iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices

Jason Kim
Georgia Tech
Atlanta, USA
nosajmik@gatech.edu

Stephan van Schaik
University of Michigan
Ann Arbor, USA
stephvs@umich.edu

Daniel Genkin
Georgia Tech
Atlanta, USA
genkin@gatech.edu

Yuval Yarom*
Ruhr University Bochum
Bochum, Germany
yuval.yarom@rub.de

Abstract

Over the past few years, the high-end CPU market has been undergoing a transformational change. Moving away from using x86 as the sole architecture for high performance devices, we have witnessed the introduction of computing devices with heavyweight Arm CPUs. Among these, perhaps the most influential was the introduction of Apple's M-series architecture, aimed at completely replacing Intel CPUs in the Apple ecosystem. However, while significant effort has been invested analyzing x86 CPUs, the Apple ecosystem remains largely unexplored.

In this paper, we set out to investigate the resilience of the Apple ecosystem to speculative side-channel attacks. We first establish the basic toolkit needed for mounting side-channel attacks, such as the structure of caches and CPU speculation depth. We then tackle Apple's degradation of the timer resolution in both native and browser-based code. Remarkably, we show that distinguishing cache misses from cache hits can be done without time measurements, replacing timing based primitives with timerless and architecture-agnostic counterparts based on race conditions. Finally, we use our distinguishing primitive to construct eviction sets and mount Spectre attacks, all while avoiding the use of timers.

We then evaluate Safari's side-channel resilience. We bypass the compressed 35-bit addressing and the value poisoning countermeasures, creating a primitive that can speculatively read and leak any 64-bit address within Safari's rendering process. Combining this with a new method for consolidating websites from different domains into the same renderer process, we demonstrate end-to-end attacks leaking sensitive information, such as passwords, inbox content, and locations from popular services such as Google.

1 Introduction

The ubiquity of computing devices has resulted in the ever-increasing popularity of web browsers. Indeed, users now consume and create content directly through the browser, using it as a main access point to cloud-backed services and infrastructure. This usage pattern makes the browser one of the most important components of a user-facing computer system, with the browser's address space routinely containing sensitive information such as login credentials, emails, documents, pictures, etc.

With browsers routinely loading and executing JavaScript code from untrusted sources, many side-channel attacks have used the browser as a convenient starting point for compromising the system through its underlying hardware [1, 2, 13, 17, 18, 20, 22, 40–42, 49, 51, 56, 61, 67, 68, 80]. As most attacks rely on the ability to measure time of microarchitectural events, vendors in turn have attempted to harden browsers against side channels by severely degrading available timer resolution [52, 55, 78], as well as limiting the use of `SharedArrayBuffer`s in some cases [14] to prevent attackers from crafting a high-resolution timer.

Next, the ever-changing computing landscape has recently resulted in the introduction of high-end Apple silicon devices, aiming to compete with their x86 counterparts. While there is a plethora of works analyzing browser-based side channels on Intel and AMD architectures, the Apple ecosystem remains poorly understood despite its popularity [20, 50, 63, 67, 76]. With the growing popularity of Apple's M-series, we investigate the following main questions:

Is the ability to measure time truly critical for mounting microarchitectural side-channel attacks? In particular, how can attackers mount transient execution attacks inside the browser, potentially without relying on any timing primitives? Finally, how resilient are Apple devices to side-channels? And how can adversaries exploit them?

1.1 Our Contributions

In this paper, we present iLeakage, a speculative type-confusion attack that can extract information from Apple's Safari browser. In particular, we can defeat Apple's low-resolution timer, compressed 35-bit addressing, and value poisoning countermeasures, allowing us to read any 64-bit address within the address space of Safari's rendering process. Combining this with a new technique for consolidating websites from different domains into the same renderer process, we craft an end-to-end attack capable of extracting sensitive information (e.g., passwords, inbox content, locations, etc.) from popular services such as Google. Finally, we note that Safari / WebKit is the only browser engine permitted on iOS devices regardless of web browser app. This makes nearly all smartphone and tablet devices made by Apple susceptible to our attack.

Next, as little is known about transient execution attacks on Apple devices, before constructing our attack we must first develop common (and often timerless) side-channel primitives on Apple silicon. Given the popularity of Apple hardware, and the timerless nature of our constructions, these might be of independent interest.

Investigating Cache Layout. We begin by investigating the topology of the cache hierarchy on Apple CPUs. As this information is not available via unprivileged performance counters, we design a set of experiments for recovering the total size, associativity, cache line size, and inclusiveness of Apple's L1-I, L1-D, and L2 caches.

Measuring Speculation Window. We then proceed to measure the number of instructions Apple CPUs can execute under speculation, before the CPU discovers a branch misprediction. Here

*Work partially done while affiliated with the University of Adelaide.

Anonymous Submission to ACM CCS 2023, Due XX Jan 2023, Copenhagen, TBD
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

we show that modern Apple hardware has deep pipelines, with speculation windows running as long as 300 cycles.

