Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks

Johannes Wikner
ETH Zurich
kwikner@ethz.ch

Cristiano Giuffrida
VU Amsterdam
giuffrida@cs.vu.nl

Herbert Bos
VU Amsterdam
herbertb@cs.vu.nl

Kaveh Razavi
ETH Zurich
kaveh@ethz.ch

***Abstract*—There has been a substantial community effort in mitigating transient execution attacks in the web browser. Lightweight "catch-all" timer mitigations, deployed in all popular browsers, are presumed to raise the bar against these attacks. More heavyweight mitigations, such as pointer and array index masking are deployed more selectively to further make such attacks impractical. How secure are browsers with these mitigations taken together?**

In this paper, we show that a combination of new techniques allows an attacker to employ Spectre-RSB and leak sensitive information from browsers that deploy all these mitigations. First, we show that queuing up many transient loads during a single speculation window and using repeated measurements enable cache covert channels, even with jittery, millisecond precision timers. Second, we reverse engineer the newer RSB structure in Intel CPUs to find that deeper call stacks allow the attacker to hijack speculative execution for bypassing pointer and array index masking mitigations. Third, we show how an attacker can leverage memory massaging to reduce the entropy of the target secret's memory address. Our end-to-end exploit, *Spring*, combines these observations to leak an access token from an unmodified version of Firefox. Our disclosure effort has led to a deployed mitigation in the latest version of the Firefox browser.

I. Introduction

Transient execution attacks have shaken the security foundation of modern computing systems [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. Whereas significant effort has been devoted to protecting operating systems [11], [12], [13], [14], [15], hypervisors [16], [17] and trusted execution environments [3], [7], [8], the most important battleground for end users is arguably the browser. Particularly dangerous are transient execution vulnerabilities that allow attackers to leak sensitive information from JavaScript [1], [18], [4], [19], [20], [21].

To thwart these attacks, browser vendors have deployed a variety of mitigations. Some are lightweight and deployed everywhere, whereas others are more heavyweight and deployed more selectively. The various timer mitigations, now present in all the major browsers, are examples of the former [22], [23], [24], [25], [26], while masking techniques [27], [26] and site isolation [24], [28] are examples of latter. Site isolation is on-by-default only on Chrome due to its high cost and has recently been shown to be vulnerable when its consolidation policy co-locates mutually distrusting security contexts into the same process [20]. Without site isolation, browsers such as Firefox rely on timer mitigations and masking techniques for protecting against transient execution attacks. The question we ask in this paper is whether the combination of these mitigations provides an adequate protection against such attacks in the browser.

We show that the answer is unfortunately negative and an advanced attacker can bypass these mitigations using a number of new techniques. We study timer amplification techniques, which have previously been discussed [29] and demonstrated [19]. However, contrary to existing amplification efforts that solely focused on controlled repetitions [19], [29], [30], we present a new technique to further amplify the signal by queuing up multiple speculative memory loads in each round of a transient execution attack. Our approach is capable of bypassing all existing browser-based timer mitigations in a generic way, without relying on low-level details of cache replacement policies as done in previous work [19].

Besides timer mitigations, masking techniques aim to prevent unauthorized Out-of-Bounds (OoB) memory accesses during speculative execution. Whereas previous work on older Intel Haswell microarchitectures shows that masking can be bypassed through return-based speculation attacks from JavaScript [18], newer microarchitectures use a different return speculation mechanism and it is unclear whether similar attacks can still be applied. We reverse engineer previously undisclosed properties of the Return Stack Buffer (RSB) in more recent Intel CPUs and demonstrate that attackers can bypass pointer and index masking using carefully crafted (and amplified) return-based speculation attacks to enable Spectre-RSB attacks on more recent systems.

Even if we can hijack speculation through the RSB, it is still unclear where the secret is stored in memory. Address space layout randomization (ASLR) is a key mitigation against memory errors deployed in all software, including web browsers. Because transient execution attacks typically have a low bandwidth, especially in the browsers where precise timing information is lacking, a practical exploit needs to reduce the entropy of the secret's memory address. To build a complete exploit using only Spectre, we show how an attacker can trick the memory allocators in the browser and the operating system such that the target secret is always allocated at a predictable memory offset.

To show the practicality of these techniques, we present *Spring*, an end-to-end exploit that leaks data at the rate of around 3 bits per second with 90% success rate on Firefox running on a Kaby Lake Intel processor, and use it to leak an access token from the WebAssembly heap of a Microsoft Blazor [31] app in 8 minutes with 96% success rate. The exploit operates entirely inside an unmodified browser without any assumptions. In summary, our analysis shows that state-of-the-art mitigations are insufficient against transient execution attacks from JavaScript.

Contributions. We make the following contributions:

- We show that controlled repetitions and multiple speculative memory loads per repetition can amplify the signal beyond the timer granularity—demonstrating that timer crippling is fundamentally broken, even if the attacker does not know the low-level details of cache replacement policies of the CPUs (contrary to [19]).
- We reverse engineer the new RSB in recent Intel CPUs to revive Spectre-RSB attacks (e.g., *ret2spec*) on recent Intel CPUs. We use this new vector to enable speculative execution over architecturally impossible execution paths that result in bypassing masking mitigations.
- Using these insights, we build *Spring*, an end-to-end Spectre-RSB exploit that bypasses ASLR using memory massaging techniques to leak access tokens from apps in the Microsoft Blazor framework running on Firefox on a Kaby Lake CPU.
- We analyze potential mitigations. One of our proposed mitigations was picked up by the Mozilla security team and is currently deployed in Firefox.

Outline. In Section II we provide background on microarchitectural and speculative execution attacks as well as deployed browser mitigations. Section III outlines our threat model and Section IV gives an overview of our proposed techniques and challenges. In Section V, we discuss our timer primitive and in Section VI, we describe our efforts to reverse engineer the RSB behavior of modern microarchitectures. We present our end-to-end attack in Section VII, discuss countermeasures in Section VIII, and give an overview of related work in Section IX before concluding the paper.

II. Background

We discuss the background of microarchitectural attacks and defenses with a special focus on the browser.

A. Cache attacks

Cache attacks such as EVICT+TIME, PRIME+PROBE [32] or FLUSH+RELOAD [33] can spy on a victim process, for instance to leak cryptographic keys, by measuring the state of shared CPU caches after secret-dependent memory operations. EVICT+TIME and PRIME+PROBE work at the granularity of cache sets, and involves an attacker who first primes a number of sets in the n-way set associative cache by accessing an eviction set (a set of at least n addresses that map to the same cache set) for each cache set and then waits for the victim to access the cache in a secret-dependent manner and thereby evict cache lines belonging to the attacker. By probing the eviction sets again and timing the accesses, the attacker determines which cache set(s) the victim has accessed. Oren et al. [34] show that it is possible to perform PRIME+PROBE attacks from JavaScript in the browser. PRIME+PROBE attacks are generally applicable, but coarse-grained—operating at the granularity of cache sets. In contrast, FLUSH+RELOAD works at the granularity of cache lines, but only if the attacker and victim share memory. In that case, the attacker flushes specific cache lines that contain shared data or code, waits for the victim to access memory, and then identifies the access by checking which cache line is now in the cache.

