation on the memory ordering—implementing speculation behavior originally proposed by Gharachorloo et al. [27], with the advantage that strict memory order principles can co-exist with aggressive out-of-order scheduling.

Notice that, in the previous example, the store on X is not even necessary to cause a memory ordering violation since `A1-B1-A0` is still an invalid order. Counterintuitively, the memory ordering violation disappears if the load on X is not performed, as `A1-B0-B1` is a perfectly valid order.

To reverse engineer the memory ordering handling behavior on Intel CPUs, we used analysis code exemplified by Algorithm 2. Our code mimics the scenario of Figure 3 with loads/stores from two threads racing against each other, but relies on a FLUSH + RELOAD [81] covert channel to observe the loaded value of Y. The synchronization through the `lock` variable ensures the desired (problematic) proximity and ordering of the memory operations.

As before, in our experiments, we observed two different hits in the reload buffer for the loaded value of Y, one for the stale (transient) value and one for the new (architectural) value. For every double hit in the reload buffer, we also measured an increase of `MACHINE_CLEARS.MEMORY_ORDERING`. We also ran experiments without the load of X. Even though memory ordering violations are no longer possible since each processor executes a single memory operation and any order is permitted, we still observed MO machine clears. The reason is that Intel CPUs seem to resort to a simple but conservative approach to memory ordering violation detection, where the CPU initiates a MO machine clear when a snoop request from

another processor matches the source of any load operation in the pipeline. We obtained the same results with the two threads running across physical or logical (hyperthreaded) cores. Similarly, the results are unchanged if the matching store and load are performed on different addresses—the only requirement we observed is that the memory operations need to refer to the same cache line. Overall, our results confirm the presence of a transient execution window and the ability of a thread to trigger a transient execution path in another thread by simply dirtying a cache line used in a ready-to-commit load. Exploitation-wise, abusing this type of MC is non-trivial due to the strict synchronization requirements and the difficulty of controlling pending stale data.

8 Memory Disambiguation Machine Clear

As suggested by the `MACHINE_CLEARS.DISAMBIGUATION` counter description, memory disambiguation (MD) mispredictions are handled via machine clears. Our experiments confirmed this behavior by observing matching increases of `MACHINE_CLEARS.COUNT`. Moreover, we observed no changes in microcode assist counters, suggesting mispredictions are resolved entirely in hardware. In case of a misprediction, a stale value is passed to subsequent loads, a primitive that was previously used to leak secret information with Spectre Variant 4 or *Speculative Store Bypass* (SSB) [55, 82].

Different from the other machine clears, MD machine clears trigger only on address aliasing mispredictions, when the CPU wrongly predicts a load does not alias a preceding store instruction and can be hoisted. Similar to branch prediction, generating a MD-based transient execution window requires mistraining the underlying predictor. The latter has complex, undocumented behavior which has been partially reverse engineered [18]. For space constraints, we discuss our full reverse engineering strategy in Appendix A. Our results show that executing the same load $64+15$ times with non-aliasing stores is sufficient to ensure the next prediction to be "no-alias" (and thus the next load to be *hoisted*). Upon reaching the hoisting prediction, one can perform the load with an aliasing store to trigger the transient path exposed to the incorrect, stale value.

Finally, we experimentally verified that *4k aliasing* [50] does not cause any machine clear but only incurs a further time penalty in case of wrong aliasing predictions. More details on 4k aliasing results can be found in Appendix A.

9 Other Types of Machine Clear

AVX vmaskmov. The AVX `vmaskmov` instructions perform conditional packed load and store operations depending on a bitmask. For example, a `vmaskmovpd` load may read 4 packed doubles from memory depending on a 4-bit mask: each double will be read only if the corresponding mask bit is set, the others will be assigned the value 0.

According to the Intel Optimization Reference Manual [36], the instruction does not generate an exception in face of invalid addresses, provided they are masked out. However, our experiments confirm that it does incur a machine clear (and a microcode assist) when accessing an invalid address (e.g., with the present bit set to 0) with a loading mask set to zero (i.e., no bytes should be loaded) to check whether the bytes in the invalid address have the corresponding mask bits set or not. We speculate the special handling is needed because the permission check is very complex, especially in the absence of memory alignment requirements.

In our experiments, `vmaskmov` instructions with all-zero masks and invalid addresses increment the `OTHER_ASSIST.ANY` and `MACHINE_CLEARS.COUNT` counters, confirming that the instruction triggers a machine clear. However, the resulting transient execution window seems short-lived or absent, as we were unable to observe cache or other microarchitectural side effects of the execution.

Exceptions. The `MACHINE_CLEARS.PAGE_FAULT` counter, present on older microarchitectures, confirms page faults are another cause of machine clears. Indeed, we verified each instruction incurring a page fault or any other exception such as "Division by zero" increments the `MACHINE_CLEARS.COUNT` counter. We also verified exceptions do not trigger microcode assists and software interrupts (*traps*) do not trigger machine clears. Indeed, the Intel documentation [37] specifies that instructions following a trap may be fetched but not speculatively executed. Transient execution windows originating from exceptions—and page faults in particular—have been extensively used in prior work, with a faulty load instruction also used as the trigger to leak information [7, 47, 63, 75].

Hardware interrupts. Although hardware interrupts are an undocumented cause of machine clears, our experiments with APIC timer interrupts showed they do increment the `MACHINE_CLEARS.COUNT` counter. While this confirms hardware interrupts are another root cause of transient execution, the asynchronous nature of these events yields a less than ideal vector for transient execution attacks. Nonetheless, hardware interrupts play an important role in other classes of microarchitectural attacks [73].

Microcode assists. Microcode assists require a pipeline flush to insert the required μOps in the frontend and represent a subclass of machine clears (and thus a root cause of transient execution) for cases where a fast path in hardware is not available. In this paper, we detailed the behavior of floating-point and `vmaskmov` assists. Prior work has discussed different situations requiring microcode assists, such as those related to page table entry Access/Dirty bits, typically in the context of assisted loads used as the trigger to leak information [8, 63, 75]. AVX-to-SSE transitions [36] represent another microcode assist which based on our experiments we could not observe on modern Intel CPUs. Remaining known microcode assists such as access control of memory pages belonging to SGX

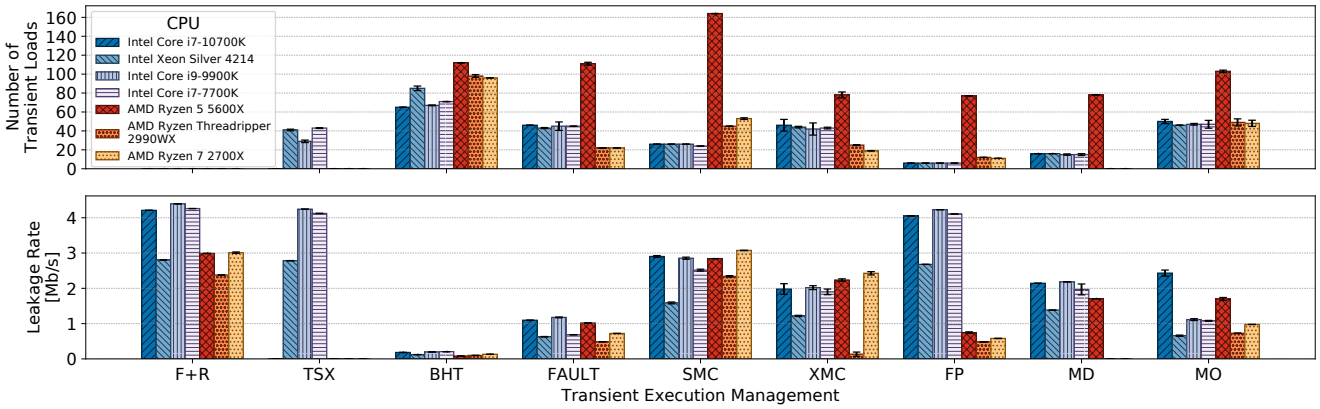


Figure 4: *Top plot*: transient window size vs. mechanism. Each bar reports the number of transient loads that complete and leave a microarchitectural trace. *Bottom plot*: leakage rate vs. mechanism for a simple Spectre Bounds-Check-Bypass attack and a 1-bit FLUSH + RELOAD (F+R) cache covert channel. F+R is the leakage rate upper bound (covert channel loop only, no actual transient window or attack).