Distinguishing Cache Misses from Cache Hits. Our next task is to distinguish cache hits from misses. While this can be achieved typically by measuring time, Apple has limited the clock resolution in both native and browser-based environments. Noting that cache hits and misses result in different sizes of speculation windows, we present a methodology for replacing timing based primitives with timerless counterparts based on race conditions. Using this approach, we are able to reliably distinguish hits from misses using the native 42 ns timer, Safari's 1 ms timer, Tor's 100 ms timer, and even without the use of timers altogether. To the best of our knowledge, this is the first browser-based demonstration of timerlessly distinguishing cache misses from cache hits, and the first primitive that does not rely on ISA-specific features.

Constructing Eviction Sets. We then shift to constructing cache eviction sets. Here, we improve prior work by Vila et al. [77], presenting a new group testing and backtracking approach which allows the algorithm to converge within minutes even without the use of timers. As degrading the timer resolution is often seen as a countermeasure to eviction set construction, our timerless technique might be of independent interest.

A Timerless Browser-based Speculative Attack. We combine all of our above-constructed primitives into a timerless Spectre v1 gadget Proof of Concept (PoC). At a high level, we achieve this by replacing the cache timing-based method of leaking secrets under speculation with our gadget for timerlessly distinguishing cache misses from cache hits. Here, we show that our attacks have near perfect accuracy, across Safari, Firefox and Tor.

Mounting Transient Execution Attacks in Safari. Next, we proceed to mount speculative side-channel attacks on the Safari browser. We begin by abusing Safari's site isolation policy, demonstrating a new technique that allows the attacker page to share the address space with arbitrary victim pages, simply by opening them using the JavaScript `window.open` API. We then construct in-browser eviction sets, side stepping Safari's timer mitigations. Finally, we bypass Apple's compressed 35-bit addressing and value poisoning countermeasures using speculative type confusion, allowing the attacker to craft and dereference arbitrary 64-bit pointers. Here, we show that assumptions used when designing architectural memory safety approaches do not hold true under speculation.

Leaking Sensitive Data. As a final contribution, we demonstrate the security implications of the techniques we developed and present end-to-end use cases of our attacks. More specifically, we show how an attacker webpage can open the target page and subsequently read information from it. Empirically demonstrating this, we show recovery of inbox contents and text messages. Next, applying this technique to password managers such as LastPass, we exploit the auto-fill feature to leak the target's Google credentials. We note that this attack is practical: it depends only on the target visiting the attacker's website, and runs to completion on Macs, iPhones, and iPads in their default configurations.

Summary of Contributions. We contribute the following:

- We study the cache topology, inclusiveness, and speculation window size on Apple CPUs (Section 4.1, 4.2, and 4.5).
- We present a new speculative-execution technique to timerlessly distinguish cache hits from misses (Section 4.3).
- We tackle the problem of constructing eviction sets in the case of low resolution or even non-existent timers, adapting prior approaches to work in this setting (Section 4.4).
- We demonstrate timerless Spectre attack PoCs with near perfect accuracy, across Safari, Firefox and Tor (Section 4.6).
- We mount transient-execution attacks in Safari, showing how we can read from arbitrary 64-bit addresses despite Apple's address space separation, low-resolution timer, caged objects with 35-bit addressing, and value poisoning countermeasures (Section 5).
- We demonstrate an end-to-end evaluation of our attack, showing how attackers can recover sensitive website content as well as the target's login credentials (Section 6).

1.2 Ethics and Artifact Availability

We initially disclosed our findings to Apple's product security team on September 12, 2022. Apple has acknowledged the issues, requesting an embargo on the paper's contents. We have actively discussed countermeasures with Safari's security team and maintained contact with Apple's head of product security. Our discussion has resulted in Apple refactoring Safari's multi-process architecture significantly, which we detail in Section 7. At the time of writing, these changes are under active development. We will publish the reverse-engineering code used in Section 4 to facilitate characterization of future Apple devices. Finally, we will coordinate the release of proof-of-concept attack code from Section 5 with Apple.

2 Background

2.1 Caches and Cache Attacks

To bridge the increasing performance gap between the CPU and main memory, the CPU contains small buffers called *caches*. These exploit locality by storing frequently and recently used data to hide the memory access latency. We primarily focus on recent Apple Silicon CPUs, which feature a heterogeneous core design with one or more clusters of high-performance P-cores and energy-efficient E-cores. While each core type differs in its cache organization, they have a private L1 cache and a shared L2 cache per core cluster.

Cache Associativity. Typically, these caches are divided into multiple cache sets that can host up to a certain number of cache lines or *ways* or *associativity*. Part of the virtual or physical address is then used to map a cache line to its respective cache set, where *congruent* addresses are those that map to the same cache set.

Cache Attacks. By monitoring the target's cache accesses, an attacker can infer secret information from the target in a shared physical system. Previous works proposed many different techniques to perform such attacks, most notably Flush+Reload [23, 24, 84] and Prime+Probe [15, 30, 34, 47, 56, 58, 60, 72, 77]. Most aforementioned cache attacks on Intel CPUs have targeted the last-level (L3) cache, as it is shared among all cores and an L3 cache miss incurs measurable spikes in latency. We instead target the L2 cache on Apple CPUs, but for the same rationale.