B. Transient execution attacks

Out-of-Order (OoO) execution is a microarchitectural optimization that lets instructions execute ahead of time to maximize the usage of the on-core execution units and thereby the overall instruction throughput of modern CPUs. Moreover, in case of long-running instructions, such as memory loads that dictate the subsequent execution path (e.g., due to a branch), the CPU will predict the most likely outcome and continue execution there *transiently*. In case the prediction was wrong, the CPU discards the transiently executed instructions and continues execution at the correct location. Transient execution attacks leak sensitive information, typically from the CPU caches, as a result of wrongly transiently-executed instructions [35].

```
1  if (x < array.length) {
2    secret = array[x] & 0xff;
3    array2[secret  * 1024];
4  }
```

Listing 1: Spectre Bounds-Check Bypass (BCB).

Spectre. The original Spectre attacks [1] show that speculative execution can leak bytes from a victim process' address space via cache side channels. They are especially dangerous for end users since they can operate from JavaScript, exposing sensitive personal data, such as passwords and access tokens. These attacks use instructions, either in the sandbox implementation itself [36], [20] or in Just-in-Time (JIT) compiled code [1], [18], [30], to perform out of bound memory accesses. They speculatively execute these instructions as a result of a mispredicted conditional branch direction [1], [36], [30], [20] or a mispredicted return target [18].

Results of speculatively executed instructions are not exposed to the architectural state (visible to software) upon a misprediction. However, they do leave traces in the microarchitectural state. Spectre abuses such traces to allow attackers to leak arbitrary memory contents. For instance, mispredictions by the branch predictor may result in a Bounds-Check Bypass (BCB), as illustrated in Listing II-B. The attacker controls `x` and first trains the branch predictor to predict that the condition in Line 1 is true. When they later provide an out-of-bound `x`, the CPU will still speculate said condition to be true and read an attacker-chosen byte (Line 2). In Line 3, the victim uses this secret byte to access an element in the attacker-controlled `array2`. Using a cache attack, the attacker can determine the index of the element the victim accessed and, hence, determine the secret byte. Speculative side channel attacks usually leverage FLUSH+RELOAD to deduce cache state. However, since flushing cache lines explicitly is not possible in constrained browser environments, attackers use eviction sets instead.

However, the typical BCB variants of Spectre, shown Listing II-B and used in the original Spectre attack [1] and *leaky.page* [19], are limited to the maximum allowed array index offset of 4 GiB that is allowed by JavaScript. Hence, such speculation primitives cannot access memory outside the attacker's local memory area. Instead, effective Spectre attacks in the web browsers typically exploit a type confusion to achieve access to full 64-bit address space [9], [20], [18] but oftentimes lack other primitives necessary for an end-to-end attack: they patch the browser to access high precision timestamps (e.g., [18]) or assume ASLR is broken (e.g., [20]).

A different variant of Spectre attacks exploits return target speculation through Return Stack Buffers (RSBs) of Intel CPUs to trigger mispredictions [18], [37]. The *ret2spec* [18] attack abuses the fact that RSBs, used for return speculation, are circular buffers of limited size, resulting in overwrites if the call stack depth exceeds its capacity, leading to an underflow condition where stale return targets are re-used as the call stack depth decreases beyond the RSB capacity. This way, *ret2spec* triggers a speculative execution path that is architecturally impossible allowing for type confusion that gives speculative access to the entire address space. However, *ret2spec* is known to affect older Intel microarchitectures such as Haswell. It is commonly thought [37], [18], [38] that the more recent microarchitectures use the other prediction schemes for serving return targets when the RSB is empty, making *ret2spec* exploitation impractical.

Browser-based mitigations. In response to speculative execution attacks, browser vendors developed several mitigations to ① prevent unauthorized speculative accesses, ② remove sensitive data from the address space and ③ remove the means to measure the side channel [39].

① To prevent unauthorized speculative accesses, Firefox uses index and pointer masking, where the former makes sure that upper bits of array indices that would go speculatively go out of bound are always zero, and the latter applies an `XOR` pattern on `JSValues` so that JavaScript value types cannot be confused with one another (e.g., speculatively treating an array as an object). While such mitigations stop the original Spectre BCB attack, attacks through RSB speculation can still bypass them (e.g., *ret2spec*), but as discussed these attacks are assumed to be impractical on recent microarchitectures.

② In addition, to remove the sensitive data from the address space, the Firefox team has been working on strict site isolation, but it is still not shipped in the latest version of Firefox since several years in development [28]. In contrast, Chrome uses strict site isolation as a primary defense against speculation attacks [40]. The sandboxed process is only responsible for JavaScript runtime and rendering content. Firefox hence refer to it as the *content process*.

③ Major browsers try to prevent the measurement of side channels by crippling the timers. After the emergence of cache attacks in the browser, the HTML5 standard was updated to reduce the precision of timestamps to a minimum of 5 μs [41]. In addition, further web-based cache attacks [42], [43] have prompted browser vendors to reduce the timer precision further. For example, Chrome and Firefox reduced the timer resolutions to 0.1 ms [24] and 1 ms [44] respectively, and added random jitter to these timers to combat techniques for increasing timer precision [42], [45] For cross-origin isolated [46] websites, Chrome however removes timer mitigations because of their observation that timer mitigations can not stop Spectre attacks [29]. Furthermore, there is also a limitation as to how much timer precision can be reduced without potentially harming the end user experience [47].

Summary. Chrome removed their timer restrictions and fully rely on site isolation for websites that are considered cross-origin isolated. For Firefox, the assumption is that Spectre attacks are defeated with timer and pointer and index masking mitigations, and that *ret2spec* is prevented by modern microarchitectures.

III. Threat Model

We consider a common browser information disclosure threat model, with attacker-controlled JavaScript code loaded in the browser by a victim user on a modern Intel processor with all mitigations against transient execution attacks present. The victim is assumed to access a malicious website that covertly embeds the website that the victim is logged in to. This means that their (secret) access token will be used whenever they open that website. We assume that no further interaction is necessary from the user, but that the browser tab remains active throughout the attack. The goal of the attacker is to leak some sensitive information from the browser such as an access token, all without relying on any software bugs. We further assume that the secret information is placed in the same address space as the malicious website. While this is the case for browsers such as Firefox by default, which is the focus of this paper, other browsers that enable site isolation (e.g., Chrome) will require the exploitation of the consolidation policy to achieve address space co-location [20]. We perform the experiments on the latest version of Firefox at the time of this research (version 73) deploying all default mitigations such low-precision timers as well as array and pointer masking. Firefox patched the issue on version 79 as a result of our responsible disclosure and backported the fix to earlier versions as part of CVE-2020-15659.

IV. Overview and Challenges

We now formulate the high-level fundamental challenges addressed in this paper, as well our approach to overcoming them. In later sections, we introduce sub-challenges for each.

A. Crippled timers

Firefox counters cache side-channel attacks by reducing timer precision and adding random jitter to measurements.

Thus, it is impossible to distinguish a single cache hit from a miss with a single measurement, giving us our first challenge:

Challenge C1 Amplify the signal of existing attacks so that the signal can be measured with a low-resolution timer with random jitter.

