Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations

Andrea Mambretti
Northeastern University
Boston, USA
mbr@ccs.neu.edu

Matthias Neugschwandtner
IBM Research - Zurich
Rueschlikon, Switzerland
mneug@iseclab.org

Alessandro Sorniotti
IBM Research - Zurich
Rueschlikon, Switzerland
aso@zurich.ibm.com

Engin Kirda
Northeastern University
Boston, USA
ek@ccs.neu.edu

William Robertson
Northeastern University
Boston, USA
wkr@ccs.neu.edu

Anil Kurmus
IBM Research - Zurich
Rueschlikon, Switzerland
kur@zurich.ibm.com

ABSTRACT

Speculative execution attacks exploit vulnerabilities at a CPU's microarchitectural level, which, until recently, remained hidden below the instruction set architecture, largely undocumented by CPU vendors. New speculative execution attacks are released on a monthly basis, showing how aspects of the so-far unexplored microarchitectural attack surface can be exploited. In this paper, we introduce, Speculator, a new tool to investigate these new microarchitectural attacks and their mitigations, which aims to be the `GDB` of speculative execution. Using speculative execution markers, set of instructions that we found are observable through performance counters during CPU speculation, Speculator can study microarchitectural behavior of single snippets of code, or more complex attacker and victim scenarios (e.g. Branch Target Injection (BTI) attacks). We also present our findings on multiple CPU platforms showing the precision and the flexibility offered by Speculator and its templates.

CCS CONCEPTS

• **Security and privacy → Security in hardware**; *Side-channel analysis and countermeasures*; **Hardware reverse engineering**.

KEYWORDS

hardware reverse engineering, hardware side-channels, hardware security

1 INTRODUCTION

A developer's view of the CPU when writing a low-level program is defined by the CPU's instruction set architecture (ISA). The ISA is a well-defined, stable interface the developer can use to access and change the architectural state of a CPU. The software is in full control over memory, registers, interrupts and I/O. At the same time, the CPU has a lower-level state of its own – the extra-architectural state of the microarchitecture, commonly referred to as the *microarchitectural state*. In general, the ISA provides no direct access to the CPU microarchitecture, allowing the microarchitecture to evolve independently while keeping the programming interface backward compatible. The microarchitecture of a CPU is subject to frequent changes between generations and models, and is different even among vendors of a given ISA. A CPU's microarchitecture typically also implements security controls, such as process isolation.

Recent works [29, 33, 54] have shown how security controls can be bypassed by submitting carefully-crafted inputs at the level of the ISA interface. These attacks exploit undocumented behavior at the microarchitectural level, and have been discovered through reverse engineering and trial-and-error. The full breadth of this class of attacks is not entirely understood, owing to the fact that details about the microarchitectural level of modern commercial CPUs are not publicly available. The research community cannot provide complete answers to questions about the existence of new attacks and the effectiveness of defenses.

More precisely, we identify two important related requirements: (1) When developing new attacks, it is often required to analyze and debug parts of the proof-of-concept code easily. For memory corruption, this would be achieved with a debugger. An equivalent for speculative execution attacks, that inspects microarchitectural state directly, is needed. (2) When testing speculative execution mitigations, the current option is either to attempt a proof-of-concept attack, or to trust the CPU flags and kernel configuration that are provided. A more granular testing tool that directly inspects microarchitectural state would be beneficial to gain confidence in the mitigations being properly implemented and enabled.

In this paper, we propose a tool, Speculator, with these two requirements in mind. Speculator records or infers microarchitectural behavior by using performance counters, supports incremental analysis (evolution of microarchitectural state over a code snippet), runs on both Intel and AMD CPUs, and enables concurrent execution (interaction of two threads in an SMT environment).

Our paper makes the following contributions:

- A new performance-counter-based method and tool, Speculator, to aid in designing attacks and mitigations.
- Insights into microarchitectural behavior relevant to attacks and defenses: we successfully verify the return stack buffer size, that nested speculative execution works, that speculation does not span across system calls and that `clflush` has no effect during speculation. We also measure the window size for indirect branches, indirect control flow transfers and store to load forwards. Finally, we document the effects of page permissions, memory protection extension and special instructions (e.g. lfence) on speculative execution.
- Examples of using Speculator against attacks and mitigations.

2 BACKGROUND

Speculative execution attacks (SEAs) exploit a new class of vulnerabilities, targeting a particular microarchitectural CPU design with specially crafted software. These attacks leverage known attack vectors such as side channels, but go much further by combining them with vulnerabilities at the microarchitectural level. Numerous variants of SEAs have been disclosed since the beginning of 2018. In this section, we propose a general definition and analysis of SEAs with the aim of clearly distinguishing SEA variants in order to motivate and guide the analysis of new attacks and defenses in this area.

Before delving into the dissection of SEAs, we need to distinguish SEAs from the more general category of out-of-order execution attacks. Spectre v1 and v2 [29, 54] are the first discovered SEAs, with Spectre v1.1 [28], Spectre v4 [22], NetSpectre [42] and Netspectre-AVX being follow-ups. In contrast, attacks such as Meltdown [33], Spectre v3a [7], Foreshadow [47] and Foreshadow-NG [51] do not rely on speculative execution behavior, and may be classified in the more general category of out-of-order execution attacks.

2.1 Attack scenarios, Privilege boundaries

SEAs, much like side channel attacks, can be performed in a variety of scenarios involving one victim and one attacker thread. The notion of *thread* here is in the general, hardware-related sense (e.g. VMM thread, guest thread, (un)-sandboxed thread, or user/kernel thread). The notion of *privilege level* here is also in the general sense: not necessarily CPU privilege level related. These attacker and victim threads run with different privileges, with the attacker thread typically running with a lower privilege. There can also be scenarios where both threads are at the same privilege level, but have access to different data. In all cases, however, a boundary separating attacker and victim contexts resides between the two threads.

2.2 SEA Phases

SEAs can be decomposed into the following five distinct phases:

❶ Prepare side channel: In this phase, the CPU performs operations that will increase the chances of the attack succeeding. For instance, the attacker can prime caches to prepare for a prime-and-probe [45] cache side channel measurement, make sure important target data is flushed, or ensure that the attacking thread and victim thread are co-located.

❷ Prepare speculative execution: In this phase, the CPU executes code that will allow speculative execution to start. This is code that is typically executed within the context of the victim.

❸ Speculative execution start: In this phase, the CPU executes an instruction whose outcome decides the next instruction to be executed, such as a conditional branch instruction. Between the time window where this instruction is issued and when it is retired, modern CPUs guess the outcome of the branch to avoid stalling the pipeline, and execute code speculatively. This is known as speculative execution [31].

❹ Speculative execution, side channel send: In this phase, the CPU executes (but does not retire) instructions that will result in a micro-architectural state change.

❺ Side channel receive: In this phase, the CPU executes instructions that transform the micro-architectural state change that occurred in the previous step into an architectural state change.

2.3 Privilege boundaries and attack impact

The core element that turns speculative execution into an attack is the breach of a privilege boundary that is established through hardware isolation support by the CPU. These privilege boundaries typically aim to provide confidentiality and integrity of the data residing within the boundary (i.e. preventing data from being read or modified directly from outside the boundary). All accesses to such data are mediated by code running within the privilege boundary, and that code may only be invoked from a lower privilege through well-defined entry points.

In the case of currently known SEAs, the attacker's aim is limited to breaching confidentiality of data residing beyond the privilege boundary by either accessing arbitrary data or leaking specific metadata, such as pointer values, of the running program. For instance, privilege boundaries that can be bypassed by some known SEAs are:

- kernel vs. user-mode code
- hardware enclave (SGX) vs. user-mode or kernel-mode code
- sandboxed code in the same process, for example JavaScript JIT code
- processes-to-process boundary
- remote node to local node boundary

We note that code at each SEA phase previously described can potentially be run either in the higher privileged mode (victim-provided code) or lower privileged one (attacker-provided code).

3 SPECULATOR

Speculative execution is not well-documented compared to other features of modern CPUs. Being part of the microarchitecture, its implementation details are hidden behind the ISA and subject to optimization, which manufacturers keep to themselves.

However, understanding the internals of speculative execution is key to comprehending the limits of Speculative Execution Attacks (SEAs), and to designing adequate mitigations and defenses against SEAs. For this reason, we have designed and implemented Speculator, a tool whose purpose is to reverse-engineer the behavior of different CPUs in order to build a deeper understanding of speculative execution. Speculator aggregates the relevant sources of information available to an observer of speculative execution, chief among them CPU performance counters and model-specific registers, so that the behavior of different code snippets can be observed from a speculative execution standpoint. In this section, we describe the design and implementation of Speculator.

