# ret2spec: Speculative Execution Using Return Stack Buffers

Accepted at ACM CCS 2018 (preliminary version)

Giorgi Maisuradze
CISPA, Saarland University
giorgi.maisuradze@cispa.saarland

Christian Rossow
CISPA, Saarland University
rossow@cispa.saarland

## ABSTRACT

Speculative execution is an optimization technique that has been part of CPUs for over a decade. It predicts the outcome and target of branch instructions to avoid stalling the execution pipeline. However, until recently, the security implications of speculative code execution have not been studied.

In this paper, we investigate a special type of branch predictor that is responsible for predicting return addresses. To the best of our knowledge, we are the first to study return address predictors and their consequences for the security of modern software. In our work, we show how return stack buffers (RSBs), the core unit of return address predictors, can be used to trigger misspeculations. Based on this knowledge, we propose two new attack variants using RSBs that give attackers similar capabilities as the documented Spectre attacks. We show how local attackers can gain arbitrary speculative code execution across processes, e.g., to leak passwords another user enters on a shared system. Our evaluation showed that the recent Spectre countermeasures deployed in operating systems also cover such RSB-based cross-process attacks. Yet we then demonstrate that attackers can trigger misspeculation in JIT environments in order to leak arbitrary memory content of browser processes. Reading outside the sandboxed memory region with JIT-compiled code is still possible with 80% accuracy on average.

## KEYWORDS

Hardware Security, Side Channel Attacks, JavaScript

## 1 INTRODUCTION

For decades, software has been able to abstract from the inner workings of operating systems and hardware, and significant research resources have been spent on assuring software security. Yet only recently, the security community has started to investigate the security guarantees of the hardware underneath. The first investigations were not reassuring, revealing multiple violations of security and privacy, e.g., demonstrating that cryptographic keys meant to be kept secret may leak via caching-based side channels [5, 31, 32]. This recent discovery has piqued interest in the general topic of microarchitectural attacks. More and more researchers aim to identify potential problems, assess their impact on security, and develop countermeasures to uphold previously-assumed security guarantees of the underlying hardware. As a consequence, a variety of novel techniques have been proposed which abuse microarchitectural features, thereby making seemingly-secure programs vulnerable to different attacks [5, 16, 17, 21, 30, 32].

One of the core drivers for recent microarchitectural attacks is the sheer complexity of modern CPUs. The advancement of software puts a lot of pressure on hardware vendors to make their product as fast as possible using a variety of optimization strategies. However, even simple CPU optimizations can severely threaten the security guarantees of software relying on the CPU. Caching-based side channels are a notable example of this problem: such side channels exist since caches that improve the access time to main memory are shared across processes. Thus, caching can result in leaking cryptographic keys [5, 32], key-stroke snooping, or even eavesdropping on messages from secure communications [17, 30].

Besides caching, modern CPUs deploy several other optimization techniques to speed up executions, two of which we will study in more detail. First, in out-of-order execution, instead of enforcing a strict execution order of programs, CPUs can reorder instructions, i.e., execute new instructions before older ones if there are no dependencies between them. Second, in speculative execution, CPUs predict the outcome/target of branch instructions. Both these strategies increase the utilization of execution units and greatly improve the performance. However, they also execute instructions in advance, meaning they can cause instructions to execute that would have not been executed in a sequential execution sequence. For example, it can happen that an older instruction raises an exception, or that the speculative execution unit mispredicts. In this case, the out-of-order executed instructions are rolled back, restoring the architectural state at the moment of the fault (or misspeculation). Ideally, the architectural state is the same as in strict sequential execution. However, this is not the case: instructions executed out of order can influence the state in a manner that can be detected. Meltdown [26] and Spectre [22] are great examples of this class of problems. Meltdown exploits a bug in Intel's out-of-order engine, allowing the privileged kernel-space data to be read from unprivileged processes. Spectre poisons the branch target buffer (BTB) and thus tricks the branch prediction unit into bypassing bounds checks in sandboxed memory accesses, or even triggering arbitrary speculative code execution in different processes on the same core. To mitigate these threats, operating systems had to make major changes in their design (e.g., isolating the kernel address space from user space [15]), and hardware vendors introduced microcode updates to add new instructions to control the degree of the aforementioned CPU optimization techniques [19].

In this paper, we further investigate speculative execution and show that attacks are possible beyond the already-documented abuse of BTBs. More specifically, we look into

the part of branch prediction units that are responsible for predicting return addresses. Since they are the core of the return address predictor, we will in particular investigate the properties of return stack buffers (RSBs). RSBs are small microarchitectural buffers that remember return addresses of the most recent calls and speed up function returns. Given that return addresses are stored on the stack, without such RSBs, a memory access is required to fetch a return destination, possibly taking hundreds of cycles if retrieved from main memory. In contrast, with RSBs, the top RSB entry can be read instantaneously. RSBs thus eliminate the waiting time in the case of a correct prediction, or in the worst case face *almost*[1] the same penalty in the case of a misprediction.

Despite being mentioned as *potential* threat in the initial report from Google Project Zero [18] and Spectre [22], the security implications of abusing RSBs have not yet been studied, and only very recent studies have started to investigate timing implication of return address mispredictions at all [37]. Nevertheless, modern operating systems' kernels also contain countermeasures against crafted RSB entries, which indicates their awareness of *potential* security implications. However, to the best of our knowledge, we are the first to systematically study and demonstrate the *actual* security implications of RSBs. We furthermore show the degree to which attackers can provoke RSB-based speculative execution by overflowing the RSB, by crafting malicious RSB entries prior to context switches, or by asymmetric function call/return pairs.

Based on these principles, we provide two RSB-based attack techniques that both allow attackers to read user-level memory that they should not be able to read. In the first attack (Section 4), we assume a local attacker that can spawn arbitrary new programs that aim to read another user's process memory. To this end, we show how one can poison RSBs to force the collocated processes (on the same logical core) to execute arbitrary code speculatively, and thus report back potential secrets. On the one hand, this attack has recently been prevented by all major operating systems that now flush RSBs upon context switches to mitigate Spectre, presumably anticipating potential RSB underflows that trigger the BTB[2]. On the other hand, the general attack concept shows that RSB-based speculated execution (i) can indeed be provoked by local attackers with non-negligible probability, and (ii) goes beyond the currently-assumed problem of falling back to the BTB (thus allowing for Spectre) when underflowing RSBs, and thus, can be generalized to the non-trustworthiness of attacker-controlled RSBs.

In our second attack (Section 5), we investigate how attackers can abuse RSBs to trigger speculation of arbitrary code inside the same process—notably *without* requiring a context switch, and thus effectively evading the aforementioned defense. We assume an attacker that controls a web site the target user visits, and by carefully crafting this web site, aims to read memory of the victim's browser process. Technically, we leverage just-in-time (JIT) compilation of WebAssembly to create code patterns that are not protected by memory isolation techniques and thus can read arbitrary memory of a browser process. By doing so, we show that adversaries can bypass memory sandboxing and read data from arbitrary memory addresses.

Both attack types demonstrate that speculative execution is not limited to attackers penetrating the BTB. While our attacks result in similar capabilities as Spectre, the underlying attack principles to manipulate the RSB are orthogonal to the known poisoning strategies. We thus also discuss how existing and new countermeasures against RSB-based attacks can mitigate this new risk (Section 6). We conclude the paper with vendor and developer reactions that we received after responsibly disclosing the internals of this new threat.

In this paper, we provide the following contributions:

- We study the return address predictor, an important yet so far overlooked module in the prediction unit. To the best of our knowledge, we are the first to demonstrate the actual abuse potential of RSBs.
- We propose attack techniques to trigger misspeculations via the RSB. This can be useful in future studies that will target speculative code execution. In contrast to using the branch predictor, which requires a prior training phase, RSBs can be forced to misspeculate to required addresses without prior training.
- We then propose cross-process speculative execution of arbitrary code (similar to Spectre/Variant 1). We evaluate the results by leaking keystrokes from a specially-crafted `bash`-like program. Using our synthetic program example, we demonstrate that such attacks are in principle conceivable, showing the importance of existing OS-based defenses to mitigate Spectre.
- Finally, we show how to trigger misspeculations via RSBs in JIT-compiled code. We leverage this to execute arbitrary code speculatively and, by doing so, bypass memory sandboxing techniques, allowing arbitrary memory reads. We evaluate our technique in Firefox 59 (with a modified timer for higher precision).