In Section V, we overcome this challenge by introducing existing transient execution attacks in their amplified forms to measure a *chain* of sequential cache hits vs. misses. Our approach is similar, in spirit, to the batching strategy used by recent website fingerprinting approaches [48], the theoretical framework by McIlroy et al. [29], and the recent *Leaky.page* attack [19]. *Leaky.page* [19] relies on the insight that signal amplification can be achieved if the attacker knows of the cache replacement policy of the L1 cache. However, as we will show, such low-level microarchitectural knowledge is not needed. To amplify the signal, we exploit two observations. The first is that transient execution attacks can directly control the execution of the victim and are thus amenable to controlled repetition. The second is that we can issue multiple speculative loads in a single speculative execution window. Using these observations, we show in Section V that we can schedule several secret-dependent memory accesses and amplify the signal.

B. Spectre mitigations

Firefox mitigates Spectre Bounds Check Bypass and Branch Target Injection (Spectre-BCB and Spectre-BTI) [1] using pointer masking, whereas the other known browser-based Spectre variant (RSB) is assumed to no longer be amenable to practical exploitation on modern microarchitectures (i.e., Intel Skylake–onwards) [18]. But without a proper understanding of RSB's behavior in these microarchitectures, it remains unclear whether Spectre-RSB exploitation is a still a possibility.

Challenge C2 Reverse engineer the RSB behavior and determine whether its new variants hinder practical Spectre-RSB exploitation in the browser.

In Section VI, we overcome this challenge by conducting experiments to reverse engineer the behavior of RSB in different microarchitectures. Our results show a tag-based RSB implementation after the Haswell microarchitecture. This insight allows us to re-enable Spectre-RSB attacks by introducing deeper call stacks to match with the attacker-controlled RSB entry with the correct tag. Our new Spectre-RSB variant, called *Spring*, can leak information from the browser's address space by creating a type confusion through a stale RSB entry controlled by the attacker.

C. Software mitigations

To defend against memory error vulnerabilities, Firefox, like many other software systems deploys certain mitigations that can complicate exploitation with transient execution attacks. A prime example is ASLR which randomizes the addresses of objects inside the address space. It appears that bypassing ASLR is trivialized next to Spectre, hence completely omitted in recent end-to-end attacks [20]. Given that leakage is usually slower in the browser due to timer mitigations, a practical Spectre exploit should also bypass ASLR.

Challenge C3 Bypass ASLR and other mechanisms that might hinder practical *Spring* exploitation.

In Section VII, we show how we can leverage the predictable behavior of memory allocators to ensure that our target secret is placed at a predictable location in the browser's address space, effectively bypassing ASLR. Our end-to-end *Spring* exploit can hijack authentication tokens from a victim Microsoft Blazer app.

In Section VI, we overcome this challenge by conducting experiments showing how to again trigger RSB underflow conditions to enable *Spring*. With these experiments, we uncover previously undisclosed behavior of modern RSBs and show that attacker-controlled RSB mispredictions are still feasible in the browser.

V. Accurate Measurements with Inaccurate Timers

In this Section, we explain the signal amplification that breaks all timer mitigations in a fundamental way, and describe how we use it to re-enable Spectre attacks from JavaScript.

A. The Firefox timer mitigation

Firefox versions 60 and later have reduced the timer resolution to 1 ms, with random jitter on the timer ticks [44]. As shown in Figure 1, the timer ticks exactly once every interval of t_{res} = $1\,ms$, at a randomly chosen t_{mid} midpoint time. t_{mid} remains fixed during the interval, but will be selected afresh for the next interval. The timer jitter may generate a tick at any given time within the interval, effectively stopping all previous methods to bypass low-resolution timers that measure time by counting the number of loop iterations they can perform until the timer ticks [49], [42], [50].

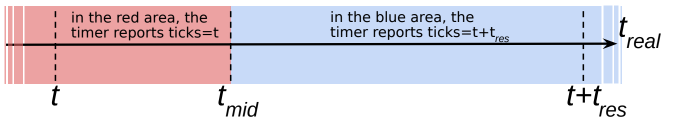

in the red area, the timer reports ticks=t
in the blue area, the timer reports ticks=t+t_res
t_real
t
t_mid
t+t_res

Fig. 1. The timer reports $time$ = t in the red area, until the actual time t_{real}, passes the randomly picked t_{mid}, after which it reports $time$ = t + t_{res}.

B. Bypassing timer mitigations

Following the theoretical model by McIlroy et al. [29], we bypass the timer mitigation by prolonging the measurements to cover a sequence of cached or uncached memory per measurement, rather than a single cache hit or miss. Our

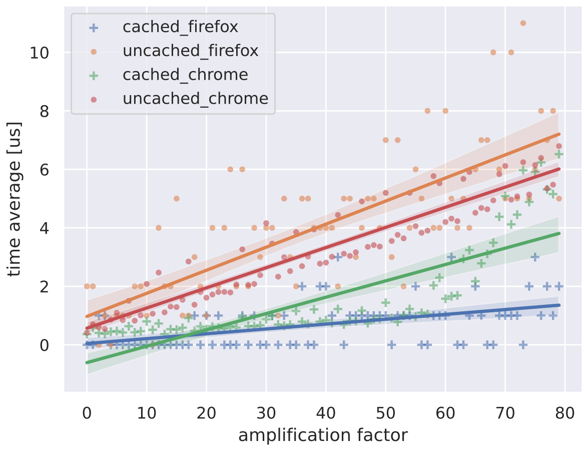


Fig. 2. The average (N=1000) access time (using browser's *performance.now()*) for cached and uncached memory on Firefox and Chrome. The regression lines shows that the difference increases with amplification factor until about 64, where the number of accesses in the sequence causes eviction of itself.

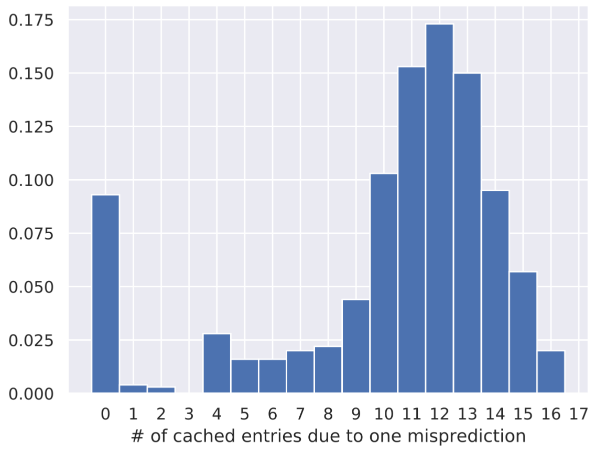


Fig. 3. Histogram of the fraction of runs with cache activity at multiple locations.

hypothesis is that, with a sufficiently long sequence of cache *misses*, the timer is more likely to tick than in the case of cache *hits*—despite jitter. An important practical question concerns the length of the sequence—the *amplification factor*. Make it too long, and parts of sequence may be evicted as it is being added to the cache, but if it is too short, the timer is less likely to tick for misses.