secure enclaves and Precise Event Based Sampling (PEBS) are not presented in this work since already studied [14] or only related to privileged performance profiling respectively.

10 Transient Execution Capabilities

Transient execution attacks rely on crafting a *transient window* to issue instructions that are never retired. For this purpose, state-of-the-art attacks traditionally rely on mechanisms based on root causes such as branch mispredictions (BHT) [41, 43], faulty loads (Fault) [8, 47, 63, 75] or memory transaction aborts (TSX) [8, 47, 59, 63, 75]. However, the different machine clears discussed in this paper provide an attacker with the exact same capabilities.

To compare the capabilities of machine clear-based transient windows with those of more traditional mechanisms, we implemented a framework able to run arbitrary attacker-controlled code in a window generated by a mechanism of choosing. We now evaluate our framework on recent processors (with all the microcode updates and mitigations enabled) to compare the transient window size and leakage rate of the different mechanisms.

10.1 Transient Window Size

The transient window size provides an indication of the number of operations an attacker can issue on a transient path before the results are squashed. Larger windows can, in principle, host more complex attacks. Using a classic FLUSH + RELOAD cache covert channel (F+R) as a reference, we measure the window size by counting how many transient loads can complete and hit entries in a designated F+R buffer. Figure 4 (*top*) presents our results.

As shown in the figure, the window size varies greatly across the different mechanisms. Broadly speaking, mechanisms that have a higher detection cost such as XMC and MO Machine Clear, yield larger window sizes. Not surprisingly, branch mispredictions yield the largest window sizes, as we can significantly slow down the branch resolution process (i.e., causing cache misses) and delay detection. FP, on the other hand, yields the shortest windows, suggesting that denormal numbers are efficiently detected inside the FPU. Our results also show that, while our framework was designed for Intel processors, similar, if not better, results can usually be obtained on AMD processors (where we use the same conservative training code for branch/memory prediction). This shows that both CPU families share a similar implementation in all cases except for MD, where the used mistraining pattern is not valid for pre-Zen3 architectures.

10.2 Leakage Rate

To compare the leakage rates for the different transient execution mechanisms, we transiently read and repeatedly leak data from a large memory region through a classic F+R cache covert channel. We report the resulting leakage rates—as the number of bits successfully leaked per second—across different microarchitectures using a 1-bit covert channel to highlight the time complexity of each mechanism. We consider data to be successfully leaked after a single correct hit in the reload buffer. In case of a miss for a particular value, we restart the leak for the same value until we get a hit (or until we get 100 misses in a row).

As shown in Figure 4 (*bottom*), different Intel and AMD microarchitectures generally yield similar leakage rates with some variations. For instance, FP MC offers better leakage rates on Intel. This difference stems from the different perfor-

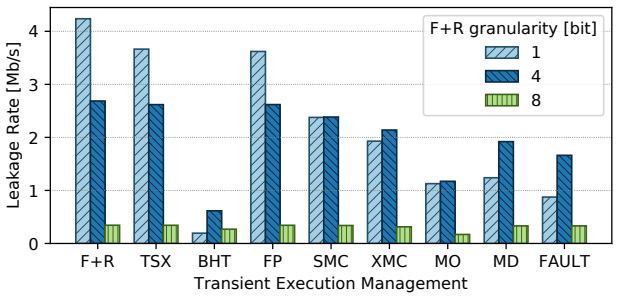


Figure 5: Leakage rate vs. mechanism with a 1-bit, 4-bit, or 8-bit FLUSH + RELOAD cache covert channel (Intel Core i9)

mance impact of the corresponding machine clears on Intel vs. AMD microarchitectures.

As shown in Figure 5, the leakage rate varies instead greatly across the different mechanisms and so does the optimal covert channel bitwidth. Indeed, while existing attacks typically rely on 8-bit covert channels, our results suggest 1-bit or 4-bit channels can be much more efficient depending on the specific mechanism. Roughly speaking, optimal leakage rates can be obtained by balancing the time complexity (and hence bitwidth) of the covert channel with that of the mechanism. For example, FP is a lightweight and reliable mechanism, hence using a comparably fast and narrow 1-bit covert channel is beneficial. In contrast, MD requires a time-consuming predictor training phase between leak iterations and leaking more bits per iteration with a 4-bit covert channel is more efficient. Interestingly, a classic 8-bit covert channel yields consistently worse and comparable leakage rates across all the mechanisms, since F+R dominates the execution time.

Our results show that only two mechanisms (TSX and FP) are close to the maximum theoretical leakage rate of pure F+R. Moreover, FP performs as efficiently as TSX, but, unlike TSX, is available on both Intel and AMD, is always enabled, and can be used from managed code (e.g., JavaScript). BHT, on the other hand, yield the worst leakage rates due to the inefficient training-based transient window. BHT leakage rate can be improved if a tailored mistrain sequence is used as in MD (Appendix A). Overall, our results show that machine clear-based windows achieve comparable and, in many cases (e.g., FP), better leakage rates compared to traditional mechanisms. Moreover, many machine clears eliminate the need for mistraining, which, other than resulting in efficient leakage rates, can escape existing pattern-based mitigations and disabled hardware extensions (e.g., Intel TSX).

11 Attack Primitives

Building on our reverse engineering results, and focusing on the unexplored SMC and FP machine clears, we now present two new transient execution attack primitives and analyze their security implications. We also present an end-to-end FPVI exploit disclosing arbitrary memory in Firefox. Later, we discuss mitigations.

11.1 Speculative Code Store Bypass (SCSB)

Our first attack primitive, Speculative Code Store Bypass (SCSB), allows an attacker to execute stale, controlled code in a transient execution window originated by a SMC machine clear. Since the primitive relies on SMC, its primary applicability is on JIT (e.g., JavaScript) engines running attacker-controlled code—although OS kernels and hypervisors storing code pages and allowing their execution without first issuing a serializing instruction are also potentially affected.