2.2 Speculative and Out-of-Order Execution

Rather than following the strict program order, modern processors execute instructions as soon as the required data is available, a concept called out-of-order execution. Furthermore, to handle branches whose condition is yet to be resolved, the processor attempts to predict the branch condition, speculatively executing instructions along the corresponding path. When the condition is eventually computed, if the branch has been mispredicted, the processor will revert the speculatively executed computation and will proceed to execute the correct path instead.

The discovery of Spectre [40] and Meltdown [46] showed that speculative execution has security implications. Specifically, while the processor can reverse all architectural effects resulting from incorrect speculation, microarchitectural effects such as cache and predictor states are not restored. Transient-execution attacks leverage this partial state reversal to extract information not available otherwise to the attacker, violating the separation between many mutually-distrusting hardware-backed security domains [6, 8–10, 16, 19, 28, 31, 35, 43, 45, 46, 48, 49, 62, 69–71, 73–75, 79, 81].

Finally, in concurrent work, Katzman et al. [33] show that speculative execution can even be used to build logic gates that operate on the cache state, which in turn can amplify cache attacks.

2.3 Side Channel Attacks on Apple CPUs

While CPUs made by Intel and AMD have received a generous amount of side channel research attention, much less is known about side channel vulnerabilities in Apple's Arm-based A- and M-series CPUs. More specifically, Shusterman et al. [67] demonstrate cache capacity attacks on the Apple M1 while Leaky.page [20] and Hetterich et al. [27] established the feasibility of Spectre v1 exploits on these machines. More recently, the m1racles [50] attack demonstrated a cross-process covert channel due to insufficient access control to system registers.

Next, Augury [76] demonstrated the existence of data-dependent memory prefetching (DMP) on the Apple M1, allowing attackers to bypass some Spectre countermeasures as well as derandomize the kernel's ASLR. Finally, PACMAN [63] shows how attackers can forge kernel pointer authentication codes (PAC) from userspace to bypass pointer authentication on M1 CPUs.

2.4 Side Channel Attacks in Web Browsers

Side-channel attacks have also been demonstrated using browser-based code (e.g., JavaScript and WebAssembly). Indeed, cache attacks running within browser tabs have been used for cryptographic key extraction [18], monitoring user activity [56, 67, 68], and even to cause Rowhammer [13, 17, 22]. Next, the data dependency of floating point instructions was also exploited in the browser, in the form of pixel stealing attacks [2, 41, 42]. Finally, early transient execution vulnerabilities have also been demonstrated in browsers [40, 49, 51], prompting large mitigation efforts.

Transient Execution Attacks in Chrome. Following the demonstration of the original Spectre [40] attack in JavaScript, Google attempted to harden Chrome against transient execution attacks. This effort led to the deployment of site isolation, where websites originating from different eTLD+1 domains are rendered in different address spaces (see Appendix A for a more complete discussion).

Leaky.page [20] shows that this hardening is necessary, as attackers can reliably mount Spectre-based attacks on modern versions of Chrome, albeit being limited to reading specific 4 GB heaps in Chrome's 64-bit address space. Finally, Spook.js [1] overcomes this 4 GB restriction, allowing attackers to read browser secrets assuming the attacker and victim pages share the same eTLD+1 domains.

In this paper, we show that disclosure of browser secrets is still possible and practical even with the degradation or removal of timing sources, through the timerless nature of our primitives. More specifically, we eliminate the dependency on `SharedArrayBuffer`-based timing primitives of Spook.js, shown to have approximately 5 ns resolution [66]. Remarkably, timers with 5 ns resolution are unavailable even to native code in Apple CPUs, let alone JavaScript in browsers: we detail this landscape in Section 2.5.

Transient Execution Attacks in Firefox. Transient-execution attacks have also been demonstrated within modern versions of the Firefox web browser. Ragab et al. [61] showed how incorrect results generated transiently by floating point units in Intel and AMD machines can lead to type-confusion attacks, allowing the attacker to dereference arbitrary 64-bit pointers. More recently, the Spring attack [80] exploits mispredictions in the return stack buffer (RSB) to again demonstrate type-confusion attacks on Firefox.

2.5 Timing Sources on Apple CPUs

A common requirement for mounting microarchitectural attacks is the capability of measuring the execution time of different instructions. We now summarize the state of time measuring capabilities in the Apple and web browser ecosystems.

Timers in Native Environments. Both macOS and iOS allow privileged native code to read a cycle-accurate timer which is accessible via the `kperf` API [5, 44]. However, unprivileged code can only access the timer provided by `mach_absolute_time()`, which PACMAN reported to have a resolution of 24 MHz [63], or 42 ns. Such a resolution is intractable for microarchitectural side channels, as most instructions executing in a single-digit number of cycles while cache accesses completing in the dozens.

Timers in Web Browsers. With timers being a critical component for side channel attacks, web browsers typically further restrict the timer resolution available to browser-based JavaScript code [52, 55, 78]. Here, Google Chrome provides a timer resolution of 100 μs, while Mozilla's Firefox and Apple's Safari provide a 1 ms timer. In an effort to resist fingerprinting of any sort, Tor Browser further coarsens the timer to 100 ms.