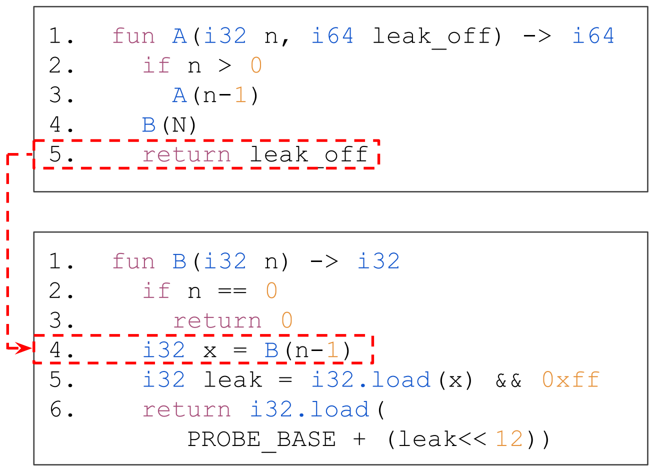
We show the validity of the theoretical model by means of an experiment that also provides an indication of the appropriate amplification factor for reliably distinguishing cache hits from misses. For this experiment, we modified Chrome and Firefox with additional functionality to allow JavaScript code to explicitly retrieve a data pointer of an `ArrayBuffer` object and then explicitly flush addresses from it via a `clflush` instruction, rather than by means of cache eviction, which would introduce additional noise. We conduct the experiment on both browsers to verify that it is browser agnostic. This allows us to isolate our experiment to timing only, but as we show in Section VII, cache evictions work as well, and for the rest of our experiments and our end to end attacks, we use an *unmodified browser*. To prevent OoO execution from loading several memory locations at once, we let the next cache line in the sequence depend on the previously-loaded cache line, a technique known as *pointer chasing*. Finally, to avoid prefetching, we use a random starting point of the sequence for every round.

In the experiment, we ① start the timer, ② access a sequence of uncached memory addresses, and ③ stop the timer. We repeat the same procedure but this time with cached memory addresses. We then flush the accessed memory to repeat both procedures for 1000 rounds and compute the average access time, as reported by the browser, for each sequence length. Figure 2 shows that with a few tens of reads, the cached and uncached accesses already form distinct clusters, allowing for low-noise measurements. In fact, even with just a single memory access (i.e., amplification factor of 1), there's already a slight difference. For instance, for an amplification factor of 50, cached and uncached accesses experience on average roughly 1 and 8 μs, respectively on Firefox and 1 and 5 μs on Chrome. The experiment not only confirms the hypothesis that we can deduce the cache state with the crippled timers in Chrome and Firefox, but implies in general for attacks that can encode a secret using a sequence of memory accesses that reducing the resolution of the timer is fundamentally unable to eliminate the side channel (although it may reduce its covert channel's bandwidth). We show next that Spectre has exactly this property.

C. *Signal amplification with Spectre*

Now that we can bypass the timer mitigations, we discuss two distinct methods for amplifying Spectre's signal from the browser and overcome Challenge C1 of Section IV. First, because the attacker controls code execution, they can trigger mispredictions to leak the same secret many times over—accessing different secret-dependent locations in every invocation. However, this is not as effective as it may seem, since the increased number of operations consequentially increases the chance of evicting recently accessed secret-dependent locations, which introduces additional noise in the measurements.

Second, the attacker may queue up several speculative loads to different secret-dependent locations from *a single* misprediction. Pipelined OoO CPUs can queue up multiple memory loads at once. We conduct an experiment to understand how many of such loads we can queue up in a single speculation window. To do this, we perform accesses to 100 different memory locations in random order in the speculative path. After triggering the misprediction, we check the corresponding locations to see how many of them are in the cache. To avoid OoO execution and prefetching from affecting



```
1.  fun A(i32 n, i64 leak_off) -> i64
2.    if n > 0
3.      A(n-1)
4.    B(N)
5.    return leak_off
```

```
1.  fun B(i32 n) -> i32
2.    if n == 0
3.      return 0
4.    i32 x = B(n-1)
5.    i32 leak = i32.load(x) && 0xff
6.    return i32.load(
          PROBE_BASE + (leak<< 12))
```

Fig. 4. Return target misprediction triggers a type confusion, where function B assumes that the returned value, x on line 4 of B is 32 bit and consequentially omits bounds checking when used in a subsequent load from the WebAssembly heap. Due to RSB underflow the control flow speculatively returns an attacker controlled 64 bit value on line 5 of A to B.

our measurements, we again use pointer chasing and a random starting point in the sequence. Additionally, we ensure that each access is to a different page since prefetchers typically operate only within memory pages [51].

The histogram in Figure 3 shows a distribution of the number of cache hits from 1000 runs, each consisting of a warm-up round, to populate instruction caches, and a real round for which we measure the results. The figure shows that we can queue up a substantial number (e.g., 12) of memory accesses in a single misprediction, enabling us to increase the amplification factor without the need for many rounds of speculative misprediction. We use this technique to improve the performance of *Spring* discussed in Section VI.

VI. Bypassing Masking with Spring

In this section, we show that RSB underflows that result in return target misprediction still occurs in recent microarchitectures. We conduct reverse engineering experiments to understand the behavior of return-based speculation and address Challenge C2 of Section IV. Using the knowledge gained from our reverse engineering, we demonstrate that with some modification the *ret2spec* attack [18], presumed dead in recent microarchitectures, it can be resurrected to leak information from arbitrary virtual memory addresses.

A. No underflow in new RSBs

The RSB is a LIFO (i.e., a stack) buffer of limited size $\hat{N}$, usually 16 entires, on Haswell microarchitecture and earlier [18], [52]. It is cyclic in nature: it overflows and evicts older entries after $\hat{N}+1$ `call` instructions without intermediate `ret` instructions. It may similarly underflow, and reuse stale return targets, after $\hat{N}+1$ `ret` instructions without intermediate `call` instructions.

ret2spec abuses this behavior to trigger a speculative type confusion in WebAssembly, enabling speculative out of bound reads of the WebAssembly heap as shown in Figure 4. Specifically, the attacker creates a call stack by calling a function A, that makes N recursive calls to itself and then calls function B, which in turn also makes N recursive calls to itself. At this point the RSB only contains return targets to B, causing mispredictions starting after N return instructions, where B returns to A, and continues to mispredict when A returns. The predicted return target from A becomes B, whereas in reality, B never called A. As shown in Figure 4, the attacker can arbitrarily manipulate input values to the speculatively hijacked instruction stream in B. Because these input values are unsanitized, they are bypassing pointer masking, hence enabling type confusion.

Unfortunately for the attacker, the type of RSB underflows that enable *ret2spec* no longer occur on Skylake and later processors [18]. We empirically verified that we could not trigger misprediction with *ret2spec* on an Intel CPU with a Kaby Lake microarchitecture. The question arises if Spectre attacks through RSB underflows are indeed no longer practical.