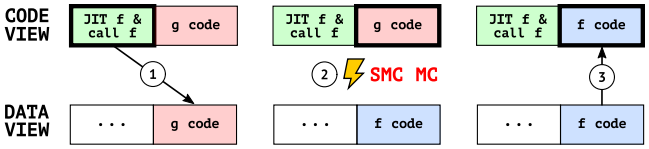
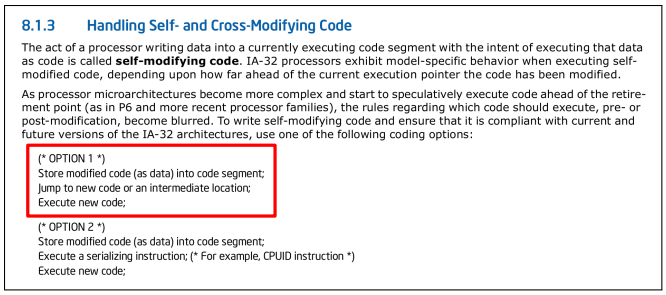


Figure 6: SCSB primitive example where the instruction pointer is pointing at the bold code blocks. `g code` is freed JIT'ed code (of some *g* function) under attacker's control. (1) Force engine to JIT and execute code of function *f* causing desynchronization of code and data views; (2) Execute stale code and SMC MC; (3) After the SMC MC, code and data view coherence is restored and the new code is executed.

As exemplified in Figure 6, the operations of the primitive can be broken down into three steps: (1) the JIT engine compiles a function *f*, storing the generated code into a JIT code cache region previously used by a (now-stale) version of function *g*; (2) the JIT engine jumps to the newly generated code for the function *f*, but due to the temporary desynchronization between the code and data views of the CPU, this causes transient execution of the stale code of *g* until the SMC machine clear is processed; (3) after the pipeline flush, the code and data views are resynchronized and the CPU restarts the execution of the correct code of *f*. For exploitation, the attacker needs to (i) massage the JIT code cache allocator to reuse a freed region with a target gadget *g* of choice; (ii) force the JIT engine to generate and execute new (*f*) code in such a region, enabling transient, out-of-context execution of the gadget and spilling secrets into the microarchitectural state.

Our primitive bears similarities with both transient and architectural primitives used in prior attacks. On the transient front, our primitive is conceptually similar to a Speculative-Store-Bypass (SSB) primitive [55, 82], but can transiently execute stale code rather than reading stale data. However, interestingly, the underlying causes of the two primitives are quite different (MD misprediction vs. SMC machine clear). On the architectural front, our primitive mimics classic Use-After-Free (UAF) exploitation on the JIT code cache, also



8.1.3 Handling Self- and Cross-Modifying Code

The act of a processor writing data into a currently executing code segment with the intent of executing that data as code is called **self-modifying code**. IA-32 processors exhibit model-specific behavior when executing self-modified code, depending upon how far ahead of the current execution pointer the code has been modified.

As processor microarchitectures become more complex and start to speculatively execute code ahead of the retirement point (as in P6 and more recent processor families), the rules regarding which code should execute, pre- or post-modification, become blurred. To write self-modifying code and ensure that it is compliant with current and future versions of the IA-32 architectures, use one of the following coding options:

(* OPTION 1 *)
Store modified code (as data) into code segment;
Jump to new code or an intermediate location;
Execute new code;

(* OPTION 2 *)
Store modified code (as data) into code segment;
Execute a serializing instruction; (* For example, CPUID instruction *)
Execute new code;

Figure 7: Coding options suggested by the Intel Architectures Software Developers Manuals to handle SMC and XMC execution. Option 1 describes the exact steps required by our Speculative Code Store Bypass attack primitive, potentially resulting in exploitable gadgets.

known as *Return-After-Free (RAF)* in the hackers community [19, 25]. An example is CVE-2018-0946, where a use-after-free vulnerability can be exploited to force the Chakra JS engine to erroneously execute freed (attacker-controlled) JIT code, resulting in arbitrary code execution after massaging the right gadget into the JIT code cache [24].

Indeed, at a high level, SCSB yields a *transient use-after-free* primitive on a JIT code cache, with exploitation properties similar to its architectural counterpart. However, there are some differences due to the transient nature of SCSB. First, we need to find an out-of-context gadget to transiently leak data rather than architecturally execute arbitrary code. In JavaScript engines, similar gadgets have already been exploited by ret2spec [48], escalating out-of-context transient execution of valid code to type confusion, arbitrary reads, and ultimately a secret-dependent load to transmit the value.

Second, we need to target short-lived JIT code cache updates, so that the newly generated code is immediately executed, fitting the target gadget in the resulting transient execution window. Interestingly, executing JIT'ed code is the next step after JIT code generation mentioned in the Intel manual (Figure 7, Option 1), suggesting that developers that follow such directives ad litteram can easily introduce such gadgets. Moreover, modern JavaScript compilers feature a multi-stage optimization pipeline [12, 54] and short-lived JIT code cache updates are favored for just-in-time (re)optimization. Indeed, we found test code in V8 to specifically test such updates and verify instruction/data cache coherence [70].

Finally, we need to ensure JIT code cache updates are not accompanied by barriers that force immediate synchronization of the code and data views. We analyzed the code of the popular SpiderMonkey and V8 JavaScript engines and verified that the functions called upon JIT code cache updates to synchronize instruction and data caches (Listing 2 and Listing 3) are always empty on x86 (as expected, given Intel's primary recommendation in Figure 7).

Overall, while the exploitation is far from trivial (i.e., having to address the challenges of traditional use-after-free exploitation as well as transient execution exploitation in the browser), we believe SCSB expands the attack surface of transient execution attacks in the browser. We found a number of candidate SCSB gadgets in real-world code, but none of them was ultimately exploitable due to the coincidental presence of some serializing instruction preventing stale code execution. Nonetheless, mitigations are needed to enforce security-by-design. Luckily, as shown later, SCSB is amenable to a practical and efficient mitigation (i.e., at a similar cost as faced by non-x86 architectures). The implementation/performance cost for mitigation is even lower than for its sibling SSB primitive, whose mitigation has been deployed in practice even in absence of known practical exploits [55].

Listing 2 Chromium instruction cache flush (`chromium/src/v8/src/codegen/x64/cpu-x64.cc`)

```
void CpuFeatures::FlushICache(void* start, size_t size) {
/* No need to flush the instruction
cache on Intel */ ...}
```

Listing 3 Firefox instruction cache flush (`mozilla-unified/js/src/jit/FlushICache.h`)

```
inline void FlushICache(void* code, size_t size,
       bool codeIsThreadLocal = true) {
/* No-op. Code and data caches are coherent on x86
↪  and x64. */ }
```

In Table 3, we show that all tested Intel and AMD processors are affected by SCSB. In contrast, ARM is not vulnerable since SMC updates require explicit software barriers.

11.2 Floating Point Value Injection (FPVI)

Our second attack primitive, Floating Point Value Injection (FPVI), allows an attacker to inject arbitrary values into a transient execution window originated by a FP machine clear. As exemplified in Figure 8, the operations of the primitive can be broken down into four steps: (1) the attacker triggers the execution of a gadget starting with a denormal FP operation in the victim application, with the x and y operands under attacker's control; (2) the transient z result of the operation is processed by the subsequent gadget instructions, leaving a microarchitectural trace; (3) the CPU detects the error condition (i.e., wrong result of a denormal operation), triggering a machine clear and thus a pipeline flush; (4) the CPU re-executes the entire gadget with the correct architectural z result. For exploitation, the attacker needs to (i) massage the x and y operands to inject the desired z value into the victim transient path and (ii) target a victim gadget so that the injected value yields a security-sensitive trace which can be observed with FLUSH + RELOAD or other microarchitectural attacks.