In older browser versions, one could craft a timer with 5 ns resolution [66] using the `SharedArrayBuffer` API in JavaScript, in a manner similar to a counting thread. However, following the use of high-resolution timers in cache covert channels as part of the Spectre and Meltdown attacks publicized in January 2018, all major browser vendors disabled `SharedArrayBuffer` [14].

Yet, `SharedArrayBuffer` was conditionally re-enabled in all major browsers by December 2021 for webpages served with cross-origin isolation headers [14], as these aim to mitigate Spectre by preventing webpages from loading cross-origin content not explicitly allowed by the server [39]. See Section 5.1 for implications of this condition on the Safari browser.

2.6 Timer-Friendly Covert Channels

Transient-execution attacks typically leak values obtained during incorrect speculation via microarchitectural covert channels. However, as browser vendors heavily restrict the timer resolution available to JavaScript code [52, 55, 78], we require a "timer-friendly" covert channel, which can reliably transmit values from the speculative domain even in the case of degraded timers.

pLRU-based Channel. Early works have resorted to timing techniques based on the clock edge [21, 41, 66] to overcome this challenge. More recent work by Google [20] abuses the pseudo least recently used (pLRU) eviction strategy of L1-D caches to construct a particularly stable covert channel on Intel and Apple CPUs.

For the remainder of this paper, we abstract the pLRU covert channel as providing four primitives. `plru.init()` initializes the covert channel, and `plru.transmit()` transmits a bit via some microarchitectural state. `plru.traverse()` traverses an L1-D cache set in a way that this routine will take a longer time to execute if `plru.transmit()` had previously transmitted a 1 bit, than if a 0 bit was transmitted or no transmission happened at all. Notably, this timing difference can be amplified significantly to be observable with a low-resolution timer by performing sufficiently many traversals over the L1-D cache set, thereby allowing us to use `plru.traverse()` to recover the transmitted bit.

Indeed, Listing 1 outlines our recovery procedure, which we call `plru.receive()`. It first times `plru.traverse` with the low-resolution timer (Line 2). Then, leveraging the amplification property, the transmitted value is recovered by comparing `plru.traverse`'s runtime against a coarse-grained calibrated threshold in Line 3.

```
1  function plru.receive() {
2      runtime = badtimer.time(plru.traverse());
3      return runtime > threshold ? 1 : 0;
4  }
```

Listing 1: Details of the `plru.receive` function.

Moreover, the pLRU strategy has also been shown to amplify the outcome of certain race conditions [83], resulting in the ability to construct eviction sets in the presence of 5 μs timers on `x86` CPUs. Finally, in concurrent and independent work, Purnal et al. [59] show an improved `plru.traverse` routine over an L2 cache set whose elements are congruent to the L1-D cache set on Intel CPUs, increasing the maximum amplification of the pLRU-based channel from 500 μs to 5 ms.

Timerless Channels. Disselkoen et al. [15] use Intel's Transactional Synchronization Extensions (TSX) to mount L3 cache attacks without a timer. However, TSX has been disabled since 2021, owing to the discovery of several side-channel security issues [12]. Next, Zhang et al. [86] observed that some of the newest Intel processors contain monitoring instructions such as `umwait` that are exploitable to build a timerless covert channel. While [86] reported a similarly-behaved instruction on an Arm Cortex-A73 CPU, it is unknown if Apple CPUs are susceptible. Moreover, the `umwait` instructions are not available to browser-based code.

Stepping away from mounting cache attack, Chen et al. [11] attempts to protect SGX applications from simultaneous multi-threading (SMT)-level side channels by ensuring that both threads are occupied by the applications' workload. As core scheduling is not exposed to enclaves, Chen et al. [11] achieves this by inducing race conditions on L1-cached variables arising from cache coherence, exploiting the fact that the L1 cache is shared among two sibling threads on Intel machines.

3 Threat Model

In this paper we focus mainly on Apple hardware. For the experiments presented in Sections 5 and 6, we assume that the target has been fully updated with Apple's MacOS 13.1 and iOS 16.2 (latest at the time of writing). In particular, we assume that side-channel countermeasures are left in their default enabled state and that the machine has no (known) software vulnerabilities.

Next, for the attacks presented in Section 6, we assume a typical model for web-based attacks, where the target visits an attacker-controlled website using the Safari web browser. Here we note that while macOS-based devices allow the installation of other browsers (e.g., Chrome), Apple prohibits non-Safari based browsers on iOS devices. In particular, all browsers installed on iOS devices must use WebKit as their underlying rendering engine, making nearly all modern iOS devices vulnerable to our attack.

4 Microarchitectural Primitives

We begin by exploring the status of basic primitives required for mounting micro-architectural attacks on Apple devices:

[$\mathcal{P}_1$] Identifying Cache Organization. We first need to determine the cache organization on Apple CPUs, as our microarchitectural primitives rely on some of these properties. Thus, we present a collection of primitives to help uncover the cache organization of Apple's Arm-based CPUs without any elevated privileges.