B. RSB underflow post-Haswell

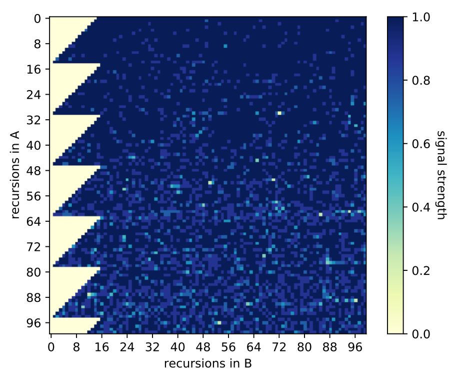
We repeat a similar experiment as the one in Section VI-A, but this time we increase the number of recursive calls in both A and B. We prepare the last return in A in such a way that, in case of a potential misprediction, we execute an instruction in B that loads a target memory location mem, as shown in Figure 4, as result.

Figures 5a and 5b show the results on a Haswell CPU on the left, which is known to be vulnerable, and Kaby Lake on the right. Surprisingly, after 64 subsequent returns in both A and B, we observe a signal from mem on Kaby Lake as we do for 16 on Haswell, suggesting that the Kaby Lake return prediction scheme under given circumstance, re-enables RSB-based prediction even if the RSB is underflowed. Moreover, by increasing the number of recursions in A or B by 16, the signal disappears, and reappears after 48 additional recursions.

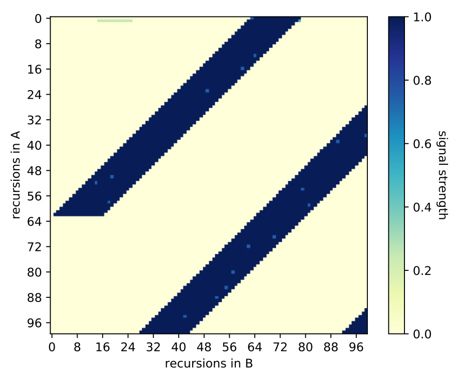
These results suggest that we are observing mispredictions from a new RSB structure. A possible implementation that exhibits similar behavior to what we observe is one that uses a 6 bit counter that increments on calls and decrements on returns [53]. The lower 4 bits determine the current top of the RSB stack and the upper 2 bits determine the current *tag*, which increments or decrements for every 16 calls or returns, respectively. Besides the return target address, each RSB entry stores also the tag, so that, upon a return, the CPU can compare the entry's tag to the current tag and use its return target address *only on tag match*.

C. Leaking arbitrary data with Spring

Our results show that it is again possible to force the CPU into address-independent misprediction through `ret` instructions on recent microarchitectures. We call this new variant of transient execution attacks *Spring* which requires additional returns and show how it can be used to leak arbitrary



(a) Haswell



(b) Kaby Lake

Fig. 5. Observing signals through additional returns in A and B. The dark color indicates a misprediction from A to B when returning from the *first* recursion in A. Haswell (a) exhibits a saw-tooth pattern because, for example, with 16 recursions in A and 0 in B, the deepest recursion in A, places A at the RSB entry that is re-used when returning from the first recursion, thus mispredicting A instead of B, which we don't measure here.

information in the browser. We evaluate the effectiveness and accuracy of *Spring* in Firefox version 73 (latest when we reported Spring to Mozilla) running on Linux 4.19. We first briefly discuss how we deploy the previously-discussed signal amplification techniques to *Spring* and how we set up our evaluation.

We select an amplification factor a = 32, meaning that our reload buffer, which we speculatively access secret-dependent memory in, must have a entries for every possible secret that can be leaked. Given that our configuration leaks a nibble (i.e., 4 bits) at a time, the number of possible secrets is N_v = 16 and the total entries of the reload buffee is thus $a \times N_v$ = 512.

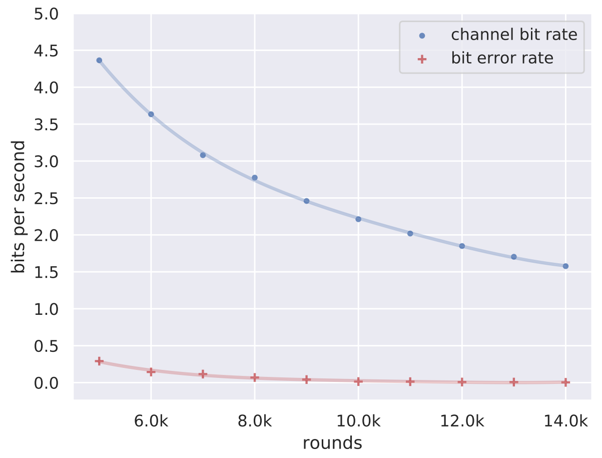


Fig. 6. *Spring* bandwidth (blue plot) and error rate (red plot) over rounds. One round includes 16 mispredictions with totally 32 secret-dependent memory loads meaning an amplification factor of 32. With 7000 rounds, Firefox has a bandwidth of 3 bit/s and 2% error rate.

A round consists of a warm-up invocation to improve the chances that the code resides in the instruction caches, and an actual run where we do 16 recursions in A and 64 in B. This leads to a series of 16 mispredictions from A to B, where we try to leak the secret to a secret-dependent locations. Since we repeatedly access the sequence, it is likely that its entirety eventually gets cached. We reload the corresponding sequence of locations of the N_v possible secrets and record how many times (if any) the timer ticks. By repeating this procedure thousands of times we are able to deduce which sequence is the overall fastest to access, which most likely corresponds to the secret.

Results. In the experiment, we try to leak a value that we have access to for measuring the quality of the signal. In Section VII, we instead leak secret values in our end-to-end exploits. Figure 6 shows the performance of our *Spring* attack. The error rate decreases as we increase the number of rounds that we attempt to leak the same secret. Consequentially, having more rounds also reduces the bandwidth of the covert channel. Since we control the address from which *Spring* leaks, these results show that we can leak information despite the crippled timer and masking mitigations in the latest version of Firefox. We confirm Kaby Lake and Coffee Lake Refresh microarchitectures running the latest microcode update as of this writing (`0xd6`) are affected by *Spring*.

VII. Exploitation with *Spring*

In this section, we show how *Spring* can be used to mount real-world browser exploitation. Again, we face several challenges to achieve reliable exploitation with Spectre and we discuss those first. We then present our primary end-to-end exploit that leaks a victim user's JSON Web Token [54] (JWT) from a Microsoft Blazor web application to hijack their session, compromising their account, using Firefox as our browser.

A. *Challenges for Reliable Exploitation*

Using Spectre in a practical attack scenario poses two challenges in addition to those discussed in previous sections.

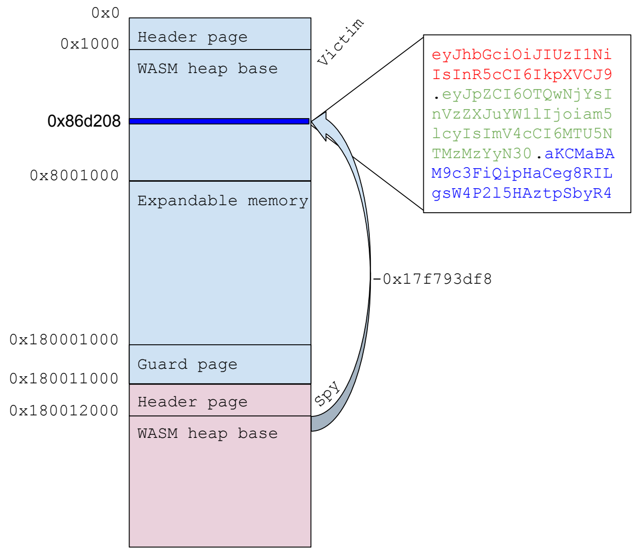


Fig. 7. The attacker (red) and victim (blue) heaps are allocated consecutively. The attacker speculatively reads memory from their own heap with an OoB offset (`-0x17f793df8`) that accesses the secret (JWT) in the victim heap.