Our primitive bears similarities with Load-Value-Injection (LVI) [71], since both allow attackers to inject controlled values into transient execution. Moreover, both primitives

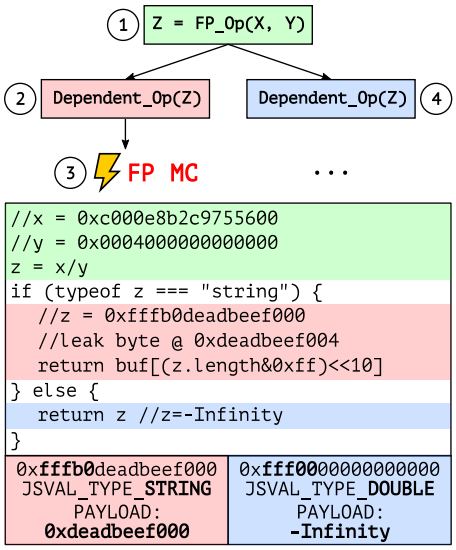


Figure 8: FPVI gadget example in SpiderMonkey. `FP_Op(x,y)` is an arbitrary denormal FP operation. The erroneous z result causes dependent operations to be executed twice (first transiently, then architecturally). The NaN-boxing z encoding allows the attacker to type-confuse the JIT'ed code and read from an arbitrary address on the transient path.

require gadgets in the victim application to process the injected value and perform security-sensitive computations on the transient path. Nonetheless, the underlying issues and hence the triggering conditions are fairly different. LVI requires the attacker to induce faulty or assisted loads on the victim execution, which is straightforward in SGX applications but more difficult in the general case [71]. FPVI imposes no such requirement, but does require an attacker to directly or indirectly control operands of a floating-point operation in the victim. Nonetheless, FPVI can extend the existing LVI attack surface (e.g., for compute-intensive SGX applications processing attacker-controlled inputs) and also provides exploitation opportunities in new scenarios. Indeed, while it is hard to draw general conclusions on the availability of exploitable FPVI gadgets in the wild—much like Spectre [41], LVI [71], etc., this would require gadget scanners subject of orthogonal research [56, 58]—we found exploitable gadgets in NaN-Boxing implementations of modern JIT engines [53]. NaN-Boxing implementations encode arbitrary data types as double values, allowing attackers running code in a JIT sandbox (and thus trivially controlling operands of FP operations) to escalate FPVI to a *speculative type confusion* primitive. The latter can be exploited similarly to NaN-Boxing-enabled architectural type confusion [26] and allows an attacker to access arbitrary data on a transient path. Figure 8 presents our end-to-end exploit for a JavaScript-enabled attacker in a SpiderMonkey (Mozilla JavaScript runtime) sandbox, illustrating a gadget unaffected by all the prior Mozilla Firefox' mitigations against transient execution attacks [52].

As exemplified in the figure, SpiderMonkey's NaN-Boxing strategy represents every variable with a IEEE-754 (64-bit) double where the highest 17 bits store the data type tag and the remaining 47 bits store the data value itself. If the tag value is less than or equal to `0x1fff0` (i.e., `JSVAL_TAG_MAX_DOUBLE`) all the 64 bits are interpreted as a double, while NaN-Boxing encoding is used otherwise. For instance, the `0xfff88` tag is used to represent 32-bit integers and the `0xfffb0` tag to represent a string with the data value storing a pointer to the string descriptor. In the example, the attacker crafts the operands of a vulnerable FP operation (in this case a division) to produce a transient result which the JIT'ed code interprets as a string pointer due to NaN-Boxing. This causes type confusion on a transient path and ultimately triggers a read with an attacker-controlled address.

To verify the attacker can inject arbitrary pointers without fully reverse engineering the complex function used by the FPU, we implemented a simple fuzzer to find FP operands that yield transient division results with the upper bits set to `0xfffb0` (i.e., string tag). With such operands, we can easily control the remaining bits by performing the inverse operation since the mantissa bits are transiently unaffected by the exponent value, as shown in Table 2. For example, using `0xc000e8b2c9755600` and `0x0004000000000000` as division operands yields *-Infinity* as the architectural result and our target string pointer `0xfffb0deadbeef000` as the transient result (see gadget in Figure 8).

Note that SpiderMonkey uses no guards or Spectre mitigations when accessing the attribute `length` of the string. This is normally safe since x86 guarantees that NaN results of FP operations will always have the lowest 52 bits set to zero—a representation known as *QNaN Floating-Point Indefinite* [37]. In other words, the implementation relies on the fact that NaN-boxed variables, such as string pointers, can never accidentally appear as the result of FP operations and can only be crafted by the JIT engine itself. Unfortunately, this invariant no longer holds on a FPVI-controlled transient path. As shown in Figure 8, this invariant violation allows an attacker to transiently read arbitrary memory. Since the `length` attribute is stored 4 bytes away from the string pointer, the `z.length` access yields a transient read to `0xdeadbeef000+4`.

We ran our exploit on an Intel i9-9900K CPU (microcode 0xde) running Linux 5.8.0 and Firefox 85.0 and, by wrapping this primitive with a variant of an EVICT + RELOAD covert channel [61], we confirmed the ability to read arbitrary memory. Since prior work has already demonstrated that custom high-precision timers are possible in JavaScript [26, 29, 60, 64], we enabled precise timers in Firefox to simplify our covert channel. With our exploit, we measured a leakage rate of ~13 KB/s and a transient window of ~12 load instructions, enabled by increasing the FPU pressure through a chain of multiple dependent FP operations.

Finally, as shown in Table 3, we observe that both Intel and

Table 3: Tested processors.

Processor	Microcode	SCSB vulnerable	FPVI vulnerable
Intel Core i7-10700K	0xe0	✓	✓
Intel Xeon Silver 4214	0x500001c	✓	✓
Intel Core i9-9900K	0xde	✓	✓
Intel Core i7-7700K	0xca	✓	✓
AMD Ryzen 5 5600X	0xa201009	✓	✓†
AMD Ryzen 2990WX	0x800820b	✓	✓†
AMD Ryzen 7 2700X	0x800820b	✓	✓†
Broadcom BCM2711 Cortex-A72 (ARM v8)	—	✗§	✗

† No exploitable NaN-boxed transient results found
§ On ARM, SMC updates require explicit software barriers

AMD are affected by FPVI, although we found no exploitable transient NaN-boxed values on AMD. On ARM, we never observed traces of transient results, yet we cannot rule out other FPU implementations being affected.

12 Mitigations

12.1 SCSB Mitigation

SCSB can be mitigated by ensuring that any freshly written code is architecturally visible before being executed. For example, on ARM architectures, where the hardware does not automatically enforce cache coherence, explicit serializing instructions (i.e., L1i cache invalidation instructions) are needed to correctly support SMC [5]. As such, spec-compliant ARM implementations cannot be affected by SCSB. On Intel and AMD, we can force eager code/data coherence using a serialization instruction—although this is normally not necessary (see Option 1 in Figure 7). In our experiments, we verified any serialization instruction such as `lfence`, `mfence`, or `cpuid` placed after the SMC store operations is indeed sufficient to suppress the transient window. Note that `sfence` cannot serve as a serialization instruction [35] to eliminate the transient path. This serialization mitigation was confirmed by CPU vendors and adopted by the Xen hypervisor [32, 33].