## 2 BACKGROUND

In the following, we will present the key features of x86 that are important to understand for the remainder of this paper. While similar concepts are also common in other architectures, for brevity and due to its popularity, we focus on x86.

### 2.1 Out-of-Order Execution

Being a CISC (Complex Instruction Set Computing) architecture, x86 has to support a multitude of instructions. Implementing all such instructions in circuits would require an enormous amount of transistors, thus also drastically increasing the power consumption. Therefore, under the hood, both main manufacturers of x86 CPUs (Intel and AMD) use micro-OPs, which can be seen as a simplified RISC (Reduced Instruction Set Computing) machine that runs inside the CPU. All instructions from the x86 ISA are then dynamically

[1] Rolling back the pipeline on misspeculation adds an overhead of a few cycles.

[2] https://patchwork.kernel.org/patch/10150765/

decoded into their corresponding micro-OPs, and are then executed on much simpler execution units. This allows manufacturers to reduce the number of required execution paths, decreasing both production cost and power consumption.

Having a variety of different instructions, sequential execution becomes another bottleneck. The concept of splitting up complex instructions into smaller operations also makes it possible to reorder the execution of micro-OPs to gain performance. In a strict sequential execution, an instruction $N$ cannot be started unless all preceding instructions, $1..N-1$, are finished executing. This is especially problematic for instructions with completely different timing properties, e.g., zeroing a register and reading a value from main memory. Out-of-order execution deals with this issue by executing instructions out of order, provided they do not depend on one another.

To implement out-of-order execution, x86 maintains a so-called reorder buffer (ROB), which keeps a FIFO buffer of micro-OPs in their original order, while executing them out of order. If a micro-OP is in the ROB it (i) is waiting for its dependencies to be resolved, (ii) is ready to be executed, (iii) is already being executed, or (iv) is done executing but was not yet committed. Committing (also called *retiring*) a micro-OP reflects its changes back to the architectural state, e.g., modifying the architectural (ISA-visible) registers or writing data back to memory. Given that the programs assume a strict sequential order, the ROB commits instructions in order such that the architectural state is updated sequentially.

## 2.2 Speculative Execution

Modern CPUs augment out-of-order execution with an orthogonal feature called speculative execution. The key observation here is that while executing instructions, CPUs can encounter a branch instruction that depends on the result of a preceding instruction. This would never happen in a strict sequential (non-parallel) execution, as all previous instructions before the branch would have been resolved. To cope with this problem in modern CPUs that execute multiple instructions in parallel, the simplest solution is to wait until the branch condition/target is resolved, and only then continue the execution. However, this would serialize branch executions, which would degrade the performance, especially given the high number of branch instructions in x86. Speculative execution represents an efficient alternative problem solution and is consequently used in all modern CPUs. Speculative execution uses a branch prediction unit (BPU), which predicts the outcome of conditional branch instructions (i.e., taken/not taken). The out-of-order engine then continues execution on the predicted path. This general concept is not limited to direct branches that always have a fixed jump target. For example, consider indirect branches (such as indirect calls and jumps) that need to be resolved before being executed, i.e., the branch target can be stored either in a register or in memory. In this case, the branch destination is the value that needs to be predicted. To support indirect branches, the branch target buffer (BTB) stores a mapping between the branch source and its likely destination.

The two recently disclosed microarchitectural attacks, Spectre and Meltdown, abuse the aforementioned out-of-order and speculative execution engines. Meltdown uses the fact that out-of-order engines do not handle exceptions until the retirement stage, and leverages it to access memory regions that would otherwise trigger a fault (e.g., kernel memory). In Meltdown, authors exploit the bug in Intel's out-of-order execution engine, which reveals the data from the faulty memory access for a few cycles. This time, however, is enough to do a dependent memory access on the data. Although the dependent memory access will be flushed from the pipeline after handling the fault, the cache line for that address will remain cached, thus creating a side channel for leaking the data. Conversely, Spectre uses legitimate features of branch predictors to mount an attack: it mistrains the BPU for conditional branches in Variant 1, and injects BTB entries with arbitrary branch targets in Variant 2. Variant 1 can be used to bypass bounds checking and thus read outside the permitted bounds, while Variant 2 allows cross-process BTB injection, allowing arbitrary speculative execution of the code in other processes on the same physical core.

## 2.3 Return Stack Buffers

Return instructions are a specific type of indirect branch and always jump to the top element on the stack (i.e., translated to `pop tmp; jmp tmp`). Consequently, in principle, BTBs can also be used here as a generic prediction mechanism. However, given that functions are called from multiple different places, BTBs will frequently mispredict the jump destination. To increase the prediction rate, hardware manufacturers rely on the fact that call and return instructions are always executed in pairs. Therefore, to predict the return address at a return site, CPUs remember the address of the instruction following the corresponding call instruction. This prediction is done via return stack buffers (RSBs) that store the $N$ most recent return addresses (i.e., the addresses of instructions after the $N$ most recent calls). Note that, in case of hyperthreading, RSBs are dedicated to a logical core. The RSB size, $N$, varies per microarchitecture. Most commonly, RSBs are $N = 16$ entries large, and the longest reported RSB contains $N = 32$ entries in AMD's Bulldozer architecture [11]. In this paper, we assume an RSB size of 16, unless explicitly stated otherwise, but our principles also work for smaller or larger RSBs.

RSBs are modified when the CPU executes a call or return instruction. Calls are simple: a new entry (the address of the next instruction) is added to the RSB and the top pointer is incremented. If the RSB was already full, the oldest entry will be discarded. Conversely, in case the of a return instruction, the value is taken from the RSB, the top pointer is decremented, and the read value is used for prediction.

Due to the their limited size, it naturally happens that the RSBs cannot fit all the return addresses. For example, $N+1$ calls followed by $N+1$ returns will underflow the RSB in the last return instruction. The way such an underflow

is handled depends on the microarchitecture. There are the following possible scenarios: (a) stop predicting whenever the RSB is empty, (b) stop using the RSB and switch to the BTB for predictions, and (c) use the RSB as a ring buffer and continue predicting (using $idx\%N$ as the RSB top pointer). Out of these scenarios, (a) is the easiest to implement; however, it stops predicting return addresses as soon as the RSB is empty. The prediction rate is improved in (b), which incorporates the BTB to predict return destinations. However, the improvement is made at the expense of BTB entries, which might detriment other branches. Finally, (c) is an optimization for deep recursive calls, where all RSB entries return to the same function. Therefore, no matter how deep the recursion is, returns to the recursive function will be correctly predicted. According to a recent study [37], most Intel CPUs use the cyclic behavior described in variant (c), while AMD's use variant (a) and stop prediction upon RSB underflows. Intel microarchitectures after Skylake are known to use variant (b)[3]. Throughout the paper, we will refer to (c) as cyclic, and (a) and (b) as non-cyclic.

## 3 GENERAL ATTACK OVERVIEW

Before detailing specific attack scenarios, in this section, we introduce the basics of how RSB-based speculative execution can be achieved and be abused. We explore whether and how attackers may manipulate the RSB entries in order to leak sensitive data using speculative execution that they could not access otherwise. Similar to recent microarchitectural attacks [8, 10, 22, 26, 29], we trick the CPU to execute instructions that would not have been executed in a sequential execution. The goal is to leak sensitive information in speculation, e.g., by caching a certain memory area that can be detected in a normal (non-speculative) execution. The general idea of our attack can be divided into three steps:

- **(A1)** trigger misspeculations in the return address predictor, i.e., enforce that returns mispredict
- **(A2)** divert the speculative execution to a known/controlled code sequence with the required context
- **(A3)** modify the architectural state in speculation, such that it can be detected from outside

**(A1) Triggering Misspeculation:** From an attacker's perspective, enforcing that the return predictor misspeculates upon function return is essential to reliably divert speculative execution to attacker-controlled code (see A2 for how to control the speculated code). Misspeculations can be achieved in several ways, depending on the RSBs underflow behavior (as discussed in Section 2.3).