3.1 Performance Monitor Capabilities

Modern CPUs provide relevant information through the performance counter interface. This interface is offered by most manufacturers, and it exposes a set of registers (some fixed and some programmable) that can be used to retrieve information on various aspects of the execution. Through these registers, counters for events or duration related to microarchitectural state changes such

as cache accesses, retired instructions, and mispredicted branches, are made available to the developer. Events are manufacturer- and architecture-specific. This interface was originally made available to provide a method for developers to improve the performance of their code. The interface is typically used as follows: through a setup step, developers can choose which events will be measured by programmable counters out of a wide set of supported ones. Measurements can be started and stopped programmatically in order to carefully control the events of which precise sequence of instructions is being measured. Setting up, starting, and stopping measurements often requires supervisor mode (ring 0 in x86 nomenclature) instructions, whereas accessing counters is usually available in user mode.

Speculator builds on top of performance counters to observe the nature and effects of speculative execution. One challenge with this approach is that the performance counters interface was not designed with this objective in mind. One of the contributions of this paper is the identification of effective ways of using the interface, and a useful set of counters to accurately infer the behavior of speculative execution.

3.2 Objectives

The main objective of Speculator is to accurately measure microarchitectural state attributes associated to the speculative portion of the execution of user-supplied snippets of code. Accuracy refers to the degree with which the tool is capable of isolating the changes to the microarchitectural state caused by the snippet being analyzed from that of the tool itself and the rest of the system (e.g. the OS or other processes). An incomplete list of Speculator observables are (1) which parts of the snippet are speculatively executed, (2) what causes speculative execution to start and stop, (3) what parameters affect the amount of speculative execution, (4) how do specific instructions affect the behavior of speculative execution, (5) which security boundaries are effective in the prevention of speculative execution, and (6) how consistently CPUs behave within the same architecture and across architectures and vendors. The creation of a new tool is justified because none of the existing ones, such as perf_events [14] or Likwid [41], provide the required information with sufficient accuracy.

More precisely, perf_events has two modes of operations, sampling and counting. During sampling, there is no way to have precise quantitative information about code execution, and therefore it is not suitable for our purpose. When evaluating perf_events' counting mode, we experienced for very small snippets a certain level of overhead (in the order of 500 μops). This overhead was caused by the perf_event design decision of integrating all its operations (e.g. start counters, stop counters) in the kernel. Since the test snippets are 20-30 instructions long on average, this overhead completely prevents inferring any kind of relevant behavior.

Likwid operates instead in user space just as Speculator, instrumenting the counters through the MSR register. However, its design only allows system-wide measurements and does not provide the same flexibility of handling the counter as the snippet progresses in its execution.

We also considered other tools and libraries such as Oprofile [32], Perfmon2 [16], Perfctl [39], and PAPI [43]. Unfortunately, all of these possess either the same issues of measure inaccuracy or lack of flexibility, or otherwise are outdated and unmaintained. Performance comparisons among some of these interfaces are provided by Zaparanuks et al. [53] and Weaver [50].

Another Speculator objective is to provide tooling for the generation and manipulation of code snippets. The ability to inspect individual snippets and snippet groups during speculative execution allows the user to focus on combinations of instructions that are relevant for specific use-cases. Additionally, support for multiple platforms enables the inference of generalizable facts about speculative execution.

3.3 Design and Implementation

Figure 1 describes the architecture of Speculator and its three main components: a pre-processing unit, a runtime unit called the Monitor, and a post-processing unit.

The task of the pre-processing unit is to compile the provided input into the appropriate execution format, and to introduce the instrumentation required by the performance monitor interface to be able to observe the value of the selected set of hardware counters. Input can be provided as a snippet of C or assembly code, or as a template for the generation of code snippets. Code snippets are generated from templates in an incremental fashion, resulting in the output of multiple snippets with an increasing number of instructions taken from a pre-compiled JSON list. Each instruction is inserted by the Speculator snippet generator in the specific location defined in the source template (Step 1 in Figure 1). The introduction of such "incremental" snippets is justified by the fact that the addition of a single assembly instruction may trigger optimizations that – while preserving the expected program semantics – alter the behavior of the CPU at a microarchitectural level and affect the nature of speculative execution. Having incremental snippets helps to verify when optimizations are triggered and take them into account during the analysis of the results.

After the generation of the executable (also referred to as the test application), the Speculator runtime is invoked on each of the generated outputs (Step 3). To ensure that the Monitor does not perturb the measurements, the process executing the snippet and the monitor are pinned on different cores. The Monitor is responsible to configure the counters on the core used by the test application (Step 2). As previously mentioned, there are many programmable counters that can be used so we provide a configuration file that can be loaded into Speculator to easily switch among them.

Once the Monitor has set up the environment, it loads and executes the snippet in a separate process, and waits for it to complete (Step 3). The test application prologue and epilogue will interact with the environment created by the Monitor, resetting, starting and stopping the counters as needed. The counters related to the core where the test application runs are stopped by the test application just before termination. When the test application terminates, the Monitor will be signaled by the Operating System. At this point, the Monitor can retrieve the values of the counters from the core where the test application runs (Step 4) and store them in a result file (Step 5). The Monitor can be configured to run a specific test N times. In this case, the result file will contain the values of each run.

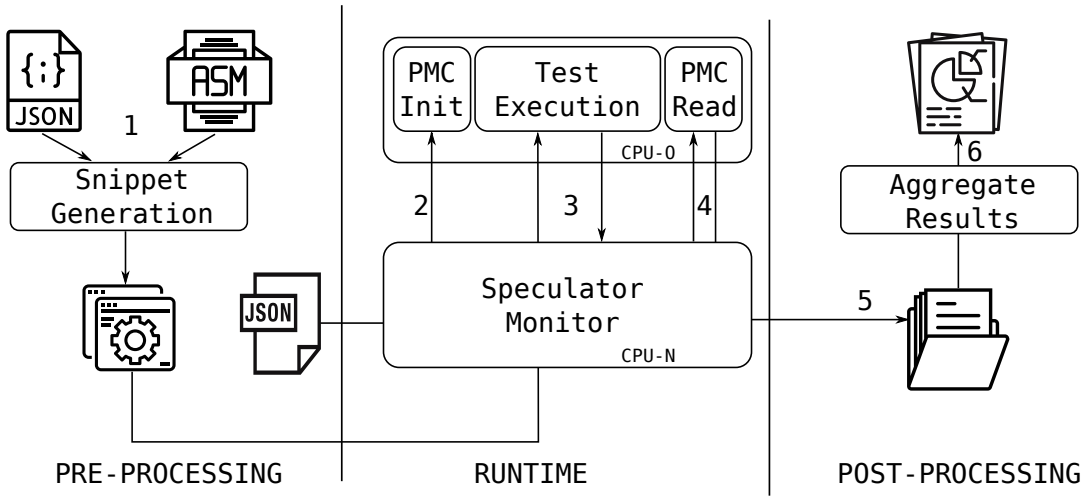


Figure 1: The architecture of Speculator. A template with the speculative execution trigger and a list of instructions to be speculatively executed are the input to the code generation. The code snippets are run repeatedly under supervision of the speculator monitor, which captures the event specified in the configuration file. Finally, the measurements are post-processed to present a final report on speculative execution behavior.

In some cases, it might be required to run two processes in an attacker and victim scenario. In this case, Speculator is able to run two tests in a co-located manner to analyze the effects of a process influencing another, like in the case of Spectre v2 or DoubleBTI [36]. Speculator collects different counters for the attacker and the victim. Under this configuration, Speculator performs no synchronization between the two processes.

Once the tests results are collected from the Monitor, they are handed to the post-processing unit (Step 6). This unit aggregates the results from multiple runs by computing statistics (e.g. mean and standard deviation) and by removing clear outliers.

3.4 Triggering Speculative Execution

The Speculator user supplies as input a code snippet to determine how the CPU behaves when speculative execution takes place. We note that in the absence of branch misprediction, instructions that are speculatively executed will eventually retire and there should be no undesired microarchitectural side-effects. The more interesting case for the Speculator user is a snippet containing a branch, or other speculative execution trigger that the CPU does not predict accurately, leading to the speculative execution of instructions that will not retire. In this scenario, Speculator helps the user detect which instructions the CPU executed and how they influenced the microarchitectural state.

To automate the generation of test cases, Speculator provides the user with a series of templates that can be used to reproduce the various speculation triggers. For instance, Speculator contains templates to study Branch Target Injection (BTI) cases including attacker and victim, or branch-based templates to study particular series of instructions, or templates that causes `ret` instructions to be speculated like in the Return Stack Buffer case, and so on.

An example using a common branch as trigger is described in Figure 2. The template is used as follows: the user supplies a snippet, expecting i) it to be speculatively executed, ii) that none of its instructions will retire, and iii) that Speculator will report counters relating to its execution. To achieve this, the template prefixes the snippet supplied by the user with a branch instruction. The template begins with a setup step that trains the branch predictor not to take that branch. After the branch predictor is trained, the program state is set to require the branch to be taken to ensure that the snippet will be speculatively executed and that none of its instructions will retire. The template then starts the performance counters that were previously setup by the Monitor and executes the branch, after which it stops performance counters. In order to prolong or shorten the speculative execution of the user snippet, the condition variable of the branch can be placed in registers or memory. On the microarchitectural level, a variable placed in memory can also be cached in one of the levels of the cache hierarchy.