To evaluate the performance impact of our mitigation, we added a `lfence` instruction inside the `FlushICache` function (Listings 2 and 3) of the two popular V8 and SpiderMonkey JavaScript engines. Such function, normally empty on x86, is called after every code update. Our repeated experiments on the popular JetStream2 and Speedometer2 [77] benchmarks did not produce any statically measurable performance overhead. This shows JavaScript execution time is heavily dominated by JIT code generation/execution and code updates have negligible impact. Our results show this mitigation is practical and can hinder SCSB-based attack primitives in JIT engines with a 1-line code change.

12.2 FPVI Mitigation

The most efficient way to mitigate FPVI is to disable the denormal representation. On Intel, this translates to enabling the *Flush-to-Zero* and *Denormals-are-Zero* flags [37], which respectively replace denormal results and inputs with zero. This is a viable mitigation for applications with modest floating-point precision requirements and has also been selectively applied to browsers [42]. However, this defense may break common real-world (denormal-dependent) applications, a concern that has led browser vendors such as Firefox to adopt other mitigations. Another option for browsers is to enable Site Isolation [13], but JIT engines such as SpiderMonkey still do not have a production implementation [23]. Yet another option for JIT engines is to conditionally mask (i.e., using a transient execution-safe `cmov` instruction) the result of FP operations to enforce QNaN Floating-Point Indefinite semantics [37], as done in the SpiderMonkey FPVI mitigation [51]. This strategy suppresses any malicious NaN-boxed transient results, but requires manual changes to the NaN-boxing implementation and only applies to NaN-boxed gadgets.

A more general and automated mitigation is for the compiler to place a serializing instruction such as `lfence` after FP operations whose (attacker-controlled) result might leak secrets by means of *transmit gadgets* (or *transmitters*). We observe this is the same high-level strategy adopted by the existing LVI mitigation in modern compilers [39], which identifies loaded values as sources and uses data-flow analysis to ensure all the sources that reach a sink (*transmitter*) are fenced. Hence, to mitigate FPVI, we can rely on the same mitigation strategy, but use computed FP values as sources instead. To identify sinks, we consider both systems vulnerable and those resistant to Microarchitectural Data Sampling (MDS) [8, 59, 63, 75, 76]. For the former, we consider both load and store instructions as sinks (e.g., FP operation result used as a load/store address), as the corresponding arbitrary data spilled into microarchitectural buffers may be leaked by a MDS attacker. For the latter, we limit ourselves to load sinks to catch all the arbitrary read values potentially disclosed by a dependent transmitter. In both cases, we add indirect branches controlled by FP operations (and thus potentially leading to speculative control-flow hijacking) to the list of sinks.

We have implemented such a mitigation in LLVM [45], with only 100 lines of code on top of the existing *x86 LVI load hardening* pass. To evaluate the performance impact of our mitigation on floating-point-intensive programs, we ran all the C/C++ SPECfp 2006 and 2017 benchmarks in four configurations: LVI instrumentation, FPVI instrumentation for both MDS-vulnerable and MDS-resistant systems, and joint LVI+FPVI instrumentation for MDS-vulnerable systems. Please note that on MDS-resistant systems, the FPVI transmitters are already covered by the LVI mitigation.

Figure 9 shows the performance overhead of such configurations compared to the baseline. As expected, the LVI

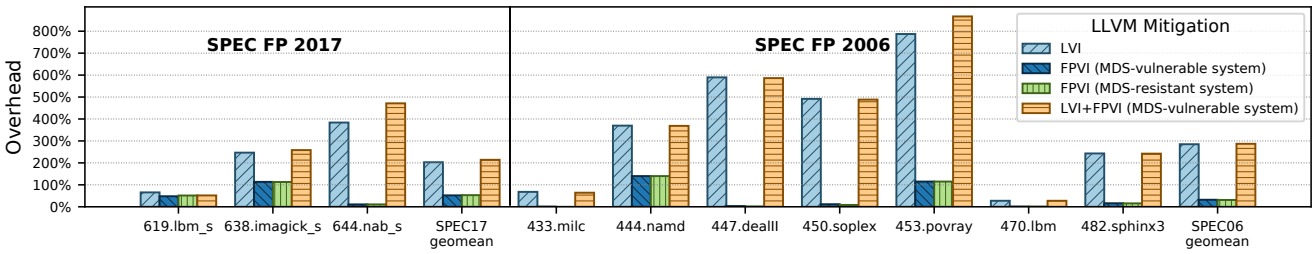


Figure 9: Performance overhead of our FPVI mitigation on the C/C++ SPECfp 2006 and 2017 benchmarks. Experimental setup: 5 SPEC iterations, Intel i9-9900K (microcode 0xde), and LLVM 11.1.0.

mitigation has non-trivial performance impact up to 280% despite targeting floating-point-intensive programs. On the other hand, our FPVI mitigation incurs in a 32 % and 53 % geomean overhead on SPECfp 2006 and 2017 respectively, with no observable performance impact difference between MDS-vulnerable and MDS-resistant variants. We observed that approximately 70% of the inserted `lfence` instructions are due to the intraprocedural design of the original LVI pass, forcing our analysis to consider every callsite with FPVI source-based arguments as a potential transmitter. This suggests the overhead can be further reduced by operating interprocedural analysis or more aggressive inlining (e.g., using LLVM LTO [45]).

13 Root Cause-based Classification

We now summarize the results of our investigation by presenting a root cause-based classification for the known transient execution paths. While there have already been several attempts to classify properties of transient execution [10,75,80], all the existing classifications are attack-centric. While certainly useful, such classifications inevitably blend together the root causes of transient execution with the attack triggers. For example, a MDS exploit based on a demand-paging page fault [75] may be simply classified as a Meltdown-like attack based on a present page fault [10]. However, in such an exploit the page fault is both the vulnerability trigger and the root cause of the transient execution window. A similar exploit may instead be embedded in an Intel TSX window, yielding a different root cause and capabilities.

To better characterize the capabilities of transient execution exploits, we propose the orthogonal root-cause-based classification in Figure 10. We draw from the Intel terminology to define the two main classes of root causes of ***Bad Speculation*** (i.e., transient execution): ***Control-Flow Misprediction*** (i.e., branch misprediction) and ***Data Misprediction*** (i.e., machine clear). Based on these two main classes, we observe that all the known root causes of transient execution paths can be classified into the following four subclasses: ***Predictors***, ***Exceptions***, ***Likely invariants violations***, and ***Interrupts***.

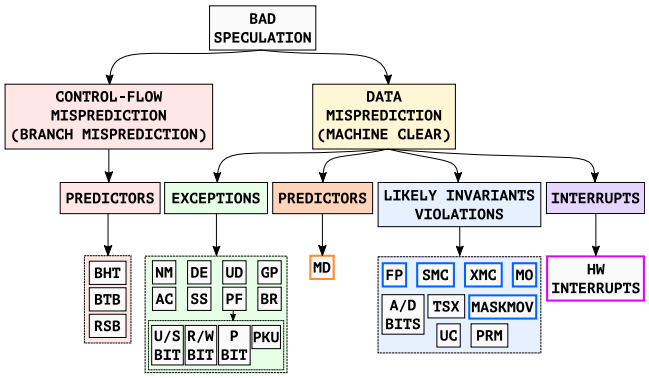


Figure 10: A root cause-centric classification of known transient execution paths (causes analyzed in the paper in **bold**). Acronyms descriptions can be found in Appendix B

Predictors. This category includes the prediction-based causes of bad speculation due to either control-flow or data mispredictions. Mistraining a predictor and forcing a misprediction is sufficient to create a transient execution path accessing erroneous code or data. It's worth noting that in Figure 10 there are two separate "predictors" subclasses under control-flow and data mispredictions, as they are different in nature. While control-flow mispredictions are failed attempts to guess the next instructions to execute, data mispredictions are failed attempts to operate on not-yet-validated data. Misprediction is a common way to manage transient execution windows in attacks like Spectre and derivatives [11,20,28,40,41,43,48,55,65,82].