*Non-Cyclic RSB:* If the RSB stops speculating upon underflow, triggering a misspeculation will require abnormal control flow that violates the common assumption that functions return to their caller. Some examples of such abnormalities are: (i) exception handling, i.e., a try-catch block in the upper call stack and throwing an exception in a lower one (Figure 1a); (ii) a `setjmp/longjmp` pair, i.e., saving the current execution context at the time of calling `setjmp`, and restoring it at any later point when (`longjmp`) is called (the stack layout will be similar to Figure 1a, only with `setjmp/longjmp` instead of `try-catch/throw`); (iii) a context switch from one process to another, where the process being evicted was doing a chain of calls, while the scheduled process will do a sequence of returns (Figure 1b); and (iv) a process that deliberately overwrites the return address to the desired destination and then returns (Figure 1c). Unsurprisingly, (iv) is not commonly used; however, it can be helpful for testing RSBs and triggering the misspeculation on demand. In fact, in contrast to branch predictors, which require training to trigger misspeculation, RSBs can be forced to misspeculate with just a single write instruction (`mov [rsp], ADDRESS; ret`, as in Figure 1c).

*Cyclic RSB:* If RSBs are cyclic, misspeculation can—in addition to the methods mentioned before—be triggered by overflowing the RSB. Figure 1d depicts a scenario in which function `A` calls `B`, function `B` calls `C`, and so on. Being limited in size ($N = 4$ in this example), the RSB only contains the 4 most recently added return addresses. Therefore, after correctly predicting four returns, when returning from `E`, the RSB will mispredict `H` as the return address instead of `D`.

Cyclic RSBs can also be leveraged and prepared by recursive functions. For example, if we have two recursive functions `A` and `B`, and we call them in the following order:

- `A` calls itself recursively $N_A$ times,
- in its deepest recursion level, `A` then calls `B`
- `B` calls itself recursively 16 times (size of the RSB)

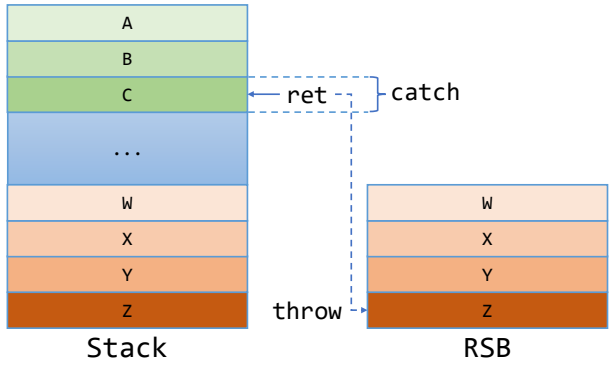
then the first 16 returns, from `B`, will be predicted correctly. However, the remaining $N_A$ returns will be mispredicted, and `B`'s call site will be speculated instead of `A`'s.

**(A2) Diverting Speculative Execution:** Being able to trigger a misspeculation, the next step is to control the code that is executed speculatively. Generally, misspeculation means that instructions from one function (e.g., `B`) are speculatively executed within the context of another (e.g., `A`). As a simple example, consider a function that returns a secret value in `rax`. If this function is predicted to return to code that accesses attacker-accessible memory relative to `rax`, this will leak the secret value. Ideally, we control both, the context and the speculated code; however, having either one or the other can also be sufficient for a successful exploitation.

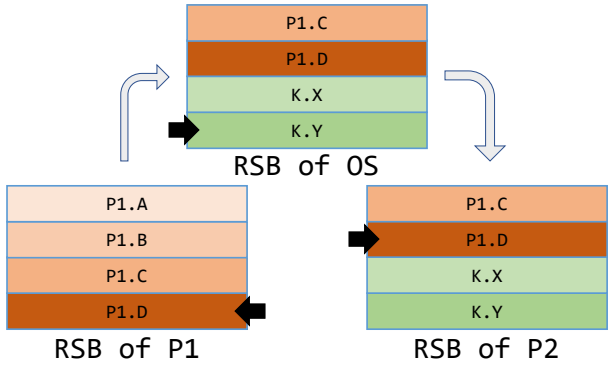
Let function `B` return and trigger a misspeculation in `A` (right after the call to `B`). In the ideal case, we control the code that is misspeculated in `A`, and the context (i.e., the contents of the registers) in `B`. Combining them together allows us to execute arbitrary code speculatively. This will be the case for our attack in Section 5. Another, more complicated case is when the context is fixed, e.g., the values of some registers are known, and we are also limited with the possibly-speculated code, e.g., it can be chosen from existing code pieces. In this case, the challenge is to find code gadgets that use the correct registers from the context to leak their values. For example, if we know that `r8` contains a secret, we need to find a gadget that leaks `r8`. This case will be shown in Section 4.

**(A3) Feedback Channel:** Finally, being able to execute arbitrary code in speculation, we have to report back the results

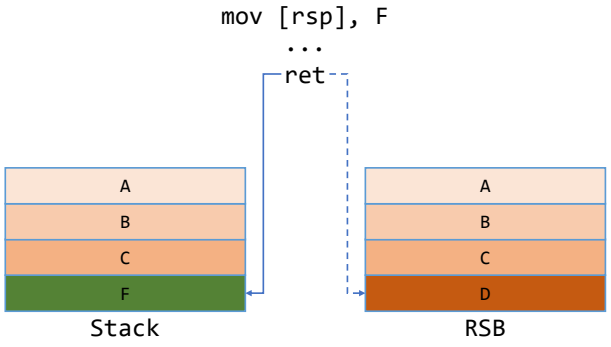
[3] https://patchwork.kernel.org/patch/10150765/



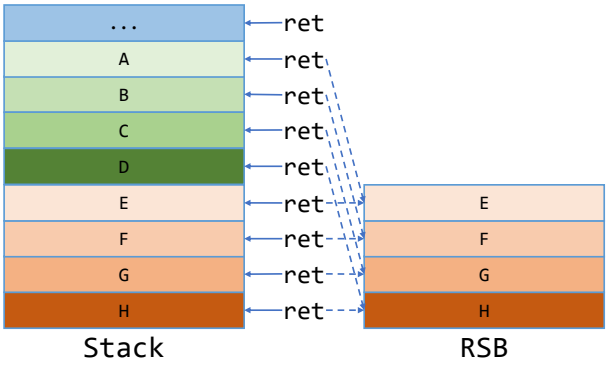

(a) Exception handling: While the RSB predicts a return to function Z, the exception is caught by function C, causing a chain of misspeculations when C returns, as the RSB is misaligned to the return addresses on the stack.




(b) Context switch: When the kernel switches from process P1 to P2, the kernel will only evict a few entries with kernel-internal functions. After the context switch, P2 may thus mispredict and return to the remaining RSB entries that were added by P1.




(c) Direct overwrite: A process can enforce return mispredictions by replacing return addresses stored on the stack.




(d) Circular RSB: After returning $N = 4$ times, the predictor cycles over and will repeat the same prediction sequence of return addresses.

**Figure 1: Ways to enforce RSB misspeculation. We reduced the RSB size to $N = 4$ entries for readability. The bold arrow points to the top element of each RSB. Thin solid arrows indicate actual returns, thin dashed arrows speculated returns.**

from within the speculative execution to the normal execution environment. To this end, similar to several side channels proposed in the past [17, 31, 38], we use secret-dependent memory accesses that modify the caching state. Technically, if we want to leak the value in `rax`, we read attacker-accessible memory using the secret value as offset, e.g., `shl rax, 12; mov r8, [rbx + rax]`. This will result in caching the corresponding memory address (`rbx+rax*4096`, where 4096 bytes is the page size). Therefore, identifying the index of the cached page from `rbx` will reveal the value of `rax`.