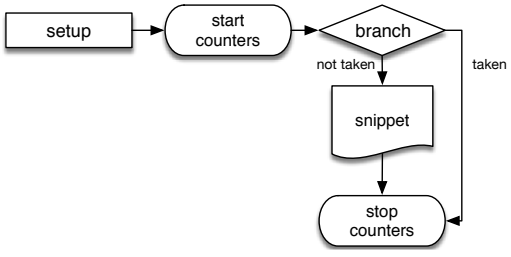


Figure 2: Flow chart of one of the experiment template used in Speculator. The setup code brings the branch predictor in a specific state that will cause the later branch to mispredict and speculatively execute the code snippet consisting of the instructions. The speculative execution of the instructions is measured by the PMC infrastructure, which is triggered by the corresponding start/stop instructions indicated in the flow chart.

3.5 Speculative Execution Markers

In the context of Speculator, we are mostly interested in determining the behavior of the CPU when instructions that are speculatively executed do not retire. A first natural question is whether non-retired instructions were speculatively executed at all and, if so, how many of them. An accurate detection of these events is (perhaps surprisingly) not trivial. Indeed, the CPU strives to undo most observable architectural side-effects from non-retired speculatively executed instructions. However, as we know from the Spectre and Meltdown works [29, 33], not all side effects are undone. One possible approach to detect non-retired speculative execution would be to rely on the side-channels exploited in these works. This approach has several shortcomings: it is noisy, i.e., it has a relatively low single-run detection accuracy, it is costly to setup and read, and it requires otherwise unnecessary changes to program observables.

A more effective approach is based on markers of speculative execution, that is, special instructions or sequences thereof (which we will refer to as markers) that are detectable by performance counters even when they do not retire. The approach requires appending the marker to the snippet which is fed as input to Speculator, and ensuring that there is no other occurrence of the marker in the snippet. If Speculator detects the marker, the detection can be used as proof that the CPU executed the snippet.

The choice of which markers to use is manufacturer- and architecture-specific, given that not all CPUs expose the same set of counters. In general, the marker must cause a microarchitectural event that is detectable by a performance counter irrespective of its retired status. For example, counters that measure *issued* or *executed* instructions of a specific type irrespective of their retired status constitute a good marker. The selection of which counter to use on a given architecture requires manual inspection of the CPU architecture programmer's manual. In what follows, we report our findings on the available markers for Intel processors:

`UOPS_EXECUTED.CORE/THREAD` counts the number of μops executed by the CPU. It can be used to report the exact number of μops that were executed out of the user-supplied snippet by subtracting the number of μops that retire in the template (the branch and the instrumentation to stop performance counters) from the output value of the counter. This counter is subject to μ-fusion of instructions and does not count instructions that do not require execution such as `NOP`. An exception to that rule is `FNOP`, which is tracked by this counter as well.

`UOPS_ISSUED.SINGLE_MUL` belongs to a group of counters triggered only by a specific set of instructions. This counter is fired whenever a single-precision floating-point instruction that operates on the `XMM` register is issued. This means that such an operation can be inserted at the end of the user-supplied snippet to verify whether this counter is incremented or not. This counter has been dropped by Intel on most recent CPUs (e.g. Skylake) and therefore its usage is limited across platforms.

Similarly to `UOPS_ISSUED.SINGLE_MUL`, `UOPS_ISSUED.SLOW_LEA` is triggered by only a specific set of instructions. It counts LEA instructions with three source operands (e.g. `lea rax, [array+rax*2]`). Unfortunately, certain operations such as `clflush` are considered by the CPU as `SLOW_LEA` operations, so extra care must be taken to subtract any number of those present outside of the user-supplied snippet.

`LD_BLOCKS.STORE_FORWARD` is incremented for each store forward that result in a failure. An example of a sequence that triggers this kind of situation is shown in Listing 1.

```
1 mov DWORD[array], eax
2 mov DWORD[array+4], edx
3 movq xmm0, QWORD[array]
```

Listing 1: Failed store forward example

The following markers are available on the AMD Zen architecture:

`DIV_OP_COUNT`, counting the number of executed `div` instructions.

`NUMBER_ OF_ MOVE_ ELIMINATION_ AND_ SCALAR_ OP_OPTIMIZATION`, like `LD_BLOCKS.STORE_FORWARD`, does not track the execution of an instruction, but rather the effect of a certain instruction sequence. In this case, it tracks in how many cases move elimination was successful.

Architecture	CPU	Design
Intel Haswell	i5-4300U	tock
Intel Broadwell	i5-5250U	tick
Intel Skylake	i7-6700K	tock
Intel Kaby Lake	i7-8650U	optimization
Intel Coffee Lake	i7-8559U	optimization
AMD Zen	Ryzen 1700	

Table 1: The CPUs per architecture we use Speculator on. While Haswell and Skylake are new designs – "tocks" in Intel nomenclature – Broadwell is a "tick", a die-shrink of Haswell. Kaby and Coffee Lake are instead optimized versions of Skylake design within the same die size

4 USING SPECULATOR: DISSECTING THE MICROARCHITECTURAL WORLD

Using Speculator, we explore the microarchitectural behavior of modern CPUs. Our goal is twofold: we aim to investigate several speculative execution properties, as well as test new PoC attacks and available mitigations in a deterministic manner using the speculative execution markers introduced in Section 3.5.

The results that we uncover are applicable to previously discovered and new attacks, and are also of independent interest. Since some of our findings are hardware-dependent, we also show the differences based on the underlying CPU architecture (Table 1).

4.1 Return Stack Buffer Size

The first set of experiments measures the size of the *Return Stack Buffer (RSB)*. The RSB is an internal buffer used by the CPU to predict where a `ret` instruction is returning to. Koruyeh et al [30] and Maisuradze et al [35] show how this buffer can be misused to perform speculative execution attacks. We start with the RSB since information on its size is available and can be used to validate the accuracy of Speculator.

To perform the measurement, we design a test template similar to the one presented in [30]. The test performs a `call` to a victim function. Whenever the CPU executes a `call` instruction, it pushes

the expected return address (the instruction after `call`) on the application stack (architecturally) and in the RSB (microarchitecturally). The victim's code further changes its return address to an exit routine by manually overwriting the stack. This way, the code at the original return address is only speculatively executed since at the microarchitectural level, the first entry in the RSB is popped and execution (speculatively) continues at that address. In order to be able to detect whether speculative execution takes place, a marker is inserted at this target.

Based on the described template, we generate a series of snippets that, between the `call` and `ret`, have a call to a `filler` function that contains an increasing number of nested calls. For each of the nested calls, an entry is added to the RSB. When the nested call stack depth is bigger than the RSB size, the RSB loses the oldest entries. In this case, once the CPU speculates the last `ret`, it has nothing to pop from the RSB because the previous nested call/ret consumed all the available entries. In that case, we expect the CPU not to be able to speculatively execute our marker.

We report in Figure 3 and Figure 4 our results for Intel Kaby Lake and AMD Ryzen. For Intel Kaby Lake, we observe that the marker is observable up to 14 nested calls. To count the slots available in the RSB, we need to also consider the additional call to the `filler` function that contains the nested calls. This results in a total of 16 entries in the RSB, which matches the value reported by Intel for the Kaby Lake RSB size. Interestingly, after 15 nested calls the number of mispredicted branches increases almost linearly, by one for each nested call added. This indicates that a second predictor is used as fallback once the RSB cannot provide any more values.

Figure 4 shows the results for AMD Ryzen. After 30 nested calls, we observe the marker hit to transition between 1 and 0.25. As before we need to account for the call to `filler`. The result for the AMD Ryzen RSB size is 31. Our result differs from the nominal value we expected from the manufacturer specification, which is 32. With further research into the optimization manual [9], we found that one entry is actually reserved for "pointer logic simplification". Therefore, the observed 31 entries is correct. On AMD, after the Return Address Stack (RAS) (the RSB in AMD nomenclature) is emptied, we still observe a correct prediction 25% of the time and not 0 as seen for Intel. This implies that the second predictor used can still predict correctly 25% of the time in this particular setup.

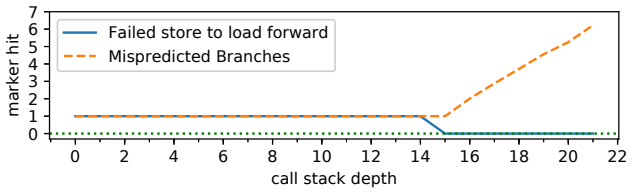


Figure 3: Return Stack Buffer test on Kabylake.

4.2 Nesting Speculative Execution

An undocumented corner case that might affect the construction of attacks is when speculative execution encounters conditional branches in its path. The questions we try to answer with this

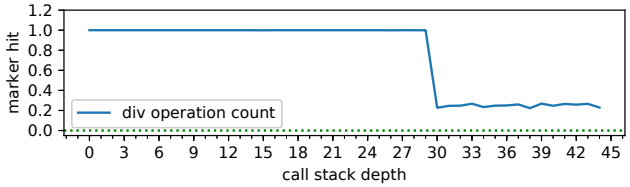


Figure 4: Return Stack Buffer test on AMD Ryzen.

experiment are "How is the speculative execution window affected by nested branches?". And "What is the overall behavior of the CPU when nested branches are speculated?".