Exceptions. This class includes the causes of machine clear due to exceptions, for instance, the different (sub)classes of page faults. Forcing an exception is sufficient to create a transient execution path erroneously executing code following the exception-inducing instruction. Exceptions are a less common way to manage transient execution windows (as they require dedicated handling), but have been extensively used as triggers in Meltdown-like attacks [7,8,47,59,63,67,71,74–76,78].

Interrupts. This class includes the causes of machine clear due to hardware (only) interrupts. Similar to exceptions, forc-

ing a HW interrupt is sufficient to create a transient execution path erroneously executing code following the interrupted instruction. Hardware interrupt are asynchronous by nature and thus difficult to control, resulting in a less than ideal way to manage transient execution windows. Nonetheless they were abused by prior side-channel attacks [72].

Likely invariants violations. This class includes all the remaining causes of machine clear, derived by *likely invariants* [15] used by the CPU. Such invariants commonly hold, but occasionally fail, allowing hardware to implement fast-path optimizations. However, compared to exceptions and interrupts, slow-path occurrences are typically more frequent, requiring more efficient handling in hardware or microcode. We discussed examples of such invariants in the paper (e.g., store instructions are expected to never target cached instructions, floating-point operations are expected to never operate on denormal numbers, etc.) and their lazy handling mechanisms (i.e., L1d/L1i resynchronization, microcode-based denormal arithmetic). Forcing a likely invariant violation is sufficient to create a transient execution path accessing erroneous code or data. In this paper, we have shown such violations are not only a realistic way to manage transient execution windows, but also provide new opportunities and primitives for transient execution attacks.

14 Related Work

Spectre [31,41] and Meltdown [47] first examined the security implications of transient execution, originating a large body of research on transient execution attacks [6–11,28,40,43,48,59,63,65,67,68,71,74–76,78,79,82]. Rather than focusing on attacks and their classification [10,75,80], ours is the first effort to systematize the root causes of transient execution and examine the many unexplored cases of machine clears.

We now briefly survey prior security efforts concerned with the major causes of machine clear discussed in this paper. Self-modifying code is commonly used by malware as an obfuscation technique [69] and has also been used to improve side-channel attacks by means of performance degradation [2]. Moreover, our SCSB primitive bears similarities with prior transient execution primitives inducing speculative control flow hijacking, either through branch target injection [41,49] or architectural branch target corruption [28]. The performance variability of floating-point operations on denormal numbers [17,46] has been previously exploited in traditional timing side-channel attacks [4]. Speculation introduced by stricter memory models is a well-known concept in the computer architecture literature [27,30,66]. While this is non-trivial to exploit, prior work did demonstrate information disclosure [57,79] by exploiting the snoop protocol discussed in Section 7. The memory disambiguation predictor has been previously abused to leak stale data in Spectre Speculative Store Bypass exploits [55,82]. Moreover, its behavior has been partially reverse engineered before [18]. In contrast to all these efforts, we focus on machine clears to systematically study all the root causes of transient execution, fully reverse engineering their behavior, and uncovering their security implications well beyond the state of the art.

15 Conclusions

We have shown that the root causes of transient execution can be quite diverse and go well beyond simple branch misprediction or similar. To support this claim, we systematically explored and reverse engineered the previously unexplored class of bad speculation known as machine clear. We discussed several transient execution paths neglected in the literature, examining their capabilities and new opportunities for attacks. Furthermore, we presented two new machine clear-based transient execution attack primitives (Floating Point Value Injection and Speculative Code Store Bypass). We also presented an end-to-end FPVI exploit disclosing arbitrary memory in Firefox and analyzed the applicability of SCSB in real-world applications such as JIT engines. Additionally, we proposed mitigations and evaluated their performance overhead. Finally, we presented a new root cause-based classification for all the known transient execution paths.

Disclosure

We disclosed Floating Point Value Injection and Speculative Code Store Bypass to CPU, browser, OS, and hypervisor vendors in February 2021. Following our reports, Intel confirmed the FPVI (CVE-2021-0086) and SCSB (CVE-2021-0089) vulnerabilities, rewarded them with the Intel Bug Bounty program, and released a security advisory with recommendations in line with our proposed mitigations [34]. Mozilla confirmed the FPVI exploit (CVE-2021-29955 [21, 22]), rewarded it with the Mozilla Security Bug Bounty program, and deployed a mitigation based on conditionally masking malicious NaN-boxed FP results in Firefox 87 [51].

Acknowledgments

We thank our shepherd Daniel Genkin and the anonymous reviewers for their valuable comments. We also thank Erik Bosman from VUSec and Andrew Cooper from Citrix for their input, Intel and Mozilla engineers for the productive mitigation discussions, Travis Downs for his MD reverse engineering, and Evan Wallace for his Float Toy tool. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and 825377 (UNICORE), by Intel Corporation through the Side Channel Vulnerability ISRA, and by the Dutch Research Council (NWO) through the INTERSECT project.

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std. 754-2019*, 2019.

[2] Alejandro Cabrera Aldaya and Billy Bob Brumley. Hyperdegrade: From ghz to mhz effective cpu frequencies. *arXiv preprint arXiv:2101.01077*.

[3] AMD. AMD64 Architecture Programmer's Manual.

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

[5] ARM. Architecture Reference Manual for Armv8-A.

[6] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *CCS'19*.

[7] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security'18*.

[8] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS'19*.

[9] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *ACM ASIA CCS 2020*.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security 19*.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE EuroS&P*.

[12] Chrome. V8 TurboFan documentation.

[13] Chromium. Site Isolation documentation.

[14] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016.

[15] David Devecsery, Peter M Chen, Jason Flinn, and Satish Narayanasamy. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. In *ASPLOS 2018*.

[16] Christopher Domas. Breaking the x86 isa. *Black Hat, USA*, 2017.

[17] Isaac Dooley and Laxmikant Kale. Quantifying the interference caused by subnormal floating-point values. In *Proceedings of the Workshop on OSIHPA*, 2006.

[18] Travis Downs. Memory Disambiguation on Skylake. https://github.com/travisdowns/uarch-bench/wiki/Memory-Disambiguation-on-Skylake, 2019.

[19] Thomas Dullien. Return after free discussion. https://twitter.com/halvarflake/status/1273220345525415937.

[20] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[21] Firefox. Firefox 87 Security Advisory. https://www.mozilla.org/en-US/security/advisories/mfsa2021-10/#CVE-2021-29955.

[22] Firefox. Firefox ESR 78.9 Security Advisory. https://www.mozilla.org/en-US/security/advisories/mfsa2021-11/#CVE-2021-29955.

[23] Firefox. Project Fission documentation.

[24] Fortinet. Use-After-Free Bug in Chakra (CVE-2018-0946). https://www.fortinet.com/blog/threat-research/an-analysis-of-the-use-after-free-bug%-in-microsoft-edge-chakra-engine.