The adversary can then use existing side channel techniques to observe these cache changes, such as Flush+Reload [38] or Prime+Probe [31]. Flush+Reload is most accurate, but requires that the attacker and victim processes share memory. Typically this is granted, given that operating systems share dynamic libraries (e.g., `libc`) to optimize memory usage. Alternatively, Prime+Probe [31] works even without shared memory. Here, the attacker measures whether the victim evicts one of the attacker-prepared cache lines. By detecting the evicted cache line, the attacker can leak the address bits corresponding to the cache line.

## 4 CROSS-PROCESS SPECULATIVE EXEC.

In this section, we will describe how an attacker can abuse the general attack methodology described in the previous section to leak sensitive data from another process. In Section 5, we will describe RSB-based attacks in scripting environments to read memory beyond the memory bounds of sandboxes.

### 4.1 Threat Model

In the following, we envision a local attacker that can execute arbitrary user-level code on the victim's system. The goal of the attacker is to leak sensitive data from another process (presumably of a different user) on the system, e.g., leaking input fed to the target process. In our concrete example, we target a command line program that waits for user input

(character-by-character), i.e., a blocking `stdin`, and we aim to read the user-entered data. This setting is in line with Linux programs such as `bash` or `sudo`. The attack principle, however, generalizes to any setting where attackers aim to read confidential in-memory data from other processes (e.g., key material, password lists, database contents, etc.).

For demonstration purposes, we assume that the kernel does not contain the Spectre patches, and thus does not flush RSBs upon a context switch. Furthermore, we assume that the victim process contains all attacker-required gadgets. In our example, we simply add these code pieces to the victim process. Finally, we assume that ASLR is either disabled or has been broken by the attacker.

## 4.2 Triggering Speculative Code Execution

We now apply the general attack principles to the scenario where an adversarial process executes alongside a victim process. The attacker aims to trigger return address misprediction in the victim's process, and divert the speculative control flow to an attacker-controlled location. The fact that victim and attacker are in different processes complicates matters, as the context of the execution (i.e., the register contents) is not under the control of the attacker. To the attacker's benefit, though, the RSB is shared across processes running on the same logical CPU core. This allows the RSB to be poisoned from one process, and then be used by another process after a context switch. For this attack to work, we have to perform the following steps:

- We first fill the RSB with addresses of suitable code gadgets that leak secrets by creating a call instruction just before these gadgets' addresses and executing the call 16 times (step A2 from Section 3).
  RSBs store virtual addresses of target instructions. Therefore, in order to inject the required address, we assume the attacker knows the target process's address space. Alternatively, in the case of a randomized address space (e.g., with ASLR), we can use RSBs the opposite way, i.e., to leak the RSB entries, and thus to reveal the addresses of the victim process.
- After filling the RSB, we force a context switch to the victim process (step A1 from Section 3). For example, the attacker could call `sched_yield` in order to ask the kernel to reschedule, ideally to the victim process. For this, we assume that our process runs on the same logical CPU core as the victim, and thus shares the RSB. This can be accomplished by changing the affinity of our process, to pin it to the victim's core (e.g., by using `taskset` in Linux), or alternatively, spawn one process per logical core.

## 4.3 Proof-of-Concept Exploit

To illustrate the general possibility of such cross-process data leaks, we picked a scenario where an attacker wants to leak user input, e.g., in order to capture user-entered passwords. Thus, in our tested example, the victim is a terminal process that waits for user input, such as `bash` or `sudo`. At a high level, such processes execute the following code:

```
while (inp = read_char(stdin)) {
  handle_user_input(inp);
}
```

The following shows the (simplified) steps taken in a typical iteration of such an input-processing loop:

(1) The main loop starts.
(2) `read_char` is called, which itself calls other intermediate functions, finally reaching the `read` system call.
(3) The `stdin` buffer will be empty (until the user starts typing) and the victim process is thus evicted.
(4) When the user presses a key, the victim process, waiting for buffer input, is scheduled.
(5) Execution continues within the `read` system call (which has just returned), and a chain of returns are executed until the execution reaches the main loop.
(6) `handle_user_input` handles the read character.

In order to leak key presses, the attacker process has to be scheduled before the victim continues execution in (5). This will guarantee that when resuming the execution from `read`, the victim process will use the attacker-tainted RSB entries.

*4.3.1 Leaking Key Presses.* To show the plausibility of this attack, we implemented three programs:

**`Attacker`:** fills up RSB entries and preempts itself, so the victim process is scheduled after it.
**`Measurer`:** probes for a predetermined set of memory pages to leak data from the victim (using Flush+Reload [38]).
**`Victim`:** simulates the behavior of `bash`, i.e., waits for new keystrokes and handles them when they are available. We replicate the call chain similarly to `bash` from the main loop to the `read` system call, and also return the same values.

There are several challenges with this approach:

(1) ASLR: Given that RSBs predict virtual addresses of jump targets, address space randomization in the victim makes tainting the RSB extremely difficult if not impossible.
(2) Small speculation window: Since we use memory accesses as a feedback mechanism, there is a race condition between adversarial memory access in speculation and reading the real return address off the stack.
(3) Post-Meltdown/Spectre ([22, 26]) patches: As RSBs have already been identified as a potential future risk that allows speculative execution, most modern OS kernels nowadays already flush (technically, overwrite) RSBs every time a user process switch to the kernel occurs.
(4) Speculation gadgets: In `bash`, the character returned by the `read` system call is moved into `rax` and the function returns. We targeted this return for speculation; thus, the required gadgets have to first shift `rax` at the very least to a cache-line boundary (i.e., 6 bits to the left), and then do a memory access relative to shared memory (e.g., `r10`, which contains the return address

of the system call, pointing into libc: `shl rax, 6; mov rbx, [r10+rax]`)

Out of these challenges, (3) is a real limitation that cannot be avoided. Flushing the RSB at context switches destroyed the aforementioned attack. To show that this prophylactic patch[4] in modern OSes is indeed fundamental, we for now assume RSBs are not flushed upon context switch. Challenge (4) strongly depends on the compiler that was used to compile the victim program. Therefore, each target requires its unique set of gadgets to be found. Using an improved gadget finder, or by scanning various dynamic libraries, we believe the issue can be solved. For our demo, we added the required gadgets. Limitation (2) can be overcome by using another thread that evicts the addresses from the cache that correspond to the victim's stack. However, this requires revealing the address space, i.e., challenge (1), first. We believe that our attack can be tweaked to derandomize ASLR of other processes, but given the fact that (3) is unavoidable, we did not further investigate this issue. In our experiments, for simplicity, we instead used `clflush` (an instruction invalidating the cache line of the provided address) to evict the victim's stack addresses in order to increase the speculation time window.

### 4.4 Evaluation

In the following, we evaluate the efficacy of our proof-of-concept implementation. We carried out our experiments on Ubuntu 16.04 (kernel 4.13.0), running on Intel's Haswell CPU (Intel® Core™ i5-4690 CPU @3.50GHz). We chose a kernel released before the publication of Spectre, as newer versions flush RSBs as part of their Spectre defense, and thus interfere with our attack.

The execution environment was set according to the attack description in the previous section. In the following, we note some implementation specifics. So as to get rescheduled after each read key, our `Victim` process did not use standard input (`stdin`) buffering, which is in line with our envisioned target programs `bash` and `sudo`. Additionally, a shared memory region is mapped in `Victim`, which will be shared with `Measurer`. In our case, it was an `mmap`-ed file, but in reality this can be any shared library (e.g., `libc`). In order to increase the speculation time, we used `clflush` in the `Victim`. In practice, this has to be done by another thread that runs in parallel to `Victim` and evicts the corresponding address of the `Victim`'s stack. Finally, we also added the required gadget to `Victim`: `shl rax, 12; mov rbx, [r12 + rax]`. At the point of speculative execution (i.e., when returning from `read`), `rax` is the read character and `r12` points to the shared memory region.