We use Speculator to evaluate the case of nested branches. This experiment has multiple potential outcomes: given two nested branches, an outer and an inner one, either *i*) the inner branch is not speculatively executed until the branch condition on the outer branch is resolved, or *ii*) speculative execution continues to the inner branch and beyond. In the second case, we are interested in the speculative execution behavior if the inner branch is resolved while the result of the outer one is still pending.

We design our experiment with three nested conditional branches, outermost to innermost, with the branch conditions being independent of one another. The conditions are set up with decreasing complexity, such that the outermost will take longest to resolve. We achieve this by involving an uncached value that is subject to multiple expensive operations (`divs`) in the outermost branch condition, a simple uncached value in the middle branch condition, and a cached value in the innermost branch condition. As usual, we train the branch predictor for all branches in the setup phase such that it is going to mispredict all targets in the measurement phase. To evaluate which code paths are (speculatively) executed, we repeat the experiment multiple times with marker instructions placed in the opposite branch target paths.

We performed this experiment on both Broadwell and Skylake, yielding identical results: in both cases, nested speculative execution takes place, i.e. speculative execution continues along the trained branch targets for all branches. Second, if a nested branch condition is resolved before its parent branch and a misprediction has occurred, speculative execution picks up the opposite branch target. If a parent branch is resolved, all mispredicted code paths, including nested speculative execution, is canceled.

4.3 Speculative execution across system calls

An interesting case to analyze for new attacks is how speculative execution behaves in case the attack spans between multiple privilege boundaries. The question we try to answer here is: "Does speculative execution continue through instructions such as `syscall` and `vmall`?"

We thus investigate whether speculative execution continues across the context switch from user- to kernel mode. To this end we design a simple test scenario, where the speculatively executed snippet issues a system call. For the system call itself we picked `sys_getppid` because of its low complexity – an execution only amounts to 47 instructions. We use the counter for executed μops

and tune it to capture either just μops executed in user mode or kernel mode.

We performed the experiment on the Broadwell and Skylake microarchitectures with identical results:

- The number of μops executed in user mode corresponds to the instructions before the system call and does not increase with additional instructions added after the system call.
- The number of μops executed in kernel mode does not increase compared to a baseline measurement taken without speculative execution of the code snippet.

We conclude that a system call effectively stops speculative execution after the system call returns from kernel mode. We further conclude that a speculative execution attack across the system call boundary is not feasible on the tested Intel CPUs.

```
1     setup
2  .loop:
3     clflush[counter]
4     clflush[var]
5     lfence
6
7     mov eax, DWORD[var]    ;cached version
8     lfence                 ;only
9
10    start_counter
11
12    cmp 12, DWORD[counter]
13    je .else
14
15    clflush[var]
16    lfence
17
18 .else:
19    mov eax, DWORD[var]    ;final load
20    lfence
21
22    stop_counter
23
24    inc DWORD[counter]
25    cmp DWORD[counter], 13
26    jl loop
```

Listing 2: Clflush test snippet structure

4.4 Flushing the Cache

The x86 instruction set provides a convenient, dedicated instruction to cause the CPU to flush the cache line indicated by a memory address from all caches, `clflush`. It is very useful in settings where an attacker can execute assembly instructions, as it allows easy eviction of data from the cache.

We use Speculator to investigate how `clflush` behaves when executed speculatively. To this end, we create a snippet that first flushes the cache line corresponding to a value stored in memory and then loads the value. This is shown at line 4 and line 19 respectively in Listing 2. We perform two runs, one where the setup code warms up the cache by loading the value from memory (line 7) and one where the value is left uncached. In both tests, within the speculated sequence, we place a `clflush` followed by an `lfence` instruction to stop the speculation, making sure that the final load is not executed during speculation as well (line 15). We measure the execution cycles on both runs, which shows a difference of over 160 clock cycles between the two settings. This is a clear indication that while `clflush` is speculatively executed, it does not affect the cache until retired. Thus, during a speculative execution attack, the attacker cannot extend the speculation window using providing in its code a `clflush` on the speculation starter variable.

Another result we draw from this experiment is that, to make sure `clflush` is effective, it needs to be combined with an instruction that stops speculative execution, such as `lfence`.

4.5 Speculation window size

The speculation window size is determined by the clock cycles that it takes until a speculation trigger is resolved. In this section, we provide our measurements of the speculation window for the different triggers used in the Spectre v1, v2, and v4 attacks. To measure clock cycles we use the facilities provided by the PMC of the respective platform: on Intel, a predefined counter tracks elapsed clock cycles according to the same settings as the configurable counters; on AMD, the APERF counter tracks elapsed clock cycles in general.

The theoretical upper limit of instructions that can be executed during speculative execution is given by the size of the reorder buffer, which we evaluated in Section A.1. In practice, it is also limited by the execution ports and units available for executing those transactions. Thus, we also investigate instruction sequences that do not lead to a bottleneck on those resources during speculative execution.

Conditional branches. Conditional branches are the speculative execution triggers used in Spectre v1 to check for an out-of-bounds access to an array. The speculation window size depends on how fast the CPU determines that the actual branch target differs from the information provided by the branch target buffer. We place the conditional value that determines the actual branch target in different locations and involve it in additional computation to investigate how this affects the size of the speculation window. As a baseline, we measure how long the execution of the additional instructions takes. We then measure how long the execution of the instructions together with the conditional branch takes. The placement of the variable and the additional instructions on it affect the time it takes the conditional branch to retire. All measurements are performed a thousand times. Note that controlling the performance counters involves a system call. Since system calls stop speculation, we can only measure how long the retirement of an instruction sequence takes.

As described by Agner Fog in [17], the APERF interface offered by AMD Ryzen for clock cycles requires careful handling due to its scaling with the CPU frequency. Hence, since the measurement technique differs between the two CPU vendors for this particular test, results for Intel and AMD might not be directly comparable. However, while the post-processing of this dataset is different, the test methodology used to gather it is the same.

Table 2 shows the results of this experiment. We see that complex instructions such as `div`, which translates to multiple μops, widen the speculation window. The same is true for a cache miss, when the CPU needs to fetch the data from main memory.

At the same time, access to cached memory contributes little to the speculation window compared to a register access. Measuring a range from four to twelve cycles, the results for Broadwell and Skylake are in accordance with Intel's performance analysis guide [1]

which states four cycles as the average for an access to L1 and ten cycles for L2.

On AMD, we see even less impact between register and cached accesses. In addition, adding a complex instruction on top of an access has a negligible effect on the speculation window size.

Conditional branch	Broadwell	Skylake	Zen
Register access	14	16	7
Access to cached memory	19	17	9
Access to uncached memory	144	280	321
Mul with register	19	19	2
Mul with cached memory	33	33	8
Mul with uncached memory	154	290	362
Div with register	35	41	17
Div with cached memory	34	39	30
Div with uncached memory	164	306	353

Table 2: Speculation window of a conditional branch depending on the type of instructions needed to resolve the branch as well as the placement of the value involved in the condition, measured in cycles.

Indirect control flow transfer. Indirect control flow transfers are the speculative execution triggers used in Spectre v2. The speculation window size depends on how fast the CPU determines that the target in the branch history buffer does not match the actual target. Table 3 shows the speculation window sizes depending on the location of the indirect branch target.

Store to load forwarding. Modern CPU designs feature store and load queues, which capture the effects and dependencies of corresponding load and store operations before the data is even written to or read from the cache. This infrastructure allows for efficient store to load forwarding: if an instruction writes to a certain memory address and a following instruction reads from that very address, the CPU can leverage the result of the first instruction, which is written to the store queue, for executing the second instruction. This avoids unnecessarily stalling the execution of the second instruction until the first is retired. In a recent attack, this behavior has been used for a "speculative buffer overflow" [28].

We are interested in the behavior a failed store to load forwarding causes. In this case, we deviate from our default Speculator template and remove the branch instruction. Instead, we create a snippet with a data dependency that is not detected by the CPU in a combination with a sequence of store and load operations that triggers store-to-load forwarding.

Indirect branch target location	Broadwell	Skylake	Zen
Register	28	22	24
Cached memory	41	34	35
Uncached memory	154	303	301

Table 3: Speculation window of an indirect control flow transfer, measured in cycles. The speculation window size depends on where the target of the indirect control flow transfer is stored.

Running the snippet in Speculator reveals that store-to-load forwarding fails and the load instruction is in fact executed twice. This means that a failed store-to-load forwarding also creates a situation similar to speculative execution results being discarded because of a mispredicted branch, although it provides a significantly smaller speculation window.

Spectre v4 (a speculative store bypass) makes use of speculative execution through store-to-load forwarding. For this trigger we measure a speculation window of 55 cycles on average on Broadwell. We also measure the speculatively executed instructions using `FNOP`, which provides us with an upper bound for the speculation window in terms of instructions. We measure an average of 15 μops with a maximum of 23 μops (Figure 5).