[25] Ivan Fratric. Return after free discussion. https://twitter.com/ifsecure/status/1273230733516177408.

[26] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P*, May 2018.

[27] Kourosh Gharachorloo, Anoop Gupta, and John L Hennessy. *Two techniques to enhance the performance of memory consistency models*. 1991.

[28] Enes Goktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*, 2020.

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

[30] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[31] Jann Horn. Reading privileged memory with a side-channel. 2018.

[32] Xen Hypervisor. block_speculation function call in invoke_stub. https://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/arch/x86/x86_emulate/x86_emulate.c;hb=HEAD.

[33] Xen Hypervisor. block_speculation function call in io_emul_stub_setup. https://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/arch/x86/pv/emul-priv-op.c;hb=HEAD.

[34] Intel. FPVI & SCSB Intel Security Advisoray 00516. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00516.html.

[35] Intel. INTEL-SA-00088 - Bounds Check Bypass .

[36] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual.

[37] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual combined volumes.

[38] Intel. Intel® VTune™ Profiler User Guide - 4K Aliasing. https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/cpu-metrics-reference/l1-bound/aliasing-of-4k-address-offset.html.

[39] Intel. Load value injection - deep dive. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-load-value-injection, 2020.

[40] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv:1807.03757*.

[41] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P'19*.

[42] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security Symposium*, 2017.

[43] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX WOOT'18*.

[44] Evgeni Krimer, Guillermo Savransky, Idan Mondjak, and Jacob Doweck. Counter-based memory disambiguation techniques for selectively predicting load/store conflicts, October 1 2013. US Patent 8,549,263.

[45] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[46] Orion Lawlor, Hari Govind, Isaac Dooley, Michael Breitenfeld, and Laxmikant Kale. Performance degradation in the presence of subnormal floating-point values. In *OSIHPA*, 2005.

[47] 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. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security'18*.

[48] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC*.

[49] Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. Two methods for exploiting speculative control flow hijacks. In *USENIX WOOT 19*.

[50] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 2019.

[51] Mozilla. Firefox Bug 1692972 mitigation. https://hg.mozilla.org/releases/mozilla-beta/rev/b129bba64358.

[52] Mozilla. Spectre mitigations for Value type checks - x86 part. https://bugzilla.mozilla.org/show_bug.cgi?id=1433111.

[53] Mozilla. Spider Monkey JS:Value. https://hg.mozilla.org/mozilla-central/file/tip/js/public/Value.h.

[54] Mozilla. SpiderMonkey IonMonkey documentation.

[55] Ken Johnson Microsoft Security Response Center (MSRC). Analysis and mitigation of speculative store bypass. https://msrc-blog.microsoft.com/2018/05/21/analysis-and-mitigation-of-speculative-%store-bypass-cve-2018-3639/, 2019.

[56] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *USENIX Security 20*.

[57] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the ring (s): Side channel attacks on the cpu on-chip ring interconnect are practical. *arXiv preprint arXiv:2103.03443*, 2021.

[58] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. 2021.

[59] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*, May 2021.

[60] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. Sok: In search of lost time: A review of javascript timers in browsers. In *IEEE EuroS&P'21*.

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

[62] Rahul Saxena and John William Phillips. Optimized rounding in underflow handlers, 2001. US Patent 6,219,684.

[63] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS'19*.

[64] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *FC IFCA 17*.

[65] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *ESORICS 19*.

[66] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. 2011.

[67] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[68] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *arXiv*.

[69] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE S & P*.

[70] Google V8. test-jump-table-assembler.cc:220 commit 251fece. https://github.com/v8/.

[71] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE S & P 20*.

[72] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *ACM CCS 2018*.

[73] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX'17*.

[74] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Sgaxe: How sgx fails in practice.

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

[76] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. *arXiv preprint*.

[77] WebKit. Browserbench. https://browserbench.org.

[78] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. 2018.

[79] Pawel Wieczorkiewicz. Intel deep-dive: snoop-assisted L1 Data Sampling.

[80] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks. *arXiv preprint*.

[81] Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *USENIX Security 14*.

[82] Jann Horn Google Project Zero. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2019.

A Reversing Memory Disambiguation

To precisely trigger memory disambiguation mispredictions, it is essential to reverse engineer the predictor and understand how it can be massaged into the desired state. When executing a load operation, the physical addresses of all prior stores must be known to decide whether the load should be forwarded the value from the store buffer (store-to-load forwarding, when the store and the load alias) or served from the memory subsystem (when they do not). Since these dependencies introduce significant bottlenecks, modern processors rely

Listing 4 A function to RE the MD predictor. The first 10 `imul`s used to delay the store address computation create the ideal conditions for mispredictions.

```
st_ld:  ;rdi: store addr, rsi: load addr
%rep 10 ;Trick to delay the store address
imul    rdi, 1
%endrep
mov     DWORD [rdi], 0x42   ;Store
mov     eax, DWORD [rsi]    ;Load
%rep 10 ;Pronounce load timing
imul    eax, 1
%endrep
ret
```

Listing 5 Observing the size and behavior of the per-address saturation counter

```
uint8_t *mem = malloc(0x1000);
//Ensure that saturing counter is set to 0
for(i=0;   i<100; i++) st_ld(mem, mem);
//Make hoisiting possible
for(i=100; i<120; i++) st_ld(mem, mem+64);
//Trigger a memory ordering machine clear
for(i=120; i<130; i++) st_ld(mem, mem)
```

on a memory disambiguation predictor to improve common-case performance. If a load is predicted not to *alias* preceding stores, it can be speculatively executed before the prior stores' addresses are known (i.e., the load is *hoisted*). Otherwise, the load is stalled until aliasing information is available.

Partial reverse engineering of the memory disambiguation unit behavior was presented in [18], based on a (complex) analysis of Intel patent US8549263B2 [44]. In contrast, we present a full reverse engineering effort (including features such as 4k aliasing, flush counter, etc.) entirely based on a simple implementation—`st_ld` function in Listing 4. By surrounding the `st_ld` function with instrumentation code to measure the timing and the number of machine clears, we were able to accurately detect the status of the predictor for every call to `st_ld`.

Our first experiment, illustrated in Listing 5, is designed to observe how many missed load hoisting opportunities are needed to switch the predictor state. As shown in the corresponding plot in Figure 11, after 15 non-aliasing loads, we observed that subsequent `st_ld` invocations are faster due to a correct hoisting prediction. This matches the design suggested in the patent, with the predictor implemented as a 4-bit saturating counter incremented every time the load does not ultimately alias with preceding stores (and reset to 0 otherwise). Load hoisting is predicted only if the counter is saturated. To ensure that the hoisting state is reached, we later scheduled an aliasing load and checked that the load was incorrectly hoisted by observing a machine clear.

According to the Intel patent [44] there are 64 per-address predictors (i.e., saturating counters) and the suggested hashing function simply uses the lowest 6 bits of the instruction pointer of the load. We verified these numbers using the function `st_ld_offset`, which is an exact copy of the `st_ld` function but with a number of `nops` added in the preamble (which we change at every run. The goal is to observe if two unrelated loads in these functions are able to influence each other when varying the number of `nops`. In our tested CPUs, we observed machine clears when two unrelated loads in `st_ld` and `st_ld_offset` are located exactly $256k$ bytes apart in memory ($k \in \mathbb{N}$). We used machine clear observations to detect that the per-address prediction of `st_ld` was affected by the execution of `st_ld_offset`. Our results match the design suggested in the patent, except we observed 256 (rather than 64) predictors hashing the lowest 8 bits of the instruction pointer. With these implementation-specific numbers, an attacker can easily mistrain the predictor of a victim load instruction just with knowledge of its location in memory.