First, the victim website needs to reside in the same address space as the attacker-controlled website. Second, even though *Spring* allows us to leak arbitrary memory, its throughput is limited, making extensive memory scanning for secrets impractical. Hence, we need to bypass ALSR so that our target secret is allocated at a predictable memory address.

Process co-residency on Firefox. We address the first challenge using `iframes`. Firefox distributes websites over several content processes. To ensure that the attacker and victim websites reside in the same content process, the attacker website renders the victim website in an `iframe` HTML element. Because iframes are in the same content process as the originating website, the attacker achieves address space co-residency with the victim website. This lets the attacker create a speculation gadget in their own address space to read OoB memory that resides in the victim's address space as seen in Figure 7.

Bypassing ASLR. The next challenge is to bypass ASLR. *Spring* leaks from addresses relative to its WebAssembly (WASM) heap. Instead of leaking the secret by first identifying its absolute address, we wish to allocate the secret at a predictable distance from the victim's WASM heap. On x64, WASM heaps spans a region of 6 GB that is split into an accessible and extensible area, with an additional 4 KB header page and a 64 KB guard page, as shown in Figure 7. There is a risk that memory regions have a gap in-between, but spraying a number memory allocations makes this unlikely. We found that spraying (c.f. Listing 2) 4096 `ArrayBuffer` objects of 64 KiB each, which we free after allocation, resulted in no gap between our attacker and victim memory regions. Larger memory regions are allocated without any address hint, allowing the OS kernel to select an appropriate address, which tends to be adjacent for larger allocations on Linux. Moreover, because memory is not randomized within the WASM heap, a cross-WASM heap leak is a promising target for our end-to-end exploit.

```
const o = {}
o.b = []
for(let i = 0 ; i <  0x1000; ++i) {
  o.b.push(new ArrayBuffer(0x10000))
}
delete o.b
```

Listing 2: Initially spraying ArrayBuffer objects allows attacker and victim WASM heaps to be consecutively allocated.

B. Attacking Microsoft Blazor

Microsoft Blazor [31] is the company's new web application framework where programmers write ASP.NET C# code that compiles into WASM to efficiently execute in the browser. Because application state is managed with C#, substantial parts of it resides on the WASM heap, which is not randomized. Different Blazor apps have different memory layouts, and some may not be susceptible to our exploit if they, e.g., do not store the secret in the WASM heap. We target a standard Blazor example app [55] for our attack.

First, the malicious website sprays `ArrayBuffer` objects as shown in Listing 2, then allocates the victim Blazor and spy WASM heaps consecutively using `iframes` that are hidden outside of the viewport. Because the victim is logged in on the Blazor app, the initial rendering results in several memory locations populated with the user's authentication details.

The Blazor app that we are targeting uses token-based authentication, a common method used in modern web applications as a CSRF-resilient alternative to standard HTTP cookies. As shown in Figure 7, the token follows the JWT specification and consists of three parts: a header, a payload and a signature, which are base64 encoded and concatenated as in Figure 8. The header specifies the cryptographic parameters employed for signing and verifying the token. The payload usually contains the issuing date and the authentication info. The signature is a computed hash over the header, the payload, and a server-side secret value. The token that we try to leak is 108 characters long.

To optimize the time to leak the entire token we observe that the header part is constant for all issued tokens and that the payload must resemble valid JSON. Thus, parts of the header can either be guessed or retrieved if the attacker can generate an access token of their own. Secondly, we can tolerate certain errors given that the leaked token must be a valid JSON object. For the signature, we can furthermore rely on that it must be valid base64.

With 24 k rounds, we can leak the token in less than 17 minutes without any errors and in 8 minutes with 96% accuracy. As we reduce the number of rounds, we can leak the token much faster with some errors. These errors can be detected due to invalid JSON or base64 and retried, or the attacker can instead bruteforce the remaining entropy.

```
          HEADER                                          PAYLOAD                                          SIGNATURE
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 .eyJpZCI6OTQwNjYsInVzZXJuYW1lIjoiam5lcyIsImV4cCI6MTU5NTMzMzYyN30 .aKCMaBAM9c3FiQipHaCeg8RILgsW4P2l5HAztpSbyR4
   "{"alg":"HS256","typ":"JWT"}"            "{"id":94066,"username":"jnes","exp":1595333627}"             HS256(header+payload+secret)
```

Fig. 8. The JSON Web Token consists of a header, a payload and a signature component. Each component is base64 encoded, reducing the entropy of the secret to characters representing valid base64 and the decoded payload be valid JSON.

VIII. Mitigations

Optimizing performance through speculative execution stipulates for occasional mispredictions. Unfortunately, completely removing speculative execution imposes tremendous performance penalties due to execution stalling. Practical mitigations instead focus on preventing only those speculative execution paths that can be used maliciously. The types of defenses we discuss target browsers, compilers and operating systems. We refer to Section IX for related work.

Site isolation. Since *Spring* leaks within the address space, site isolation effectively defeats its impact by isolating sensitive information in separate processes. Evidently, site isolation is hard to achieve in practice, not in the least for out-of-process `iframe` implementations, and it imposes grave performance costs. Despite years of development of site isolation for Firefox, even the latest shipped ESR (Extended Support Release) versions that most Linux users have still do not enable it. Site isolation has limitations, however: cross-origin resources are, by design, allowed to be brought into an attacker website's address space [46], and to potentially distrusting websites may be consolidated to improve performance [20].

Timer throttling with anomaly detection. We have already seen that imprecise timers slows down timing attacks, but do not stop them. However, they may result in less stealthy attacks as they limit the covert channel's bandwidth, making extensive scanning for sensitive data more difficult. In addition, at least for our example attack the execution patterns are sufficiently repetitive and to be detectable via performance counters. As always, anomaly detection is not without drawbacks, as it requires a careful trade-off between security and false positives.

Index masking. Existing compiler mitigations fail to address *Spring* as it triggers speculative execution paths that are impossible architecturally. Since functions returning 32 bit or smaller values will not set the upper bits of the result, it is therefore assumed safe to omit bounds checking. The assumption fails to consider that mispredicted return operations, from functions with 64-bit return values, may lead to control flows where the upper bits are set, enabling the leak. A possible mitigation is therefore to mask the upper bits of return values before they are used in memory operations. This way, all such speculative loads will reside within the guard pages of the WASM heap. The Firefox team opted for this approach as a short-term mitigation against *Spring*. We are however unsure whether there could be other architecturally impossible execution paths that can be exploited with RSB underflow attacks, resulting in type confusion, that are not covered by this mitigation. We leave that exploration for future work.