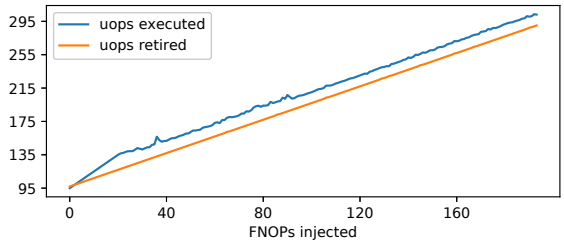


Figure 5: Speculation window of a store-to-load forward failure, measured in executed `FNOP`s on Broadwell.

Max speculation with optimized instruction sequence. During our experiments, we observed multiple situations in which the CPU back-end stalled. For instance, the CPU could stall due to exhaustion of execution units for a certain operation (e.g. `MOV`, `MUL`) or, for instance, data dependencies of multiple operations where one or more data loads caused cache misses. In a hypothetical scenario, we wanted to verify how many non-`NOP` executed μops the CPU speculates within the maximum time window (e.g. access to uncached memory in combination with a `DIV` instruction). Based on the layout of the back-end of our Broadwell CPU under test, to the best of our abilities, we crafted an optimized sequence of instructions to account for the delay of each operation and the available execution unit. Our tests show that the maximum number of non-trivial speculated instructions we could achieve was 160, with 187 being the maximum for `FNOP`.

4.6 Stopping Speculative Execution

Many instruction set architectures feature an instruction that stops speculative execution in the sense that no following instruction is speculatively executed. On x86 (and x86_64), one such instruction is `lfence`, short for "load fence", the name reflecting its initial purpose of serializing all memory load operations issued prior to this instruction. In addition to this behavior, it also works as a barrier for speculative execution: the operational description in Intel's manual [5] specifies that `lfence` waits on following instructions until preceding instructions complete.

We verify this behavior using Speculator by creating a snippet with an `lfence` instruction followed by an increasing sequence of regular instructions. As expected, the counter for executed μops

remains constant among the test runs irrespective of the number of instructions following `lfence`.

4.7 Executable Page Permission

Memory page permissions control access to memory regions at page-level granularity. As we have seen with Meltdown and Foreshadow, such permission checks might be lazily evaluated after an instruction is already executed, but before it is retired. Related work has so far focused on data read or write access to memory pages.

In this test we focus on whether execute permissions set by the NX bit are enforced. The NX bit is part of a hardware extension introduced by modern processors to mitigate stack-based code injection exploits, among others. If the control flow of a program is diverted to a page without execute permissions, the processor will trap into the kernel to handle the fault. This raises the question of whether during speculative execution it is possible to execute instructions from a page without such a permission set.

Our corresponding experiment sets up a branch misprediction with a following control flow transfer to a non-executable memory region, to test whether instructions in it are (speculatively) executed. We ensure that the data from the page is in the L2 cache during speculative execution and the addresses are in the TLB. The result of the experiment is that the execute page table permission is honored during speculative execution by all architectures we examined. This is even true if an instruction spans over two pages: it will not be executed if the second page is set non-executable.

4.8 Memory Protection Extensions

Instead of performing bounds checks purely in software, Intel's MPX instruction set extension [40] available on the Skylake platform provides hardware support for both efficiently keeping track of bounds information associated with pointers and corresponding spatial memory checks before dereferencing pointers. Pointer bounds information is stored in memory and loaded to dedicated registers before it can be used to check the upper bound using the `bndcu` and the lower bound using the `bndcl` instruction. If a bound check fails, a `#BR` exception is raised and the CPU traps into the kernel.

We used SPECULATOR to measure if and how much code following a bounds check instruction is speculatively executed. The setup executes the regular code path without the bounds violation for ten iterations and then fails on a `bndcu` twice. To measure the speculative execution window size, we first used an increasing run of `NOP`s in conjunction with a terminating slow `LEA` marker instruction. In this experiment, we measured that we speculatively execute the marker instruction for a sled of up to 122 `NOP`s. In our second experiment, we used `FNOP` instead of regular `NOP`, which is tracked by the `UOPS_EXECUTED` counter. As is shown in Figure 6a, in this case, the number of executed μops increases up to a sled of 22 `FNOP`s and remains constant beyond.

4.9 Issued vs. Executed μops

Some of the counters that we adopt as markers (e.g. `UOPS_ISSUED.SINGLE_MUL`, `UOPS_ISSUED.SLOW_LEA`) count the number of μops that are *issued*, as opposed to *executed*. Since issued μops are not necessarily executed, as is the case for the `NOP` instruction, we performed

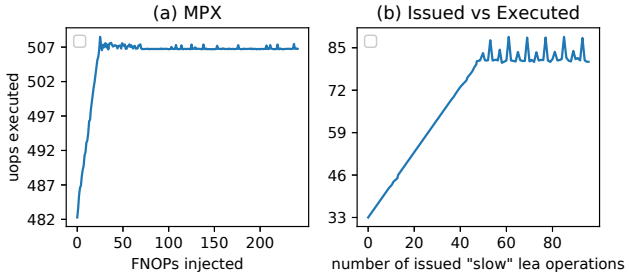


Figure 6: a) Speculative execution after an MPX bounds violation. b) Performance counter numbers for an increasing number of speculatively executed relative load instructions. The graph shows that the number of issued instructions corresponds to the number of executed instructions, justifying the use of such instructions as markers.

a dedicated experiment to verify whether they are also executed. We use the template introduced in Section 3.4 and generate tests where the code snippet just contains an increasing number of `RIP`-relative load instructions. As Figure 6b demonstrates, the number of executed μops increases at the same rate as the counter for slow load effective address instructions, which are load μops with three sources. This result confirms that the instruction is not only issued: its speculative execution does takes place. We obtain similar results for other markers.

5 USING SPECULATOR: ANALYZING ATTACKS AND MITIGATIONS

We also use SPECULATOR to investigate new techniques to exploit speculative execution attacks. On one hand, we can use SPECULATOR to perform measurements on snippet of code to verify their behavior during speculation and verify that an attack might be feasible through those instructions (An example is described in the Annex B).

On the other hand, some of the attacks require two threads interacting with each other through a shared element such as the cache, the branch predictor or the RSB. For instance, during a Branch Target Injection (BTI) generally there is an attacker thread that trains the branch predictor which is shared between threads on the same physical core, and a victim thread that is condition by the attacker's training. We design SPECULATOR to support also an attack/victim scenario and used to analyze RSB and BTI (Section 5.2).

Even though speculative execution markers cannot be used in a real world attack since they require *root* access to the machine, they represent a valuable information source to verify the feasibility of a technique in a controlled and noise-free environment. Once an attack is proven to be working with speculative markers, it is easier to transition to methodology that do not require *root* access but that tend to be more noisy like cache side channels.

5.1 SPLITSPECTRE

Here we try to mount a modified version of Spectre v1 we call SPLITSPECTRE. Conceptually, we want to try to run a Spectre v1 attack with the attacker being able to provide the second of the two

array accesses required. The aim is to lower the requirements for the attack, as gadgets for Spectre v1 are difficult to find in real software. We provide a detailed description of this attack in the Annex B, Figure 9.

We implement SplitSpectre on SpiderMonkey 52.7.4, Firefox's Javascript engine with standard configuration parameters. Although we found a real-world gadget corresponding to this attack easily (using `string.charCodeAt`), we were not able to make the exploit work. For a depiction of this attack, we refer to the Annex B, Figure 10.

To better understand the issues leading to the failed exploitation, we extracted the corresponding sequence of instructions from the trace of the attack and used them as a test inside Speculator. The result of the experiment show that the speculation window is too short to perform both accesses. This fact is further confirmed when we run the attack using a shorter function that we manually provide in the Javascript engine: in this case, the attack is successful.

We can draw two important conclusions from the outcome of this experiment. First of all, Speculator enables a systematic approach to the study of new attacks: *i*) formulate an hypothesis on a possible speculative execution attack; *ii*) identify a target and collect execution traces; *iii*) use the execution traces as part of a Speculator snippet; *iv*) insert appropriate markers and gather results; *v*) repeat on all the desired architectures. Secondly, although no exploitation of SplitSpectre is known, the attack is theoretically feasible and there may be either architectures with a long enough speculation window to enable immediate exploitation on SpiderMonkey, or other exploitable targets with shorter gadgets.

5.2 BTI

One interesting scenario, we investigated with Speculator, is the feasibility of BTI poisoning between co-located processes. We leverage the capability of Speculator to run in attacker and victim mode. We design the victim process to perform an indirect `call` to a certain location `A`. Also, at a location `B`, we insert a marker instruction that is never executed by the victim process. We structure the attacker process with an indirect `call` aligned with the `call` in the victim process.

We run the test with attacker and victim on two co-located threads. We start the attacker before the victim to make sure that the indirect `call` in the attacker precede the one in the victim. Then, we start the victim and we observe the counter of the speculative execution marker at `B`. When the injection is successful, we observe the marker at `B` to be speculative executed by the victim. Our success rate is up to 82% over a thousand runs on Skylake and Kaby Lake and up to 55% on Coffe Lake and Broadwell. We report no success on AMD Ryzen.