[$\mathcal{P}_2$] Measuring Speculation Depth. Another building block for our microarchitectural primitives is speculation. Knowing the cache layout allows us to evict data from each cache level, and subsequently measure the number of instructions that can execute speculatively before the data becomes reinstated.

[$\mathcal{P}_3$] Distinguishing Cache Hits From Misses Without Timers. Tackling the degradation of timer resolution as a widespread side-channel countermeasure, we present a primitive that uses speculation to distinguish cache misses from hits at each level of cache hierarchy using only low-resolution timers. We then show an improved variant of this primitive that does not use timers at all, instead using race conditions and shared memory to distinguish between cache hit vs. miss latencies.

[$\mathcal{P}_4$] Constructing Minimal Eviction Sets. We use our cache distinguishing primitive to develop a method for reliably finding minimal eviction sets amid the degradation or absence of timers. We improve on the group-testing eviction set finding algorithm [77] in ways that add resiliency, allowing us to integrate [$\mathcal{P}_3$].

[$\mathcal{P}_5$] Testing for L2 Inclusiveness. We combine [$\mathcal{P}_3$] and [$\mathcal{P}_4$] to determine whether the L2 cache is inclusive of the L1 caches.

[$\mathcal{P}_6$] Timerless Spectre Attacks. Finally, we demonstrate a Spectre-v1 gadget that can reliably leak data amid severely degraded or completely lacking timers.

4.1 [$\mathcal{P}_1$]: Cache Organization

We now proceed to reverse engineer the cache topology of Arm-based Apple CPUs. While kernel extensions can read model-specific

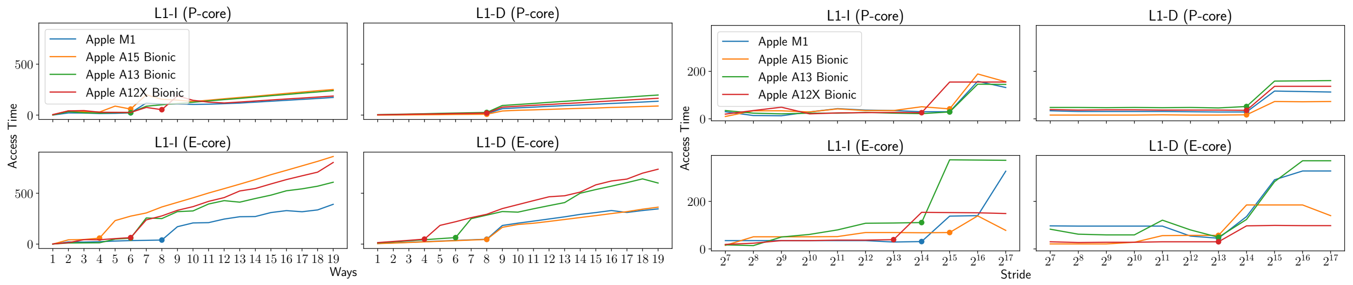


Figure 1: The access times when accessing the same offset in a loop for a varying number of pages to determine the ways (on the left) and the stride (on the right), and thus the size, of the L1-D cache set and the L1-I cache set on the P-cores and E-cores of various Apple CPUs. The round markers indicate the detected number of ways/stride.

registers to recover this information, doing so on sandboxed iOS devices is rather challenging. Thus, we now describe our empirical method to determine the cache topology.

Determining the Associativity of the L1 Cache. Since L1 caches are typically virtually indexed, identical page offsets of multiple pages are *congruent*: that is, they map to the same cache set. To determine the associativity a, we measure the runtime to loop through an increasing number of congruent addresses n, where a slowdown will occur if $n > a$. For the L1 instruction cache, we achieve this using chains of branch instructions which are all at congruent addresses. Figure 1 (left) summarizes our findings, showing the associativity of the L1 caches on various Apple CPUs.

L1 Cache Size. To parallelize address translation with cache lookups, L1 caches typically limit the indexing bits to the page offset bits. With Apple CPUs using 16KiB pages, we can upper-bound the size of the L1 cache, s, as $s \leq 16\,\text{KiB} \cdot a$. Next, cache designs often try to avoid self-eviction of blocks of continuous memory. Thus, we first allocate a block of size $16\,\text{KiB} \cdot a$, and repeatedly access $a + 1$ elements from it using a given stride δ. We keep doubling δ until we observe a slowdown, indicating the presence of self-evictions. Figure 1 (right) presents the access time across different strides. As the largest stride with fast access times, δ^*, still allows $a + 1$ elements to map to different sets, each cache way has δ^* capacity. Thus, the size of the entire cache is $s = a \cdot \delta^*$.

L1 Cache Line Size. To determine the cache line size, we loop through n' congruent addresses, where $n' > a$ and causes a slowdown. Then, we shift $\frac{n'}{2}$ addresses by an increasing offset. The smallest value of this offset where we observe a relative speedup is the cache line size l, since it will result in $\frac{n'}{2}$ addresses mapping to the next line, and thus the next cache set. Finally, once l is known, we can compute the number of sets t, since $t = \frac{s}{la}$.