Hardened memory randomization. The *Spring* exploit relies on predictable allocation mechanisms of `mmap`, which allows us to leak the secret without breaking ASLR and without time-consuming scanning or pointer traversal procedures. We can therefore defend against this particular attack with hardened memory randomization of the operating system. Again, while doing so slows down the attack, it is unclear whether this is sufficient to make it impractical.

IX. Related work

In this section, we review related work on microarchitectural attacks and defenses in web browsers.

CPU cache timing attacks from the browser. CPU cache attacks have been discussed since the early 90s by Hu et al. [56] and Kocher [57] among others. Practical attacks, including PRIME+PROBE and EVICT+TIME, were introduced over a decade later concurrently by Osvik et al. [32] and Bernstein [58]. Yet another decade later, Oren et al. [34] demonstrated practical fingerprinting attacks using PRIME+PROBE in the browser. To resist such timing attacks, browser vendors have gradually lowered timer precision [23], [24], [25], [26], and JavaScript timers today are vastly less accurate than they were even five years ago. However, on certain conditions, browsers may increase the timer precision [46]. Prototype browsers such as FuzzyFox and DeterFox attempt to stop the attackers from using timers by further introducing random jitter [49] or making timing measurements deterministic [59]. A Chrome Zero enables a permission system to prevent side-channels in JavaScript [60]. Low-precision, fuzzy timers have since been adopted by all major browsers. Even so, Shusterman et al. [48] showed that attackers can still conduct cache attacks at a coarser granularity by sampling access times to a dataset that spans the entire LLC, and more recently even without JavaScript [61].

Moreover, it has been shown in multiple instances that, in JavaScript, it may be possible to reconstruct high-precision timers from low-resolution ones or indirectly [42], [50], [62]. In particular, these timers can acquire accurate timing information from caches to break ASLR [42] and to construct minimal cache eviction sets for Rowhammer [63], [64], [43], [65], and for transient execution attacks [20], [9], [19]. As mentioned, browser vendors responded by removing the features required to build such timers and by adding jitter [49], [59].

This work shows in practice that attackers *do not need* a high-precision timer at all, neither do they need knowledge of cache replacement policies [19]. Even the fuzzy, low-resolution timers are good enough if signal amplification is possible.

Variable instruction timing attacks. Andrysco et al. [66] demonstrate *pixel stealing* attacks that leak pixel values from cross-origin websites by applying filter effects `iframe` elements, a technique refined in follow-up work by Kohlbrenner et al. [67]. Rather than relying on the code-path timing of the original pixel stealing attacks [68], [69], [70], they rely on the variable instruction execution time of floating point operations. Instruction timing attacks appear to be a relatively unexplored area and potentially an interesting field for future research on amplification techniques. In contrast, in this paper we use different timing side channels and leak information accessed by the CPU under speculative execution.

Spectre and other transient execution attacks. Where the original Spectre attacks [1] use CPU caches as a side channel, later work showed that contention-based side channels can also be used [71]. Spectre attacks rely on a conditional branch or an indirect branch target to trigger incorrect speculations. In principle, Spectre could be operated from different security domains, including the browser. Horn described how speculative store bypass [72] could lead to an overwritten pointer being speculatively dereferenced to its previous value. Similarly, Maisuradze and Rossow [18] showed that return target mispredictions were also susceptible to unauthorized speculative accesses via RSB underflow conditions. Spectre and transient execution attacks using type confusion [18], [9], [20] allows for full address space access. In particular, SpookJs [20] combines type confusion with a weakness in Chrome's site isolation that consolidates distrusting websites into the same address space. In this paper, we show in addition that current, in-process mitigations in modern browsers are not sufficient to stop Spectre attacks. Recently proposed academic solutions such as SafeSpec, ConTExt, or SPECCFI do not help with existing systems as they require hardware changes [73], [74], [75] and annotations in software [74].

MDS-based attacks [4], [6], [5], [7], [8] take speculative execution attacks beyond branch prediction, showing that an assisting memory load (e.g., via a page fault) can result in a speculative load of stale data from internal in-flight buffers that may hold data from other execution contexts. While Intel responded by releasing a set of mitigations [76], patching the root cause has proven to be complicated; since their discovery in the first half of 2019, many new MDS-based leaks have surfaced [77], [78], [5], [7], [4]. Because of the high abstraction level of web programming languages (i.e., JavaScript and WebAssembly), prototyping and assessing susceptibility of MDS and other microarchitectural attacks in web browsers under different microarchitectures is a difficult and tedious process. To facilitate this, tooling for rapid prototyping is currently ongoing research [21]. MDS attacks, in particular, are relevant for future work, as they are capable of leaking across process boundaries that site isolation relies on.

X. Conclusion

In this paper, we investigated the effectiveness of current mitigations against speculative execution attacks from JavaScript in the browser. Through experiments, we reverse engineered the behavior of the return stack buffer (RSB) on recent Intel microarchitectures. We found that RSBs can be manipulated to still trigger return target mispredictions through RSB underflows, enabling Spectre-RSB attacks on recent systems. Bypassing browser-based mitigations, we then implemented Spectre on Firefox on recent CPUs to prove that timer mitigations and index masking are insufficient to stop speculative execution attacks that allow amplification through repetitions. We then used our new Spectre-RSB variant, called *Spring*, to build an end-to-end exploit. To achieve this, we leveraged memory massaging to place the target secret in a predictable location. Using the *Spring* exploit, we were able to hijack a victim's session on Microsoft Blazor apps with our *Spring* Spectre variant. We analyzed the possible courses of action for mitigating *Spring* and collaborated with Firefox developers on a patch against *Spring* that is currently deployed on the latest version of Firefox.

Responsible Disclosure

We disclosed the issues discovered in this paper to Mozilla in April of 2020. Since the disclosure, we have been collaborating with Mozilla engineers to develop a patch against *Spring* by sanitizing registers used for array accesses. The vulnerability was fixed in Firefox 79, ESR-68 and ESR-78 as part of CVE-2020-15659, and our internal tests shows that Firefox is since immune against *Spring*.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work was supported by Intel Corporation through the Side Channel Vulnerability ISRA and by Netherlands Organisation for Scientific Research through projects "TROPICS" and "Intersect". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," *arXiv preprint arXiv:1801.01203*, 2018.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[3] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *SEC*, 2018.

[4] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *S&P*, May 2019.

[5] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[6] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[7] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "Lvi: Hijacking transient execution through microarchitectural load value injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020, pp. 1399–1417.

[8] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," in *S&P*, May 2021.

[9] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021, pp. 1451–1468.

[10] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks," in *USENIX Security*, 2022.

[11] D. Gruss, D. Hansen, and B. Gregg, "Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer," in *; login: the USENIX Magazine*, 2019.

[12] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," *https://support.google.com/faqs/answer/7625886; Retr. 2020-04-18*, 2018.

[13] Linux, "kernel.org: L1TF - L1 Terminal Fault," https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/l1tf.html, accessed on 4.9.2020.

[14] ——, "kernel.org: MDS - Microarchitectural Data Sampling," https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/mds.html, accessed on 4.9.2020.

[15] Microsoft, "Protect your windows devices against speculative execution side-channel attacks," https://support.microsoft.com/en-us/help/4073757/protect-windows-devices-from-speculative-execution-side-channel-attack, May 2018.