`Measurer` maps the shared (with `Victim`) memory in its address space, and constantly monitors the first 128 pages (each corresponding to an ASCII character). In our experiments, we use Flush+Reload [38] as a feedback channel. Finally, to be able to inject entries into `Victim`'s RSB, `Attacker` needs to run on the same logical core as `Victim`. To this end, we modify both `Victim`'s and `Attacker`'s affinities to pin them to the same core (e.g., using `taskset` in Linux). After that, `Attacker` runs in an infinite loop, pushing gadget addresses to the RSB and rescheduling itself (`sched_yield`), hoping that `Victim` will be scheduled afterwards.

To measure the precision of our attack prototype, we determine the fraction of input bytes that `Measurer` read successfully. To this end, we compute the Levenshtein distance [24], which measures the similarity between the source (S) and the destination (D) character sequences, by counting the number of insertions, deletions, and substitutions required to get D from S. To measure the technique for each character in the alphabet, we used the famous pangram "The quick brown fox jumps over the lazy dog". In the experiment, a new character from the pangram was provided to `Victim` every 50 milliseconds (i.e., 1200 cpm, to cover even very fast typers). Running the total of 1000 sentences resulted in an average Levenshtein distance of 7, i.e., an overall precision of ≈84%. It therefore requires just two password inputs to derive the complete password entered by a user using RSB-based speculative execution.

## 5 SPECULATIVE EXEC. IN BROWSERS

The cross-process attack presented in Section 4 has demonstrated how a victim process might accidentally leak secrets via RSB-based misspeculations. In this section, we consider a different setting with just a single process, in which a sandbox-contained attacker aims to read arbitrary memory of a browser process *outside* of their allowed memory bounds.

### 5.1 Threat Model

Scripting environments in web browsers have become ubiquitous. The recent shift towards dynamic Web content has led to the fact that web sites include a plenitude of scripts (e.g., JavaScript, WebAssembly). Browser vendors thus optimize script execution as much as possible. For example, Just-in-Time (JIT) compilation of JavaScript code was a product of this demand, i.e., compiling JavaScript into native code at runtime. Yet running possibly adversarial native code has its own security implications. Multiple attacks have been proposed abusing JIT compilation for code injection [4, 27, 28, 35]. Consequently, browser vendors strive to restrict their JIT environments as much as possible. One such restriction, which our attack can evade, is sandboxing the generated JIT code such that it cannot read or write memory outside the permitted range. For example, browsers compare the object being accessed and its corresponding bounds. Any unsanitized memory access would escape such checks, and thus enable adversaries to read browser-based secrets or to leak data from the currently (or possibly all) open tabs, including their cross-origin frames. Even in the face of site isolation (i.e., one process per site), an adversary can still gather code pointers to break ASLR, e.g., to identify gadgets for privilege escalation (e.g., sandbox escapes) via code-reuse attacks.

In our threat model, we envision that the victim visits an attacker-controlled website. The victim's browser supports

[4] https://patchwork.kernel.org/patch/10150765/

JIT-compiled languages such as WebAssembly or JavaScript, as done by all major browsers nowadays. We assume that the browser either has a high precision timer, or the attacker has an indirect timer source through which precise timing information can be extracted.

## 5.2 WebAssembly-Based Speculation

Our second attack scenario is also based on the general principles described in Section 3. However, in contrast to the scenario in Section 4, victim and target share the same process. Furthermore, the attacker can now add (and therefore control) code that will be executed by the browser. To this end, an attacker just has to ship JavaScript or WebAssembly code, both of which will be JIT-compiled by off-the-shelf browsers. For illustration purposes and to have more control over the generated code, we focus on WebAssembly.

WebAssembly is a new assembly-like language, that is supported by all modern browsers (Edge from version 16, Chrome from 57, Firefox from 52, and Safari from 11.2)[5]. As it is already low-level, compiling WebAssembly bytecode into native code is very fast. The key benefit of WebAssembly is that arbitrary programs can be compiled into it, allowing them to run in browsers. The currently proposed WebAssembly specification considers 4 GiB accessible memory. This makes sandboxing the generated code easier. For example, in Firefox, usually a single register (`r15` in x86) is dedicated as the pointer to the beginning of the memory, called the WebAssembly heap. Consequently, all the memory is accessed relative to the heap. To restrict all possible accesses into the 4 GiB area, Firefox generates code that uses 32-bit x86 registers for encoding the offset. As a result, modifying a 32-bit register in x86-64 will zero the upper bits (e.g., `add eax, 1` will set the upper 32 bits of `rax` to 0).

For our browser-based attack, we leverage cyclic RSBs to trigger misspeculation. More precisely, we define two recursive functions $A$ and $B$, as shown in Figure 2. In step (1), $A$ calls itself $N_A$ times and then calls $B$ in step (2). In step (3), $B$ then calls itself recursively $N_B$ times, $N_B$ being the size of the RSB in this case. The function $B$ follows two purposes. First, being a recursive function, $B$ will overwrite all RSB entries with return addresses pointing to the instruction in $B$ following its recursive call. Second, $B$ includes code right after calling itself to leak sensitive data using speculative execution in the context of $A$. In step (4), $B$ returns $N_B$ times to itself, consuming all $N_B$ entries of the RSB. However, since the RSB is cyclic, all the entries still remain there. At this point, the return instruction in step (5) returns from $B$ to $A$ and triggers the first misprediction. In step (6), $N_A$ more returns will be executed, all of them mispredicting $B$ as the return target. The state of the RSB (shortened to $N = 4$) after each of these steps is also depicted in Figure 2.

## 5.3 Reading Arbitrary Memory

Compiling functions like those in Figure 2 into WebAssembly bytecode will result in arbitrary speculation of the generated

[5] https://caniuse.com/#feat=wasm

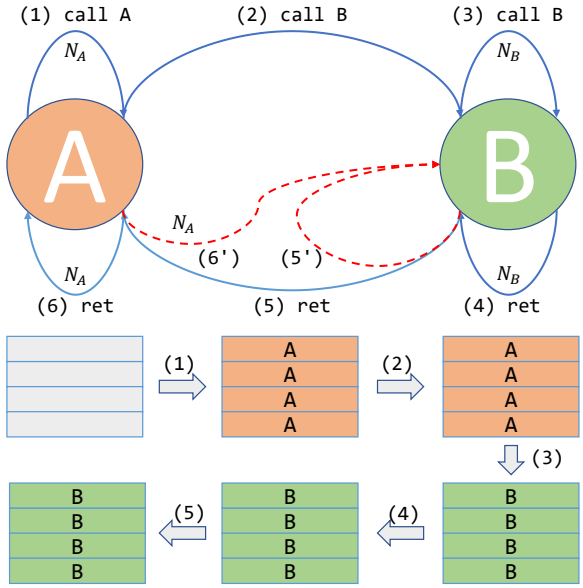



**Figure 2: Cyclic RSB with recursive functions `A` and `B`. Dashed arrows show mispredicted returns, solid ones actual returns.**

```
uint8_t *B(int rec_N) {
    unsigned char *loc;
    if (rec_N > 0)
        loc = B(rec_N-1);
        // <-- speculation
    return &bytearray[bytearray[loc[0]<<12]];
}
uint64_t A(int rec_N) {
    uint_64 res = 0;
    if(rec_N > 0)
        res += A(rec_N-1);
        // <-- speculation context
    else
        res += *B(16);
    return ADDRESS; // attacker-controlled value
}
```