L2 Cache Organization. The `sysctl` interface on macOS and iOS provides the line size l and total size s of the L2 cache. However, as the L2 cache is physically indexed and we therefore cannot retrieve a or t here, we rely on the ability to find minimal eviction sets, as its size must equal a. We detail this in Section 4.4.

Experimental Results. Running the above methodology across multiple iOS and MacOS devices, we were able to recover and document the organization of the L1 caches. Table 1 presents a summary of our findings across multiple generations of Apple CPUs, both for P-cores and E-cores.

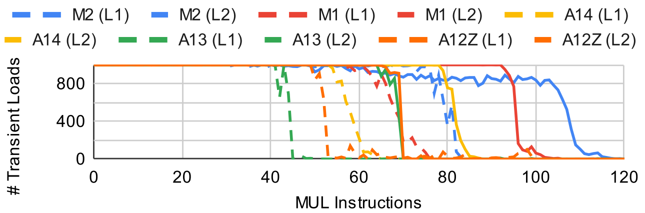


Figure 2: The number of `mul` instructions that can be executed speculatively when the condition is evicted from the L1 cache (dashed) vs. the L2 cache (solid) on various CPUs.

4.2 [$\mathcal{P}_2$]: Measuring Speculation Depth

Knowing the cache organization on modern Apple platforms, we now measure the number of instructions that can execute speculatively, also known as the speculation window length. We hypothesize that speculated branches will resolve much quicker when the data is in the L1 cache than the L2 cache or main memory.

Measuring L1 and L2 Speculation Depth. We use the cycle-accurate timer described in Section 2.5 to evict a series of values determining the condition of a branch instruction, thereby opening a speculation window. At the branch target, we introduce a sequence of data-dependent `mul` instructions followed by a load instruction for an uncached probe element. Here, we hypothesize that if we introduce enough `mul` instructions, the speculation window will be too short to reach the load instruction when the data is in either the L1 or L2 cache. Accordingly, we conduct an experiment where we evict the branch condition from the L1 and L2 caches, execute the branch, and measure the load latency to the probe element to determine how many `mul` instructions we need to introduce. We perform 1,000 trials for each number of `mul` instructions.

Results. Figure 2 shows the number of `mul` instructions that we can introduce before the probe element remains uncached. While each platform is different, we see that Apple's M series CPUs can execute about 100 `mul` instructions under speculation. With each `mul` requiring 3 cycles [32], this translates to speculative windows of about 300 instructions on high-end Apple platforms.

4.3 [$\mathcal{P}_3$]: Discerning Cache Hits From Misses

In this section, we consider the problem of discerning cache hits from misses in timer-restricted environments. We recall that in non-privileged environments, we cannot rely on high-resolution timers to achieve this for individual addresses. A solution to this problem is also suitable for other environments with restricted

		L1-I Cache				L1-D Cache				L2 Cache			
Name	CPU	W	S	Size	CL	W	S	Size	CL	W	S	Size	CL
MacBook Air (M2, 2022)	M2 (P)	6	512	192 KiB	64B	8	256	128 KiB	64B	16	8192	16 MiB	128B
(Mac14,2) A2681	M2 (E)	4	512	128 KiB	64B	8	128	64 KiB	64B	16*	2048*	4 MiB	128B
MacBook Pro 14"	M1 Max (P)	6	512	192 KiB	64B	8	256	128 KiB	64B	12	8192	12 MiB	128B
(MacBookPro18,4) A2442	M1 Max (E)	8	256	128 KiB	64B	8	128	64 KiB	64B	16*	2048*	4 MiB	128B
MacBook Pro 14"	M1 Pro (P)	6	512	192 KiB	64B	8	256	128 KiB	64B	12	8192	12 MiB	128B
(MacBookPro18,3) A2442	M1 Pro (E)	8	256	128 KiB	64B	8	128	64 KiB	64B	16*	2048*	4 MiB	128B
Mac Mini (M1, 2020)	M1 (P)	6	512	192 KiB	64B	8	256	128 KiB	64B	12	8192	12 MiB	128B
(Macmini9,1) A2348	M1 (E)	8	256	128 KiB	64B	8	128	64 KiB	64B	16	2048	4 MiB	128B
iPhone 13 mini	A15 Bionic (P)	6	512	192 KiB	64B	8	256	128 KiB	64B	12*	8192*	12 MiB	128B
(iPhone14,4) A2481	A15 Bionic (E)	4	512	128 KiB	64B	8	128	64 KiB	64B	16*	2048*	4 MiB	128B
iPhone 12	A14 Bionic (P)	6	512	192 KiB	64B	8	256	128 KiB	64B	16	4096	8 MiB	128B
(iPhone13,2) A2172	A14 Bionic (E)	8	256	128 KiB	64B	8	128	64 KiB	64B	16*	2048*	4 MiB	128B
iPhone 11	A13 Bionic (P)	6	512	192 KiB	64B	8	256	128 KiB	64B	16	4096	8 MiB	128B
(iPhone12,1) A2111	A13 Bionic (E)	6	256	96 KiB	64B	6	128	48 KiB	64B	16*	2048*	4 MiB	128B
iPad Pro 11"	A12X (P)	8	256	128 KiB	64B	8	256	128 KiB	64B	16	4096	8 MiB	128B
(iPad8,3) A2013	A12X (E)	6	128	48 KiB	64B	4	128	32 KiB	64B	–	–	2 MiB	128B
iPad 6th Gen (iPad7,5) A1893	A10X Fusion	4	256	64 KiB	64B	4	256	64 KiB	64B	†	†	3 MiB	128B
iPhone 7 Plus (iPhone9,4) A1784	A10 Fusion	4	256	64 KiB	64B	4	256	64 KiB	64B	†	†	3 MiB	128B