[16] ——, "Windows guidance to protect against speculative execution side-channel vulnerabilities," https://support.microsoft.com/en-us/help/4457951/windows-guidance-to-protect-against-speculative-execution-side-channel, May 2019.

[17] Citrix, "Citrix Hypervisor Security Update," https://support.citrix.com/article/CTX251995, May 2018.

[18] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, New York, NY, USA, 2018.

[19] P. Röttger and A. Janc, "A Spectre proof-of-concept for a Spectreproof web," https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html, 2021, accessed on 5.6.2021.

[20] A. Agarwal, S. O'Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom, "Spook.js: Attacking chrome strict site isolation via speculative execution," in *43rd IEEE Symposium on Security and Privacy (S&P'22)*, 2022.

[21] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Rapid prototyping for microarchitectural attacks," in *USENIX Security Symposium*, 2022.

[22] "Re-enable sharedarraybuffer and atomics by default," https://bugzilla.mozilla.org/show_bug.cgi?id=1477743, 2018.

[23] L. Wagner, "Mitigations landing for new class of timing attack," https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/, 2018.

[24] Chromium, "Mitigating side-channel attacks," https://www.chromium.org/Home/chromium-security/ssca, 2018.

[25] M. E. Team, "Mitigating speculative execution side-channel attacks in microsoft edge and internet explorer," https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/, Jan 2018.

[26] F. Pizlo, "What spectre and meltdown mean for webkit," https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/, Jan 2018.

[27] J. de Mooij, "[meta] spectre bounds check mitigations," https://bugzilla.mozilla.org/show_bug.cgi?id=1430051; Retr. 2020-04-10, 2018.

[28] MozillaWiki, "Project fission," https://wiki.mozilla.org/Project_Fission, 2018.

[29] R. McIlroy, J. Sevcík, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *CoRR*, vol. abs/1902.05178, 2019. [Online]. Available: http://arxiv.org/abs/1902.05178

[30] N. Hadad and J. Afek, "Overcoming (some) spectre browser mitigations," https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/, Jun 2018.

[31] "Microsoft blazor," https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor, 2020.

[32] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," *IACR Cryptology ePrint Archive*, vol. 2005, p. 271, 2005.

[33] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014, pp. 719–732.

[34] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "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, 2015, pp. 1406–1418.

[35] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

[36] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Let's not speculate: Discovering and analyzing speculative execution attacks," *IBM Research Library*, 2018.

[37] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[38] "x86/retpoline: Avoid return buffer underflows on context switch," https://lore.kernel.org/patchwork/patch/871060, 2018.

[39] M. Miller, A. Fogh, and C. Ertl, "Wrangling with the Ghost: An Inside Story of Mitigating Speculative Execution Side Channel Vulnerabilities," ser. BlackHat, 2018.

[40] R. McIlroy, "Disable v8 untrusted code mitigations when site isolation is enabled," https://chromium.xieyaokun.com/chromium/+/3ba9207178ac7303393cb74dba77509755dbc4e4, 2018.

[41] I. Grigorik, "High resolution time level 2," *W3C recommendation*, 2019.

[42] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *NDSS*, vol. 17, 2017, p. 13.

[43] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 195–210.

[44] T. Ritter, "Set timer resolution to 1ms with jitter," https://bugzilla.mozilla.org/show_bug.cgi?id=1451790, 2018.

[45] D. Kohlbrenner and H. Shacham, "Trusted Browsers for Uncertain Times," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 463–480. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kohlbrenner

[46] E. Kitamura, "Making your website "cross-origin isolated" using COOP and COEP," https://web.dev/coop-coep/, accessed on 5.6.2021.

[47] M. Rejhon, "1440863 - unanticipated security/usability degradation from precision-lowering of performance.now() to 2ms," https://bugzilla.mozilla.org/show_bug.cgi?id=1440863, 2018.

[48] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *USENIX Security*, 2019.

[49] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 463–480.

[50] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: high-resolution microarchitectural attacks in javascript," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 247–267.

[51] R. Intel, "Intel 64 and ia-32 architectures optimization reference manual," *Intel Corporation, Sept*, 2019.

[52] A. Fog, "The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers," *Technical University of Denmark*, 2020.

[53] J. A. M. Stephan J. Jourdan and N. Jaisimha, "Us6898699b2 return address stack including speculative return address buffer with back pointers," https://patentimages.storage.googleapis.com/59/a3/6d/ec8cca6fc3bc01/US6898699.pdf, 2005.

[54] M. Jones, J. Bradley, and N. Sakimura, "Json web token (jwt)," *Internet Engineering Task Force (IETF)*, 2015.

[55] "Blazor real-world example app," https://github.com/torhovland/blazor-realworld-example-app, 2018.

[56] W. Hu, "Lattice scheduling and covert channels," in *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy(SP)*, vol. 00, 05 1992, p. 52. [Online]. Available: doi.ieeecomputersociety.org/10.1109/RISP.1992.213271

[57] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[58] D. J. Bernstein, "Cache-timing attacks on AES," The University of Illinois at Chicago, Tech. Rep., 2005.

[59] Y. Cao, Z. Chen, S. Li, and S. Wu, "Deterministic browser," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 163–178.

[60] M. Schwarz, M. Lipp, and D. Gruss, "Javascript zero: Real javascript and zero side-channel attacks." in *NDSS*, vol. 18, 2018, p. 12.

[61] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "{Prime+ Probe} 1,{JavaScript} 0: Overcoming browser-based {Side-Channel} defenses," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2863–2880.

[62] P. Vila and B. Köpf, "Loophole: Timing attacks on shared event loops in chrome," in *USENIX Security Symposium*, 2017.

[63] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

[64] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 987–1004.

[65] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "{SMASH}: Synchronized many-sided rowhammer attacks from {JavaScript}," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1001–1018.

[66] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 623–639.

[67] D. Kohlbrenner and H. Shacham, "On the effectiveness of mitigations against floating-point timing channels," in *USENIX Security*, 2017.

[68] P. Stone, "Pixel perfect timing attacks with html5," *Context Information Security (White Paper)*, 2013.

[69] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: timing attacks using css filters," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1055–1062.

[70] J. Forshaw, "Webgl-a new dimension for browser exploitation," *Online: http://www. contextis. com/resources/blog/webgl*, 2011.

[71] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 785800. [Online]. Available: https://doi.org/10.1145/3319535.3363194

[72] J. Horn, "speculative execution, variant 4: speculative store bypass," 2018.

[73] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317903

[74] M. Schwarz, R. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss, "Context: Leakage-free transient execution," in *Proc. of NDSS*, San Diego, CA, Feb 2020.

[75] *SPECCFI: Mitigating Spectre Attacks using CFI Informed Speculation*, San Diego, CA, May 2020.

[76] Intel, "Microarchitectural data sampling / cve-2018-12126 , cve-2018-12127,cve-2018-12130,cve-2019-11091 / intel-sa-00233," *Security software guidance*, 2019.

[77] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Addendum to RIDL: Rogue in-flight data load," in *S&P*, Oct. 2019.

[78] ——, "Addendum 2 to RIDL: Rogue in-flight data load," in *S&P*, Mar. 2020.