**Listing 1: Arbitrary memory read in speculation**

native code. As a next step, we need to generate speculated code that leaks memory outside of the sandboxed memory region. The key observation here is that whenever we trigger a speculation, we execute instructions of one function in the context of another. For example, in the case of the functions `A` and `B` from Figure 2, after returning from `B` to `A`, code of function `B` will be executed, while the register values will stem from `A`. This confusion of contexts allows evasion of defenses that are in place to sandbox JIT-compiled code. As a simple example of context confusion, consider the following instruction accessing memory: `mov rbx, [rax]`. In normal execution, `rax` will always be sanitized, e.g., by using 32-bit registers for offset calculation. However, in speculation, triggered by another function (e.g., `mov rax, 0x12345678; ret`), `rax` can be set to an arbitrary value, thus reading the data at an arbitrary memory location.

```
1  B:   ...
2  call B
3  mov  al, [r15+rax] ; r15=heap, rax=ADDRESS
4  shl  eax, 12       ; eax=leaked byte
5  mov  al, [r15+rax] ; report back the byte
6
7  A:   ...
8  mov  rax, ADDRESS
9  ret  ; trigger speculation in A, at line 3
10 ; rax=ADDRESS will be used in speculation
```

**Listing 2: Disassembly of functions A and B (important parts)**

We will use these basic principles to generate speculative code that reads arbitrary memory contents—notably *outside* of the sandboxed region. To this end, we extend the general concept presented in Figure 2 and derive WebAssembly code that emits the required instructions after compilation (Listing 1). The key concept here stays the same: function `A` calls itself recursively `rec_N` times before calling `B`, which then recursively calls itself 16 times in order to fill up the RSB. After 16 returns from `B`, `A` will return `rec_N` times, each time triggering the speculation of instructions following the call statement in `B`, notably with the register contents of `A`.

The disassembly of the compiled functions `A` and `B` from Listing 1 are shown in Listing 2. After executing 16 returns from `B` (all with correct return prediction), execution reaches function `A`. In `A`, the return value (`rax`) is set (line 8) and the function returns (line 9). At this point, as RSB was underflowed by executing 16 returns, the return address is mispredicted. Namely, RSB's top entry will point to `B` (line 3). While the correct return address is being read from the stack, lines 3 onwards are being executed speculatively. The initial memory read operation (line 3) assumes a return value (`rax`) to be set by `B`, which is supposed to be sanitized. The base address, `r15`, is a fixed pointer to WebAssembly's heap, which is also assumed to remain the same. However, in our case, `rax` was set in `A` with the attacker-controlled value. This allows the attacker to read arbitrary memory relative to the WebAssembly heap. Lines 4–5 are then used to report the value back by caching a value-dependent page. That is, line 4 multiplies the read byte by 4096, aligning it to a memory page. The page-aligned address is then used in line 5, where the $N$-th page of WebAssembly's heap is accessed. After speculation, WebAssembly's heap can be probed to see which page was cached, revealing the leaked byte.

In our example, memory is leaked from an address relative to `r15`, which points to WebAssembly's heap. While the attacker-controlled offset (`rax`) is a 64-bit register and covers the entire address space, it might still be desirable to read absolute addresses, e.g., in case one wants to leak the data from non-ASLRed sections of the memory. This is easily doable with a simple modification of the WebAssembly bytecode. Instead of using a direct call (`call` opcode in WebAssembly), we can use an indirect call (`indirect_call`). The JIT compiler assumes that indirect calls might modify the `r15` register, and therefore restores it from the stack when the callee returns. Listing 3 shows the disassembly of Listing 2 with this simple modification that added lines 3 and 4. Line 3 restores the WebAssembly context register from the stack, while line 4 reads the heap pointer. However, in speculative execution with `A`'s context, `rsp` points to one of the arguments passed to `A`, which is controlled by the attacker. Thus, the attacker controls the value of the heap pointer, and, by setting it to 0, can allow absolute memory accesses.

```
1  A:   ...
2  call rcx           ; rcx=A, dynamically set
3  mov  r14,[rsp]     ; rsp=&argN of B
4  mov  r15,[r14+24]  ; r15= argN of B
5  mov  al, [r15+rax] ; al = argN[ADDRESS]
6  shl  eax, 12       ; eax=leaked byte
7  mov  al, [r15+rax] ; report back the byte
```

**Listing 3: Disassembly of the function B with indirect call**

## 5.4 Evaluation

We now evaluate the efficacy and precision of our attack when applied for reading arbitrary memory in browsers. We implemented our proof of concept in Firefox 59 on Windows 10 (version 10.0.16299), running on Intel's Haswell CPU (Intel® Core™ i5-4690 CPU @3.50GHz). It is worth noting that Firefox, together with other browsers, has recently reduced the precision of performance counters to 2 milliseconds as a defensive measure against caching-based side channels[6]. Given that finding alternative and more precise timing sources is out of the scope of this paper, we manually increased the performance counters to the old, more precise, state.

The main part of the proof of concept is a WebAssembly module that triggers the speculation. The number of speculatively executed returns is customizable in the module by choosing a different recursion depth of function `A` ($N_A$); we set it to 64 return predictions in our experiments. To feed back the speculatively read value, we used the WebAssembly heap of our module (from offset `0x4000` to avoid collision with other variables). To avoid hardware prefetching interference, we access the heap at a page granularity, i.e., `Heap + 0x4000 + value*4096`. After running the speculative code, we access the WebAssembly heap from JavaScript and measure the access times of each page. Leaking the entire byte will require walking 256 memory pages, which would be very slow. To optimize this, we split the byte in half (e.g., into `(value»4)&0xf` and `value&0xf`) and leak each nibble separately. This only requires scanning 16 pages per nibble, i.e., 32 scans per byte. This could be further optimized to 8 per-bit reads.

Our measurements worked in the following order: (a) Using JavaScript, write the same pangram from Section 4.4 into a 1024-byte buffer. (b) Compute the offset from the WebAssembly heap to the buffer containing the text. (c) Trigger the eviction of the feedback cache lines from the cache, by doing random memory accesses to the same cache line in JavaScript. (d) Call the WebAssembly module to speculatively execute

[6] https://developer.mozilla.org/docs/Web/API/Performance/now

the gadget from Listing 2, reading the value from the specified offset. (e) Scan the WebAssembly heap from JavaScript, and record the access times to each page. (f) Repeat steps (c)–(e) 100 times to increase the confidence in the leaked data. (g) Process the timings, recorded in (e), to find the page with the fastest average access time. (h) Return the index of the found page.

In our evaluation, we ran each 1024-byte reading iteration 10 times. Each iteration, on average, took 150 seconds, i.e., ≈55 bps reading speed—leaking a single byte thus takes 146 ms. Note that the main bottleneck in our measurements constitutes the code that evicts the cache lines (step (c)). In our proof of concept, we simply map an L3 cache-sized buffer in JavaScript and then access each page to the corresponding cache line. This approach can be further improved by initializing the eviction set prior to attack, and then walking the smaller set for eviction, as shown in [16].

To measure the accuracy, similar to Section 4.4, we used Levenshtein distance. The evaluation showed that the read byte was correct ≈80% of the time. Increasing the iterations or number of speculations will increase the precision, however at the expense of reading speed. We leave a more accurate and efficient implementation open for future work.

# 6 COUNTERMEASURES

Seeing the immense impact of this new attack vector, in this section, we discuss countermeasures against RSB-based speculative execution. Furthermore, we will describe the vendor reactions that followed our responsible disclosure process.

## 6.1 Possible Mitigations

In the following, we discuss possible mitigation techniques that can be employed to defend against our attacks.

*6.1.1 Hardware-based Mitigations.* A naive approach to get rid of all speculative execution problems in hardware is to simply disable speculative execution. That would, however, decrease performance drastically—making branch instructions serializing and forcing the execution of only a few instructions (between branches) at a time. Of course, one could try to enable speculative execution while prohibiting speculative memory accesses, or at least caching them in speculation. However, given that memory accesses are already a bottleneck for modern CPUs, blocking their speculative execution would incur a significant slowdown.

To counter our first attack in hardware, RSBs could be flushed by the CPU at every context switch, e.g., during system calls. Arguably, this will not impose any significant slowdown on performance, as the predictions after context switches will mispredict anyway in the vast majority of cases. In fact, hardware-assisted flushing will be more efficient than a software-based solution that requires several artificially introduced calls (as implemented right now). Hardware-backed RSB flushing would reliably prevent our cross-process attack, even in operating systems that do not flush RSBs themselves.