5.3 Mitigations

Another interesting application of Speculator is to test attack scenarios in the presence of mitigations. For instance, using the BTI poisoning test describes in Section 5.2, we test the current Spectre v2 mitigations available in the kernel. We focus on the following three: `STIBP` [26], `IBRS` [25] and `IBPB` [24]. These countermeasures require either microcode updates, or kernel updates or both. Our findings show that `BTI` between user space processes is mitigated only if `STIBP` is forced on all the applications or enabled conditionally by the use of `SECCOMP` or `prctl` from within the application.

It is worth noting that while these countermeasures are effective, the default settings in all the machines we analysed do not enable them, and very few application uses `SECCOMP`, and none `prctl`, to enable request `STIBP` protection leaving them vulnerable to such attacks. We leave the complete analysis of the remaining security countermeasures implemented against SEAs to future work.

6 RELATED WORK

Speculative Execution. Optimizing CPU instruction throughput through speculative execution has been extensively analyzed and implemented in the 1990s [31, 44, 49]. For information about the microarchitecture of CPUs with respect to out-of-order and speculative execution, we mostly have to rely on the material provided by the CPU manufacturers [2, 4, 5]. Unfortunately this material often just provides software performance optimization related aspects, not providing details on how mechanisms such as the branch predictor work. Agner Fog's work [18] sheds light on those details, providing detailed information backed by a substantial amount of experimental research on the microarchitectural aspects of CPUs. This information is leveraged in processor simulators such as gem5 [10].

Cache Side Channels. Many Spectre variants rely on cache side channels to infer the memory contents accessed by speculative execution. Cache side channels have been extensively studied: First, Tromer et al. introduced both the "evict-and-time" and "prime-and-probe" techniques to efficiently perform a cache attack on AES [45]. Prime and probe is a popular technique, which was also used for certain Spectre variants. "Flush-and-reload" [52] is a technique that allows for higher precision and is used in NetSpectre. Recently, other techniques such as "flush-and-flush" [21] and "prime-and-abort" [15] were presented. Flush and flush leverages the fact that `clflush` executes faster in case of a cache hit. Prime and abort makes use of Intel's transactional memory mechanism to detect when an eviction has happened without the need to probe the cache.

Security Issues. At the beginning of 2018, three security issues related to speculative execution known as Spectre and Meltdown were revealed [29, 33, 54]. CPU vendors reacted with reports on those issues [20, 23] and how they affected their CPU architectures. These initial reports were followed by more security issues, involving further speculative execution triggers [3, 8, 22] and side channels [19], even affecting Intel's virtualization and secure enclave technology SGX [47]. In addition to that, research groups have established remote Spectre attack vectors over the network [42]. The classic buffer overflow to overwrite the return address on the stack also has a speculative execution context twist, as shown in [28, 30, 35].

Mitigations. Apart from the microcode updates shipped by CPU vendors that are discussed earlier, certain mitigations against SEAs can be implemented in software. Especially JavaScript engines deployed mitigations against Spectre v1 such as diluting timing precision, disabling concurrent threads to prevent homebrew-timers and masking pointer accesses to prevent speculative out-of-bounds accesses [6, 11, 48]. Linux has deployed retpoline [46] in the kernel

to mitigate Spectre v2 by trapping mispredicted indirect branches and the KAISER patches [13] to protect against Meltdown by separating page tables organization for user- and kernel space. Also compiler tool chains have picked up the topic, with LLVM working on introducing data dependencies on loads that might be speculatively executed [12, 37] and MSVC adding speculation barrier instructions such as `lfence` to the compiled binary code [38]. At the same time, research groups have proposed to address the issue in silicon, such as adding microarchitectural shadow structures to the CPU for leakage-free speculation [27] or exposing the microarchitectural state in the ISA [34].

7 CONCLUSION

In this paper, we shed light on security-relevant speculative execution and microarchitectural behavior. We presented Speculator, a novel tool that allow targeted and precise measures of microarchitectural characteristics. Using Speculator, we then investigate speculative execution. We study aspects such as the speculation window for various speculative execution triggers, which is an important factor for the payload of a speculative execution attack. We also show which events stop speculative execution and that some security controls such as NX are still in effect during speculative execution, while others do not act as a barrier such as Intel's MPX bounds checks.

We also show the use of Speculator to write PoC attacks and test mitigations. Writing Speculator-ready tests, highly increase the portability of PoCs and generic tests. We foresee Speculator being used to share reproducible tests that can be employed to verify in a more precise and reliable way the status of a system protection as well as to further reverse engineer CPUs undocumented features.

We released our tool, Speculator, as open source and can be found at: **https://github.com/ibm-research/speculator**

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation (NSF) under grant CNS-1703454 award, ONR grant "In-Situ Malware Containment and Deception through Dynamic In-Process Virtualization", and Secure Business Austria.

REFERENCES

[1] 2009. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon Processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.
[2] 2017. Preliminary Processor Programming Reference (PPR) for AMD Family 17h Models 00h-0Fh Processors. http://support.amd.com/TechDocs/54945_PPR_Family_17h_Models_00h-0Fh.pdf.
[3] 2018. Analysis and mitigation of speculative store bypass. https://blogs.technet.microsoft.com/srd/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass-cve-2018-3639/.
[4] 2018. Intel Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.
[5] 2018. Intel Software Developer Manual. https://software.intel.com/en-us/articles/intel-sdm.
[6] 2018. JIT mitigations for Spectre. https://github.com/Microsoft/ChakraCore/commit/08b82b8d33e9b36c0d6628b856f280234c87ba13.
[7] 2018. Rogue System Register Read. https://software.intel.com/security-software-guidance/software-guidance/rogue-system-register-read.
[8] 2018. SPECULATIVE STORE BYPASS DISABLE.
[9] AMD. 2017. Software Optimization Guide for AMD Family 17th Processors. https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf.
[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (Aug. 2011).
[11] Mathias Bynens. 2018. V8 Untrusted code mitigations. https://github.com/v8/v8/wiki/Untrusted-code-mitigations.
[12] Chandler Carruth. 2018. Speculative Load Hardening. https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html.
[13] Jonathan Corbet. 2017. KAISER: hiding the kernel from user space. https://lwn.net/Articles/738975/.
[14] Arnaldo Carvalho de Melo. 2010. The New Linux 'perf' tools. http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf.
[15] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 51–67. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen
[16] Stephane Eranian. 2006. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. of the 2006 Ottawa Linux Symposium*. 269–288.
[17] Agner Fog. 2017. Test results for AMD Ryzen. https://www.agner.org/optimize/blog/read.php?i=838&v=t.
[18] Agner Fog. 2018. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. https://www.agner.org/optimize/microarchitecture.pdf.
[19] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*.
[20] Richard Grisenthwaite. 2018. Cache Speculation Side-channels. https://developer.arm.com/-/media/Files/pdf/Cache_Speculation_Side-channels.pdf.
[21] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer International Publishing, Cham, 279–299.
[22] Jann Horn. 2018. Spectre v4. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.
[23] Intel. 2018. Analysis of Speculative Execution Side Channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.
[24] Intel. 2018. Deep Dive: Indirect Branch Predictor Barrier. https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-predictor-barrier.
[25] Intel. 2018. Deep Dive: Indirect Branch Restricted Speculation. https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-restricted-speculation.
[26] Intel. 2018. Deep Dive: Single Thread Indirect Branch Predictors. https://software.intel.com/security-software-guidance/insights/deep-dive-single-thread-indirect-branch-predictors.
[27] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. 2018. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. *CoRR* (2018). http://arxiv.org/abs/1806.05179
[28] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. https://people.csail.mit.edu/vlk/spectre11.pdf.
[29] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy*.
[30] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. *CoRR* (2018). http://arxiv.org/abs/1807.07940
[31] Butler W. Lampson. 2008. Lazy and Speculative Execution in Computer Systems. In *ACM SIGPLAN Conference on Functional Programming*.
[32] John Levon. 2002. Oprofile. http://oprofile.sourceforge.net.
[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.
[34] Jason Lowe-Power, Venkatesh Akella, Matthew K. Farrens, Samuel T. King, and Christopher J. Nitta. 2018. Position Paper: A Case for Exposing Extra-architectural State in the ISA. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*.
[35] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 2109–2122. https://doi.org/10.1145/3243734.3243761
[36] Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. 2019. Two methods for exploiting speculative

control flow hijacks. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/woot19/presentation/mambretti
[37] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. 2018. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *CoRR* (2018). http://arxiv.org/abs/1805.08506
[38] Andrew Pardoe. 2018. Spectre mitigations in MSVC. https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/.
[39] Mikael Pettersson. 2006. PerfCtr. http://user.it.uu.se/~mikpe/linux/perfctr/.
[40] Sundaram Ramakesavan and Juan Rodriguez. 2016. Intel Memory Protection Extensions Enabling Guide. https://software.intel.com/en-us/articles/intel-memory-protection-extensions-enabling-guide.
[41] T. RÃűhl, J. Eitzinger, G. Hager, and G. Wellein. 2017. LIKWID Monitoring Stack: A Flexible Framework Enabling Job Specific Performance monitoring for the masses. In *IEEE International Conference on Cluster Computing (CLUSTER)*.
[42] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Computer Security – ESORICS 2019*, Kazue Sako, Steve Schneider, and Peter Y. A. Ryan (Eds.). Springer International Publishing, Cham, 279–299.
[43] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 157–173.
[44] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. 1993. Speculative Execution and Branch Prediction on Parallel Machines. In *International Conference on Supercomputing*.
[45] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology* 23, 1 (01 Jan 2010), 37–71. https://doi.org/10.1007/s00145-009-9049-y
[46] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886.
[47] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.
[48] Luke Wagner. 2018. Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/.
[49] David W. Wall. 1991. Limits of Instruction-level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. ACM, New York, NY, USA, 176–188. https://doi.org/10.1145/106972.106991
[50] Vincent M Weaver. 2013. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, Vol. 13.
[51] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. https://foreshadowattack.eu/foreshadow-NG.pdf.
[52] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 719–732. http://dl.acm.org/citation.cfm?id=2671225.2671271
[53] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. 2009. Accuracy of performance counter measurements. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 23–32.
[54] Google Project Zero. 2018. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.ch/2018/01/reading-privileged-memory-with-side.html.