Table 1: The cache organization of various Apple CPUs. W: Ways, S: Sets, CL: Cache Line size. –: the pLRU-based minimal eviction set algorithm did not resolve, and we could not confirm using an alternative information source. †: these devices use a random replacement policy for the L1 cache [25], precluding us from using the eviction set algorithm. *: we could not confirm the results, but these are the most likely based on other Apple CPUs with similar cache organization.

timer resolution, such as all web browsers (see Section 5), and is thus of independent interest. We first describe a primitive to discern cache hits from misses at each cache level, using speculation to make measurement possible with only low-resolution timers. We further extend this gadget to a timerless variant, which is made possible by race conditions instead of timing data.

Constructing a Distinguisher Using Speculation. We now combine the speculation depth information for each cache level from Section 4.2 with the pLRU gadget as described in Section 2.6 to construct a distinguisher gadget that can tell whether the data for a given address is present or not in a specific level (L1 or L2) of the cache hierarchy. The distinguisher gadget resembles our setup for measuring speculation depth. More specifically, it speculates on a conditional branch, where this time the condition is the target data to be measured. The branch target begins with the number of `mul` instructions corresponding to the cache level to measure, where this number is derived from our results in Figure 2. Afterwards, instead of loading an uncached probe element, the gadget transmits a 1 using the pLRU gadget. That is, a 1 will be transmitted only if the target data is not present in the cache level being measured.

```
1  plru.init()
2  if (*target != 0) {
3      x *= 1;
4      ... // target cached - speculation ends here
5      x *= 1;
6      plru.transmit(1);
7      // target not cached - speculation ends here
8  }
```

Listing 2: Our speculation-based gadget to distinguish cache hits from misses in the absence of high-resolution timers.

Gadget Overview. Listing 2 is the pseudocode of our distinguisher gadget. After initializing the pLRU channel (Line 1), we branch on the value of the address to which `target` points (Line 2). If the CPU reaches Line 6 under speculation, the pLRU gadget transmits a 1, indicating a cache miss. Otherwise, the address to which `target` points is a cache hit, and no transmission occurs because speculation ends before Line 6.

Next, if a low-resolution timer is available, we can use it to measure the execution time of `plru.traverse()` in an attempt to ascertain whether a transmission had occurred inside our distinguisher gadget:

```
is_cache_miss =
badtimer.time(plru.traverse()) > threshold;
```

While this is similar to `plru.receive` from Listing 1 in that we time `plru.traverse` with a low-resolution timer against a coarse-grained threshold, we note that the resulting value now directly corresponds to the target's cache state. That is, `plru.traverse`'s runtime only exceeds `threshold` in case Line 6 was speculatively executed due to a prolonged speculation window induced by a cache miss in Line 2.

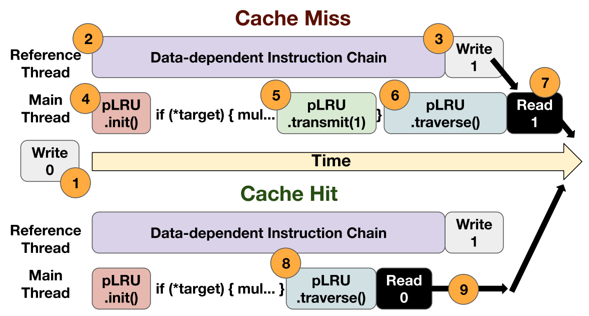


Figure 3: An overview of how our race condition-based gadget runs to distinguish cache hits from cache misses.

Measuring Without Timers via Race Conditions. We recall from Section 2.6 that `plru.traverse` allows one to arbitrarily amplify L1 miss latencies, making them visible using low-resolution timers. We now use this amplification property to remove the need for a timer altogether, allowing our gadgets to operate in a timerless environment. At a high level, we race two threads that update a shared variable, such that we can determine the order in which the threads finished running. Figure 3 outlines our gadget, where the syntax #⃝ is used to describe each part of the figure in detail.

Our race consists of a zero-initialized shared variable ①, a reference thread, and a main thread. The reference thread ② always executes a chain of data-dependent instructions, and then writes 1 ③ to the shared variable. The main thread ④ starts concurrently, runs the distinguisher gadget in Listing 2, calls `plru.traverse`, and then reads the shared variable, outputting the result.