To counter our second attack, one could scrutinize the cyclic structure of RSBs and argue that switching to stack-based implementations mitigates the problem. However, even triggering misspeculation in a size-bound (16-entry) RSB is still possible, e.g., by using exceptions in JavaScript, or relying on bailouts from JIT-compiled code (cf. Section 3). We believe resorting to a combination of hardware/compiler solutions would allow more reliable security guarantees to defend against the second attack.

*6.1.2 Compiler-based Mitigations.* To study how our second attack can be defended against in software, it is natural to ask how JIT compilers can be hardened. Despite the fact that the general problem of speculative execution is caused by hardware, we can make our software environments more robust to these types of attacks. The importance of this issue was recently highlighted, when multiple researchers proposed severe microarchitectural attacks, breaking down the core assumptions we had about hardware-based isolation and execution models [22, 26].

For example, JIT compilers can aim to ensure that the code at call sites cannot be abused with any possible execution context. The safest bet would be to stop all speculative executions at call sites, e.g., by using already-proposed solutions, such as `lfence`/`mfence` instructions (e.g., adding an `lfence` instruction after every `call` instruction). Alternatively, one could introduce a modified version of a retpoline[7] that replaces all return instructions emitted by JIT compilers by a construct that destroys the RSB entry before returning:

```
  call return_new ;
speculate:        ; this will speculate
  pause           ; trap speculation until...
  jmp  speculate  ; ...return address is read
return_new:       ;
  add  rsp, 8     ; return to original addr.
  ret             ; predict to <speculate>
```

Alternatively, one could improve the memory access sanitization in JIT compilers. For example, JIT-compiled code could always use 32-bit registers as a natural way to constrain addresses to a 4 GiB range in memory—the current memory limit in WebAssembly. However, this by itself does not provide strong security guarantees. As we have shown in Section 5.3, the base addresses can also be modified in speculation. Having said this, WebAssembly is a relatively new addition to browsers, and new features are still being frequently suggested/developed. Each of these feature needs to be reevaluated in our context. In particular, the proposals to add exception handling and threading support to WebAssembly need to be carefully revisited. Built-in exception handling will allow RSB speculation even with a non-cyclic RSB, while adding WebAssembly support for threading might introduce new precise timing side-channels.

Regardless of the precise countermeasure, one can limit the overhead of compiler-based defenses. In particular, code parts that are guaranteed to be secure against all potential

[7] https://support.google.com/faqs/answer/7625886

abuses (e.g., possibly speculated code that does not have memory accesses) can be left as is.

*6.1.3 Browser-based Mitigations.* One of the directions that browser vendors take to mitigate side-channel attacks is to deprive the attackers of precise timings. Having no timers, adversaries cannot distinguish between cached and non-cached memory accesses, which is a fundamental requirement for cache- and timing-based side-channel attacks. Given the complexity of JavaScript environments, merely decreasing the `performance.now` counter (as done in most browsers) is insufficient. For example, Gras *et al.* [14] showed that `SharedArrayBuffer` can be used to acquire a timing source of nanosecond precision, while Schwarz *et al.* [34] studied different timing sources in modern browsers, ranging from nanosecond to microsecond precision. Approaches presented in academia thus aim to advance this protection to the next level. For example, the "Deterministic Browser" from Cao *et al.* [6] tries to tackle the issue by proposing deterministic timers, so that any two measurements from the same place will always result in the same timing value, thus making it useless for the attacker. In another study, Kohlbrenner and Shacham [23] propose Fuzzyfox, which aims to eliminate timers by introducing randomness, while also randomizing the execution to remove indirect time sources. Motivated by these works, and by the recent discovery of Spectre, browsers decreased their timing precision to 2 milliseconds, while also introducing a jitter, such that the edge thresholding technique shown by Schwarz *et al.* [34] is also mitigated.

Alternatively, browsers can alleviate the threats by stronger isolation concepts. In the most extreme case, browsers can add dedicated processes for each entity (e.g., per site) to enforce a strict memory separation and to isolate pages from each other. By doing so, one can guarantee that even with a severe vulnerability at hand, such as arbitrary memory read, adversaries are constrained to read memory of the current per-page process. Reportedly, modern browsers already consider this technique, e.g., Chrome uses a dedicated sandboxed process per domain [13], while Firefox plans to switch to a similar architecture in the near future. While isolation prevents cross-page data leaks, it still allows leaking sensitive pointers of the current process. This effectively evades ASLR and can be used as a preparation for exploiting potential browser vulnerabilities with code-reuse attacks.

## 6.2 Responsible Disclosure

Seeing the severity of our findings, we have reported the documented attacks to the major CPU vendors (Intel, AMD, ARM), OS vendors (Microsoft, Redhat) and browser developers (Mozilla, Google, Apple, Microsoft) in April 2018, and subsequently engaged in follow-up discussions. In the following, we will summarize their reactions and our risk analysis.

**Intel:** Intel acknowledged this "very interesting" issue of RSB-based speculative execution and will further review the attack and its implications. Their immediate advice is to resort to mitigations similar to Spectre is to defend against our attack (see Section 6.1); this is, however, subject to change as part of their ongoing RSB investigations that we triggered.

**Mozilla Foundation:** The Mozilla Foundation likewise acknowledged the issue. They decided to refrain from using compiler-assisted defenses, as they would seemingly require complex changes to JIT-compiled and C++ code. Instead, they aim to remove all (fine-granular) timers from Firefox to destroy caching-based feedback channels. Furthermore, they referred to an upcoming Firefox release that includes time jittering features similar to those described in FuzzyFox [23], which further harden against accurate timers.

**Google:** Google acknowledged the problem in principle also affects Chrome. Similar to Firefox, they do not aim to address the problem with compiler-assisted solutions. Instead, they also refer to inaccurate timers, but more importantly, focus on a stronger isolation between sites of different origins. Chrome's so-called Site Isolation prevents attackers from reading across origins (e.g., sites of other domains). However, as discussed in Section 6.1, this does not mitigate the problem that attackers can break ASLR with our attack technique.

**AMD / ARM:** Although we have not tested our attacks against ARM and AMD architectures, they acknowledged the general problem.

**Microsoft:** Microsoft has acknowledged the problem and is working on fixes, but has not disclosed technical details yet.

**Apple:** As of 07/23/2018, we have not heard back from Apple yet.

**Redhat:** Redhat was thankful for our disclosure and mentioned that the current Spectre defenses (especially flushing RSBs)—without considering RSB-based attacks—might otherwise have been removed by the kernel developers in the near future. In particular, Redhat mentioned that fixing RSB underflows will not fully solve the problems pointed out in our paper.

# 7 RELATED WORK

In the following, we discuss concepts related to our paper. First, we provide an overview of the two recent papers on speculative and out-of-order executions that are both closest to our work. We will then briefly summarize other similar work done in that area. Further, we look into microarchitectural attacks in general, discussing some notable examples. Finally, we also discuss proposed defense techniques and their efficacy against our proposed attacks.

## 7.1 Out-of-Order/Speculative Execution

Despite being implemented in CPUs since the early 90s, out-of-order and speculative executions have only recently caught the attention of security researchers. The initial discovery of security issues in both these concepts is attributed to a Google Project Zero researcher. Horn was the first to disclose the vulnerability to the vendors [18]. Concurrently, Fogh also inspected speculative execution and reported his (so far negative) findings [12]. Another concurrent work from Maisuradze and Rossow [29] also studied speculative execution, leading to a discovery of a side channel to derandomize kernel-level

ASLR, and reading arbitrary user memory in speculation. All these discovered issues were then distilled in two major research papers that nicely summarize the general threats.