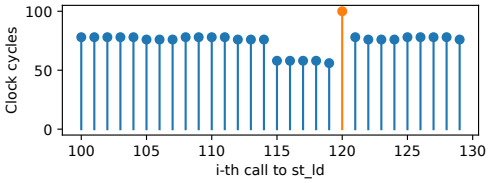


Figure 11: Timing measurement of the experiment in Listing 5. Orange bar: machine clear memory ordering observed

Listing 6 Snippet observing activation of never-hoisting state

```
uint8_t *mem = malloc(0x1000);
for(int i=0; i<10; i++)
    for(int j=0; j<19; j++)
        st_ld(mem, mem+64);
    st_ld(mem, mem);
```

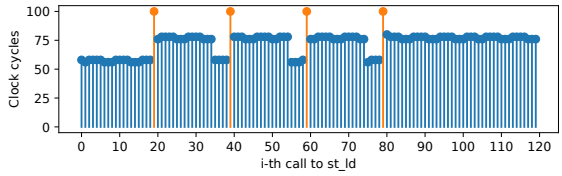


Figure 12: Never-hoisting state. Orange bar: MC observed

One important additional component of the predictor is the presence of a watchdog. A *never-hoisting* global state is used as a fallback to temporarily disable the predictor when the CPU has decided it may be counterproductive. To reverse engineer the behavior, we triggered as many mispredictions as possible to check if hoisting was eventually disabled. The resulting code is illustrated in Listing 6 and the numbers in Figure 12 shows that, after 4 machine clears, the predictor is disabled. Indeed, even after 19 further non-aliasing loads (normally abundantly sufficient to switch to a hoisting state), the execution time of `st_ld` does not decrease.

We also reversed the conditions under which the watchdog is enabled/disabled. The patent suggests the watchdog

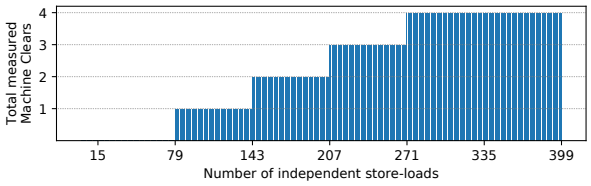


Figure 13: MCs observed after n independent store-loads starting from the never-hoisting state

is enabled when the value of a *flush counter* is smaller than 0, and disabled otherwise. The flush counter is decremented every MD MC and incremented every n correctly hoisted loads. To reverse engineer this behavior, we measure how many MCs can be triggered in a row before the flush counter is decremented to -1 and thus the predictor is disabled. As shown in the figure 13, we never observed more than 4 MCs in a row. This suggests that the flush counter is a 2-bit saturating counter. The machine clear patterns also reveal the flush counter is incremented every 64 correctly hoisted loads. Lastly, since every run starts with the watchdog disabled, our results show that, to switch from a never-hoisting to a predict-hoisting state, 15+64 non-aliasing loads are sufficient. The first 15 loads are necessary to bring the per-address predictor to the hoisting state. The next 64 loads record a *would-be correct prediction* of the per-address predictor. After 64 such (unused) predictions, the flush counter is incremented to leave the never-hoisting state.

Listing 7 Snippet verifying *4k aliasing*-MD unit interaction

```
//Force no-hoisting prediction
for(i=0;  i<10; i++) st_ld(mem+0x1000, mem+0x1000);
for(i=10; i<20; i++) st_ld(mem+0x2000, mem+0x2000);
//Cause 4k aliasing
for(i=20; i<40; i++) st_ld(mem+0x1000, mem+0x2000);
```

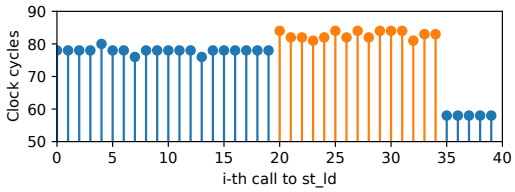


Figure 14: Timing measurements of Listing 7. Orange: increment of `LD_BLOCKS_PARTIAL.ADDRESS_ALIAS`

Finally, we examined the interaction between memory disambiguation and *4k aliasing*. On Intel CPUs, when a store is followed by a load matching its 4KB page offset, the *store-to-load forwarding* (STL) logic forwards the stored value, and, in case of a false match (*4k aliasing*), a few-cycle overhead is needed to re-issue the load [38]. With the help of the performance counter `LD_BLOCKS_PARTIAL.ADDRESS_ALIAS` and the experiment shown in Listing 7, we verified that *4k aliasing* can only happen when a *no-hoist* prediction is made

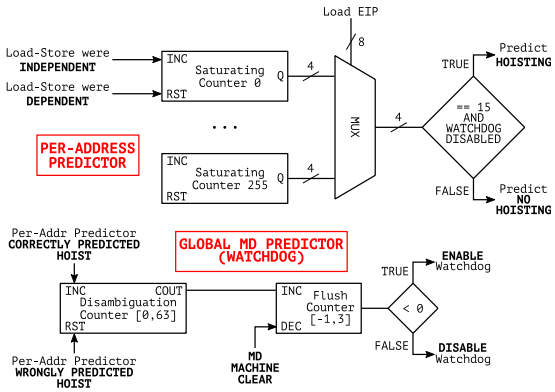


Figure 15: Memory disambiguation unit – simplified view

as shown in Figure 14. Indeed, STL can be performed only if the store-load pair is executed in order. Additionally, Figure 14 shows that *4k aliasing* introduces a slight performance overhead on top of an incorrect no-hoisting guess of the MD predictor. Figure 15 presents a simplified view of the reversed memory disambiguation predictor. In conclusion, an attacker can precisely mistrain the memory disambiguation predictor by satisfying only two requirements: (1) knowing the instruction pointer of the victim load; (2) issuing (in the worst-case scenario) 15+64=79 non-aliasing store-load pairs to train the predictor to the hoisting state.

B Root-Causes Description Table

Acronym	Description
BHT	Branch History Table
BTB	Branch Target Buffer
RSB	Return Stack Buffer
MD	Memory Disabmbiguation Unit
NM	Device Not Available Exception
DE	Divide-by-Zero Exception
UD	Invalid Opcode Exception
GP	General Protection Fault
AC	Alignment Check Exception
SS	Stack Segment Exception
PF	Page Fault
PF - U/S Bit	Page Table Entry User/Supervisor Bit PF
PF - R/W Bit	Page Table Entry Read/Write Bit PF
PF - P Bit	Page Table Entry Present Bit PF
PF - PKU	Page Table Entry Protection Keys PF
BR	Bound Range Exceeded Exception
FP	Floating Point Assist
SMC	Self-Modifying Code
XMC	Cross-Modifying Code
MO	Memory Ordering Principles Violation
MASKMOV	Masked Load/Store Instruction Assist
A/D Bits	Page Table Entry Access/Dirty Bits Assist
TSX	Intel TSX Transaction Abort
UC	Uncachable Memory Assist
PRM	Processor Reserved Memory Assist
HW Interrupts	Hardware Interrupts