A FURTHER FINDINGS

A.1 Out-of-order execution bandwidth

Speculative execution is no different in how it uses the resources available in both the front- and the back-end of a CPU compared to regular execution. On Intel platforms, instructions that have been fetched and decoded into μops by the front-end are entered in the reorder buffer of the back-end. This buffer contains all μops that are currently "in flight", which means they are either ready for execution, are currently being executed, or have finished execution. The buffer's name derives from the fact that on modern CPUs μops are executed *out-of-order*. This means they are dispatched to execution units based on their data flow dependencies, rather than the control flow of the program. After being executed, they remain in the reorder buffer until they are retired. Retirement of μops happens at an assembly-instruction granularity and *in-order*, honoring the control flow of the program. When μops are retired, the outcome of their computation is committed to the program's state.

The size of the reorder buffer is a natural upper bound on the length of a sequence of instructions that can be speculatively executed. That is, the reorder buffer would hold the branch instruction that triggered speculative execution plus the instructions of the code path being speculatively executed. The branch instruction is the first one that is retired in-order, potentially causing all other μops in the buffer to be canceled in case of misprediction. If the branch instruction takes time to retire, e.g. because it depends on a compare that requires a slow memory access, chances are higher that the reorder buffer is filled with μops that are speculatively executed than for a branch that retires quickly. If the reorder buffer is full, the whole CPU back-end stalls.

A large reorder buffer is beneficial for attacks that exploit speculative execution because it lets a larger amount of instructions be speculatively executed, enhancing the capabilities of a speculative execution attacker. While the size of the reorder buffer is typically a known attribute of a CPU, we decided to empirically verify this number to show how precise measurements taken by Speculator are. In our experiment, we use the `UOPS_EXECUTED.CORE` counter (see Section 3.5). Since the counter operates at the granularity of a core, we disable SMT to reduce the noise caused by Hyperthreads that are scheduled on the same core. We also use the `BR_MISP_RETIRED` counter, which counts the number of mispredicted, retired branch instructions.

When relying on the count of executed μops to measure the reorder buffer size, we need to keep in mind that the μops actually need to execute before the branch that triggered speculative execution is retired. This means we need instructions that execute quickly to achieve maximum throughput. Since "regular" instructions would easily saturate the available execution ports and units, we pick the `NOP` instruction. `NOP` is decoded into a single μop, which occupies a single slot in the reorder buffer. It does not actually execute and thus neither requires an execution unit nor is it captured by the counter that measures executed μops. We thus put an arbitrary regular instruction as a marker at the end of the `NOP`-sled, increasing the latter in size for each test generated. When running this test with Speculator, we expect to measure a constant amount of μops executed up to the point, where the `NOP`-sled takes up all slots in the reorder buffer and the terminating instruction is no longer speculatively executed. Indeed, the results match our expectation: as can be seen in Figure 7, the number of executed μops is constant up until 188 `NOP`s on Broadwell and 220 `NOP`s on Skylake. In addition to the `NOP`s we also need to account for the branch instruction, taking up two slots in the reorder buffer as well as the marker instruction, taking up yet another two entries. In total, this is in line with the specifications published by Intel, which state a reorder buffer size of 192 entries for Broadwell and 224 entries for Skylake.

Interestingly, the number of executed instructions differs for the architectures: it is 34 and 32 for Broadwell and 32 to 30 for Skylake, in spite of the code being exactly the same. Presumably, this is caused by extended μop-fusion introduced as optimization on Skylake. Fused μops count as a single μop.

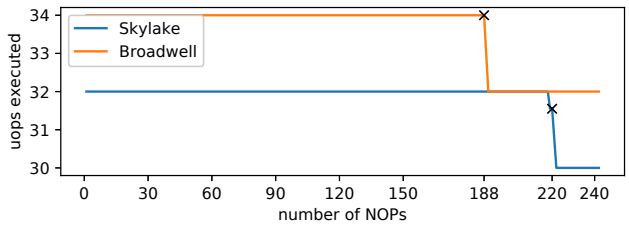


Figure 7: Reorder buffer size test results on Broadwell and Skylake. Since the marker instruction is no longer executed for a sufficiently large number of `NOP`s, the number of executed μops drops at the size of the reorder buffer.

AMD's Zen platform has a construct similar to Intel's reorder buffer: the retire queue. Every μop that has entered the back-end and not been either retired or canceled takes a slot in this queue. Our Ryzen CPU does not feature a counter for executed μops, so we can only provide a measurement based on our marker instruction in this case. The marker instruction, which takes up four μops in this case, is executed up until 186 `NOP`s. This is in line with the size of the retire queue, which is specified to have 192 entries (= 186 + 2 + 4). Interestingly, the speculation window seems to be halved when we switch off SMT: we recognize execution of the marker instruction only up to 91 `NOP`s.

A.2 Empty RSB behavior in pre-Skylake CPUs

During our RSB test, we run the test on all the Intel machines we have available that are listed in Table 1. Meanwhile the results related to the actual length of the RSB give us expected results, we notice that the behavior of the CPU after the RSB is emptied is different for machine pre-Skylake.

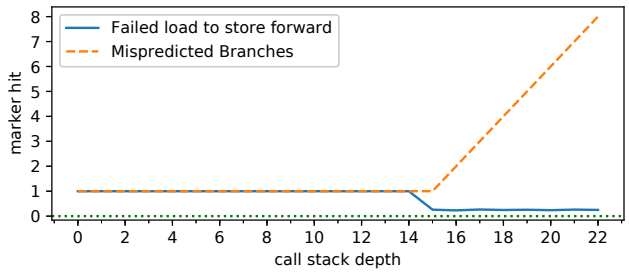


Figure 8: RSB test on Broadwell. As in the AMD case, Broadwell is able to predict the location of our target even if the RSB is empty.

As Figure 8 shows, the CPU (in this case a Broadwell CPU) is still able to hit the expected return location around 25% of the time even though the RSB is actually empty. This behavior differs with the one of newer machines like Kaby Lake presented in Figure 3. This indicates that the re-design that happened in Skylake, and its optimizations, affected the second line of prediction in case of very deep call stack like the one we purposefully tested.

B SPLITSPECTRE

B.1 The Gadget

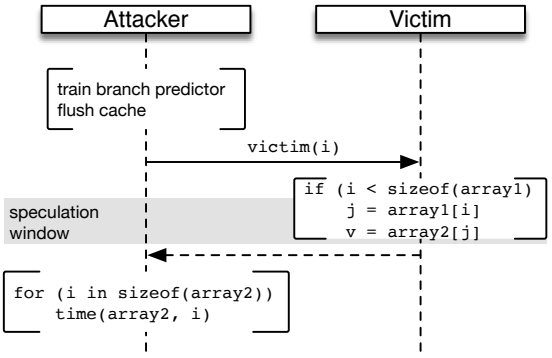
In Spectre v1, the victim code that is executed speculatively ("gadget") consists of three components: *i*) a conditional branch on a variable, typically a length check, *ii*) a first array access that uses the variable from the conditional branch as an offset, and *iii*) a second array access that uses the result of the first array access as an offset. If the conditional branch triggers speculative execution of the following array accesses (phase ❸ described in Section 2.2), the first array access may access an out-of-bounds memory region, revealing the contents of this region through a side channel (phase ❹) by measuring the access time to the second array after executing the gadget (phase ❺).

Although Spectre v1 is powerful and does not rely on SMT, it requires such a gadget to be present in the victim's attack surface. Google Project Zero writes in their original blog post on Spectre v1 [54] that they could not identify such a vulnerable code pattern in the kernel, and instead relied on eBPF to place one there themselves.