On the one hand, Meltdown [26] (a.k.a. Variant 3) abuses a flaw in Intel's out-of-order execution engine, allowing adversaries to have access to data for a split-second without checking the privileges. This race condition in the execution core allows attackers to disclose arbitrary privileged data from kernel space. While it is the most severe, Meltdown is relatively easy to counter with both microcode updates and/or stronger separation between user and kernel space.

Spectre [22] (a.k.a. Variants 1 and 2), on the other hand, does not rely on implementation bugs in CPUs, and is therefore also significantly harder to tackle. Technically, Spectre uses a benign CPU feature: speculative execution. The problem, however, is that the branch predictor is shared between different processes and even between different privileges running on the same core. Therefore, in Spectre, adversaries are able to inject arbitrary branch targets into predictors and, by doing so, trigger arbitrary speculative code execution in victim processes (similar to our first attack). Furthermore, similar to our second attack, Spectre also proposed an in-browser attack to abuse the branch predictor in the same process, where mispredicting a branch path can lead to leakage of unauthorized data. Spectre is thus closely related to our approach. The difference is that we achieve similar attack goals by abusing a completely different prediction mechanism of the CPU: return stack buffers. While RSBs were already mentioned as a *potential* security risk [18, 22], it was so far unclear whether RSBs indeed pose a threat similarly severe as BTBs. Our work answers this open question and provides countermeasures to this problem.

Follow-up works naturally arose out of the general discovery of Meltdown and Spectre. In SgxPectre, for example, Chen *et al.* [8] showed that it is possible to use branch target injection to extract critical information from an SGX enclave. Similarly, in BranchScope, Evtyushkin *et al.* [10] studied the possible abuses of direct branch predictors to leak sensitive data from different processes, including SGX enclaves.

### 7.2 Cache-Based Side Channels

Given that accessing main memory in modern CPUs takes hundreds of cycles, current architectures employ multiple layers of caches. Each layer has various characteristics and timing properties, thus providing unique information as side channels. The key idea of cache side channel attacks is to distinguish the access times between cache hits and misses, revealing whether the corresponding data was cached or not.

Cache attacks can be divided into attacks on instruction and data caches. The attacks on instruction caches aim to leak information about the execution of the target program. For example, information from instruction caches can be used to reconstruct the execution trace of collocated programs [1–3, 7] or even VMs on the same machine [39].

In contrast, side channels from data caches reveal data access patterns in the target program, which again can be either a collocated program or a VM, depending on the level of the attacked cache. Per-core caches (e.g., L1 and L2) can be used as side channels against programs running on the same physical core. This has been shown to be useful for reconstructing cryptographic keys [36]. Conversely, shared or last-level caches (LLC) can be used to leak information, e.g., keystrokes or user-typed passwords, from any process running on the same CPU—notably even across VMs [33].

There are different ways to leak data via caches. Most notably, Flush+Reload [38] uses `clflush` to flush the required cache lines from the last-level cache shared with the victim. By measuring the same cache line, the attacker can detect whether the victim has accessed a certain cache line. Some variations of the Flush+Reload attack include Evict+Reload [17], which tries to evict the target cache line by doing memory accesses instead of the `clflush` instruction. This is important for cases where `clflush` cannot be used, e.g., in JIT code (cf. Section 5), or architectures without an instruction similar to `clflush` [25]. The inverse of Flush+Reload is Prime+Probe [31], where the adversary allocates (primes) the entire cache with its own data, and then triggers the execution of the victim. Then, the attacker will probe the caches to see which cache lines have been evicted (i.e., which cache lines have been accessed) by the victim.

### 7.3 Other Microarchitectural Side Channels

Given the complexity and abundance of optimizations, side channels in microarchitectures is not surprising anymore. Therefore, there are plenty of different attack techniques proposed by researchers, each of which target microarchitectural features of modern CPUs. For example, Evtyushkin *et al.* [9] use collisions in BTBs to leak information about the kernel address space, and by doing so derandomize the kernel-level ASLR (KASLR). Similar to Meltdown, which uses out-of-order execution to suppress exceptions, Jang *et al.* [20] use Intel's transactional synchronization extensions (TSX). By accessing kernel pages with TSX, depending on the type of the generated exception (e.g., a segmentation fault if memory is unmapped, or a general protection fault if the process does not have the privileges to access certain memory regions), the time to roll back the transaction differs. This constitutes a timing side channel that can be used to bypass KASLR, as an attacker can probe pages mapped in the kernel.

## 8 CONCLUSION

In this work, we investigate the security implications of speculative execution caused by return stack buffers (RSBs), presenting general principles of RSB-based speculative execution. We show that RSBs are a powerful tool in the hands of an adversary, fueled by the simplicity of triggering speculative execution via RSBs. We demonstrate that return address speculation can lead to arbitrary speculative code execution across processes (unless RSBs are flushed upon context switches). Furthermore, we show that in-process speculative code execution can be achieved in a sandboxed process, resulting in arbitrary memory disclosure.

# REFERENCES


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

[2] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New results on instruction cache attacks. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, Vol. 2010. Springer, 110–124.

[3] Onur Aciiçmez and Werner Schindler. 2008. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *CT-RSA*, Vol. 8. Springer, 256–273.

[4] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. 2015. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*.

[5] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).

[6] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. 2017. Deterministic Browser. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 163–178.

[7] Caisen Chen, Tao Wang, Yingzhan Kou, Xiaocen Chen, and Xiong Li. 2013. Improvement of trace-driven I-Cache timing attack on the RSA algorithm. *Journal of Systems and Software* 86, 1 (2013), 100–107.

[8] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2018. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv preprint arXiv:1802.09085* (2018).

[9] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–13.

[10] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 693–707.

[11] Agner Fog. 2018. The microarchitecture of Intel, AMD and VIA CPUs. http://www.agner.org/optimize/microarchitecture.pdf

[12] Anders Fogh. 2018. Negative Result: Reading Kernel Memory From User Mode. https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/

[13] Google. 2018. Site Isolation Design Document. https://www.chromium.org/developers/design-documents/site-isolation

[14] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. 2017. ASLR on the line: Practical cache attacks on the MMU. *NDSS (Feb. 2017)* (2017).

[15] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*. Springer, 161–176.

[16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer. js: A remote software-induced fault attack in javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 300–321.

[17] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches.. In *USENIX Security Symposium*. 897–912.

[18] Jann Horn. 2018. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.de/2018/01/reading-privileged-memory-with-side.html

[19] Intel. 2018. Intel 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

[20] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 380–392.

[21] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, 361–372.

[22] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[23] David Kohlbrenner and Hovav Shacham. 2016. Trusted Browsers for Uncertain Times.. In *USENIX Security Symposium*. 463–480.

[24] VI Levenshtein. 1992. On perfect codes in deletion and insertion metric. *Discrete Mathematics and Applications* 2, 3 (1992), 241–258.

[25] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices.. In *USENIX Security Symposium*. 549–564.

[26] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[27] Giorgi Maisuradze, Michael Backes, and Christian Rossow. 2016. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX.

[28] Giorgi Maisuradze, Michael Backes, and Christian Rossow. 2017. Dachshund: Digging for and Securing Against (Non-) Blinded Constants in JIT Code. In *Proceedings of the 15th Conference on Network and Distributed System Security Symposium (NDSS)*.

[29] Giorgi Maisuradze and Christian Rossow. 2018. Speculose: Analyzing the Security Implications of Speculative Execution in CPUs. *CoRR* abs/1801.04084 (2018). arXiv:1801.04084 http://arxiv.org/abs/1801.04084

[30] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. 2015. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1406–1418.

[31] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*. Springer, 1–20.

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

[33] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 199–212.

[34] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript. In *International Conference on Financial Cryptography and Data Security*. Springer, 247–267.

[35] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. 2015. Exploiting and Protecting Dynamic Code Generation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*.

[36] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23, 1 (2010), 37–71.

[37] Henry Wong. 2018. Microbenchmarking Return Address Branch Prediction. http://blog.stuffedcow.net/2018/04/ras-microbenchmarks

[38] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*. 719–732.

[39] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 305–316.