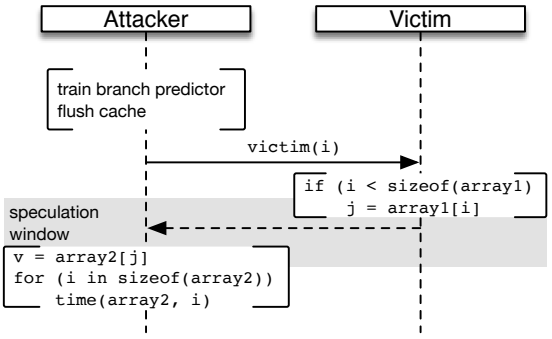
In this point lies the idea of our Spectre v1 variant, SplitSpectre. As its name implies, it splits the Spectre v1 gadget into two parts: one consisting of the conditional branch and the array access (phase ❸), and the other one consisting of the second array access that constitutes the sending part of the side channel (phase ❹). This has the advantage that the second part, phase ❹, can now be placed into the attacker-controlled code. It is more likely that an attacker finds such gadgets, thereby alleviating one of the main difficulties of performing a v1 attack. Furthermore, the attacker can choose to employ amplification of a v1 attack by reading multiple indices of the second array to deal with imprecise time sources.

Figure 9 compares the regular Spectre v1 with our split version. As shown in the figure, the speculation window needs to be sufficiently large such that it still covers the second part. We define the *speculation window* (short for speculative execution window) as the time interval between the event that triggers speculative execution, e.g. a branch condition, and the point in time when it is resolved and the speculatively executed instructions are either retired or rolled back. The speculation window is measured in cycles and determines how many instructions of a given sequence are speculatively executed. The number of instructions of a given sequence that can be speculatively executed at a given time also depends on the CPU's microarchitecture. For example, some instructions are more "expensive" in the sense that they are split into a number of μops, and thus take a long time to execute. Also, the combination of instructions in a sequence affects how fast they execute: similar instructions might lead to congestion on the execution ports, as they require similar execution units.

The speculation window caps the maximum number of instructions executed between the two parts. Extending the length of the speculation window is an instrumental part in extending the capabilities of a speculative execution attacker and the reach of a SplitSpectre attack. In the course of the paper, we show how we use Speculator to evaluate SplitSpectre and speculative execution aspects relevant to its feasibility.



(a) Regular Spectre v1. The gadget requires two dependent array accesses in the victim's attack surface.



(b) Split Spectre v1. The second, dependent array access from a regular v1 gadget moves to the attacker code.

Figure 9: A comparison of regular Spectre v1 and SplitSpectre. While SplitSpectre only requires a simple array access, the speculation window needs to be sufficiently large to contain both the gadget and the second array access exercised by the attacker.

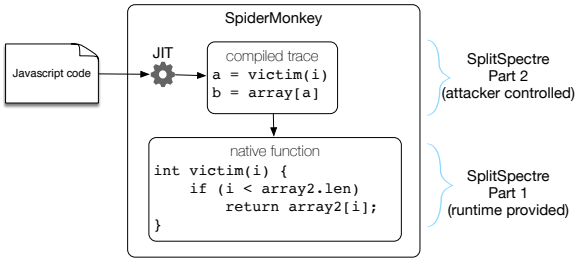


Figure 10: A conceptual view of a SplitSpectre attack instance with JavaScript.

B.2 The Analysis

We mounted a SplitSpectre attack in a real-world setting. We chose a browser-like setting, where untrusted JavaScript is executed in a trusted runtime environment, establishing a privilege boundary. Recall that a v1 gadget consists of a bounds check and two array accesses, the first one using the provided index and the second one using the content of the first array at that position as an index into the second array. In order to mount a regular Spectre v1 attack, we would require a complete Spectre v1 gadget available in the

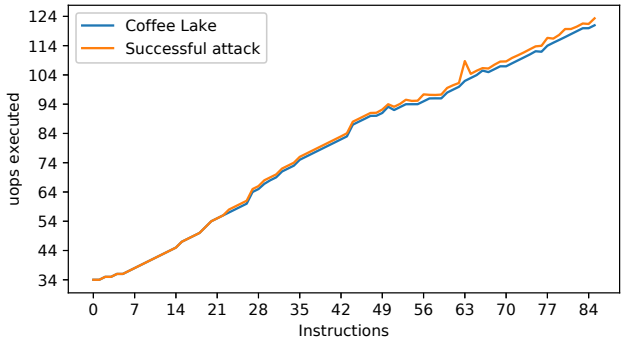


Figure 11: An examination of the SplitSpectre execution trace between the length check of `string.charCodeAt_impl()` and the second array access using Speculator. The graph shows our results of the test on a Coffee Lake machine. It shows that, on average, we are not reaching the second array access in speculative execution. The small spikes in the graph are caused by mispredicted branches in the trace itself, which lead to nested speculative execution of fast-executing code paths.

JavaScript engine. The intuition behind SplitSpectre permits us to relax this requirement and only require the first half of a V1 gadget, i.e. the bounds check and the first array access. The second half of this gadget is provided by attacker-controlled JavaScript code (Figure 10). The attack can only work if speculative execution spans across the privilege boundary from the bounds check in the runtime environment to the second array access in the attacker-controlled, unprivileged code.

We implemented SplitSpectre on SpiderMonkey 52.7.4 – Firefox's JavaScript engine. We use the standard configuration parameters and conducted experiments on our Haswell, Coffee Lake, and Ryzen CPUs.

We start our experiments by introducing a built-in native JavaScript accessor function to SpiderMonkey's source code that returns the content of a pre-allocated array at a given index. This function is the first part of the speculative execution gadget that needs to be part of the victim's attack surface. To simplify the code, we explicitly flush the bounds of the array. Our attacker code is an adapted regular V1 PoC code for JavaScript JIT engines, with just the first array access replaced by the call to the victim function. The time measurement is done using the SharedArrayBuffer technique, which reads the content of such a buffer while it is being incremented in the background by a web worker that is running in parallel.

The attack is successful. That is, on our Coffee Lake platform, we leak a string of ten characters with a success rate of over 86.2%, and we leak the full string with a success rate of 46% (i.e., see Table 4). Investigating the distance between the two parts of the speculation gadget, we measure the distance after 50 training runs of the JavaScript code that causes Spidermonkey's tracing JIT to compile an optimized IonJIT trace implementing the JavaScript code in assembly. The distance between the bounds check and the second array access is 43 instructions, which is small enough for the attack to produce reliable results.

	Haswell	Coffee Lake
Runs	100	100
Only highest scoring char	76.6%	76.8%
1st and 2nd highest scoring char	80.7%	86.2%
Full string leaked	10%	46%

Table 4: Success rates for the SplitSpectre attack on JavaScript. We perform 100 runs, each run trying to leak a string of 10 consecutive characters. We provide numbers on both the highest and the second highest scoring characters.

We proceed with our experiments by replacing our native built-in function with code already present in the SpiderMonkey source. Our scan for a suitable gadget reveals the built-in `string.charCodeAt()` function, which returns the character code of a string at a given index and is implemented in native code. Internally, `string.charCodeAt()` calls `string.charCodeAt_impl()`, which includes the bounds check and actual access. Unfortunately, the speculation window is not large enough for the attack to work with `string.charCodeAt()`: After 50 training runs, the distance between the compare in `string.charCodeAt_impl()` and the dereference of the second array in the JIT trace is 90 instructions. An examination of the extracted execution trace with Speculator shows that the number of speculatively executed μops is, on average, slightly lower than necessary for a successful attack (Figure 11). This means that in this scenario, the crucial load instruction is not always reached during speculative execution.

We also examine the execution trace on an AMD Ryzen CPU using a marker instruction, since the Zen performance counters do not feature a generic counter for executed instructions. We observe the marker instruction being executed for the full length of the trace. However, even here, the granularity of the time measurement is too coarse-grained to permit a successful read of the cache side channel. Amplifying the attack by adding multiple dependent array accesses would extend the trace so that it no longer fits into the speculation window.

We further optimize the attack by reducing the amount of code that is executed between the bounds check and the second access. This is achieved by implementing the second access and the call to the victim function in web assembly, which allows even more attacker control over the compiled JIT trace. However, using WebAssembly actually increases the number of instructions between the compare and the second access to 107. This is because the native call is not made directly from within the WebAssembly. Rather, additional JavaScript glue code is invoked.

JIT engine authors have already reacted with countermeasures [11, 48] in order to mitigate Spectre v1 in the context of browsers. These countermeasures mostly address sources for high-precision timers. Diluting the timing and disabling homebrew sources such as SharedArrayBuffers mitigate this version of JavaScript SplitSpectre. However, it remains to be seen if amplification of the attack's timing properties make it feasible if only coarse-grained time sources are available.

On top of timing-related countermeasures, the V8 engine also masks addresses and array indices in JITted code before dereferences. While this mitigates a standard Spectre v1 attack, it does not help with SplitSpectre, where the bounds check is actually not exercised in JITted code, but the engine code itself.

Our analysis lead us to conclude that the attack is viable, and that the ability to trigger it in practice depends on the identified microarchitectural properties of individual CPU families. We leave a comprehensive analysis of these properties for the various CPU architectures/models as an item of future work, which can be aided by Speculator.