We now argue that after calibrating `plru.traverse`, the output of the main thread in fact corresponds to the cache state of the target variable. More specifically, if the target data is not cached, the CPU has a long speculation window for our distinguisher gadget in Listing 2, reaching `plru.transmit(1)` ⑤ under speculation. This causes the subsequent `plru.traverse` routine ⑥ to take longer, finishing after the reference thread has written 1. Therefore, the main thread reads 1 ⑦ in this case.

In contrast, if the target data is cached, the CPU only has a short speculation window in Listing 2, never reaching the `plru.transmit(1)` operation. Next, as nothing is transmitted over the pLRU channel, the `plru.traverse` routine terminates quickly ⑧. With the race results being flipped, this causes the main thread to read 0 before the reference thread can write 1. ⑨

Experimental Results. We now evaluate how accurately the distinguisher gadget classifies four outcomes: hits and misses, on the L1 and L2 caches. Table 2 summarizes our findings, as we degrade the timer resolution on an Apple M1 machine from an unprivileged 42 ns timer to removing the timer altogether. As can be seen, we observe perfect discernment even with a timer resolution of 1 ms. The timerless version likewise achieves high accuracy, particularly for distinguishing L2 hits from misses.

Timer Resolution	42 ns	10 μs	100 μs	1 ms	Timerless
L1 Hit	100%	100%	100%	100%	82%
L1 Miss	100%	100%	100%	100%	77%
L2 Hit	100%	100%	100%	100%	99%
L2 Miss	100%	100%	100%	100%	97%

Table 2: Probability of a correct observation (out of 1000 runs) by our distinguisher gadget, across various timer resolutions.

4.4 [$\mathcal{P}_4$]: Constructing Minimal Eviction Sets

Now, we use our distinguisher gadget to find minimal L2 eviction sets from an unprivileged user, even in environments that only provide low-resolution timers, or no timer at all. As browser-based code does not have access to any cache flush instructions, finding minimal eviction sets in timer-restricted environments is a common side-channel task, making this technique of independent interest.

Baseline Approach of [77]. Eviction set finding begins with an inflate step, where we keep allocating and accessing elements until the victim address is evicted. Its output is called a *conflict set*, and the goal is to reduce it to a minimal eviction set (i.e., whose size is the cache's associativity a). Vila et al. [77] use group testing to expedite reduction, assuming a is known. In each iteration, their algorithm divides the conflict set into $a+1$ bins, withholds one bin at a time, and checks if the remaining bins still evict the victim. If so, the withheld bin is discarded, and the remaining bins become the conflict set for the next iteration. Due to the Pigeonhole principle, there are ≥ 1 bins that can be discarded each iteration from a conflict set of size n. Hence, the algorithm discards $\frac{n}{a+1}$ elements per iteration until $n = a$.

Improving the Reduction. We improve this technique by generalizing it to operate on an upper bound k of a, rather than the exact number of ways. Then, during each iteration, we divide the conflict set into k bins instead of $a+1$ bins. We deviate from [77] by testing the remaining bins after discarding a bin, instead of starting the next iteration. This lets the next iteration start with the minimal set of bins required for eviction. We note that in the case k is large (i.e., 64), most bins are redundant. Thus, our reduction removes $k-a$ bins on average per iteration, instead of just one.

Improving the Backtracking. Noise sometimes causes the reduction step to remove bins that are essential for eviction. [77] remediates this with a backtracking step that adds the last removed bin back to the conflict set. However, we found this approach is still susceptible to noise, especially when used with the speculation-based distinguisher gadget from Section 4.3, and often the algorithm fails to converge. We improve this by reallocating as many of the previous elements until the conflict set evicts the victim again.

Empirical Results. We first implement both [77] and our algorithm using the privileged cycle-accurate timer to time the victim's load latency as a baseline, and measure the time to convergence and success probability over 20 trials on an Apple M1. Here, we define success as the resulting eviction set being minimal and able to evict the victim. We then introduce the distinguisher gadget and use it alongside coarser timing sources, starting with the unprivileged timer in macOS with 42ns resolution and ending at no timer (for the latter, we use the variant with race conditions). We present the results in Figure 4. While our reduction algorithm's time to convergence is longer, it eliminates false positives (i.e., when the reduction converges but the resulting eviction set fails to evict the victim) for all timer resolutions except when the timer is removed.

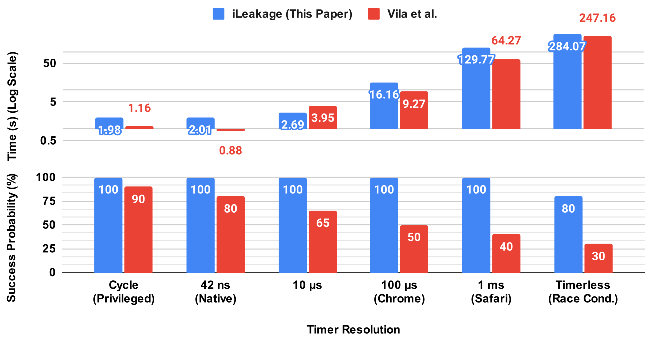


Figure 4: Comparison of eviction set-finding algorithms (ours and [77]) across degrading timer resolutions.

4.5 [$\mathcal{P}_5$]: Testing for L2 Inclusiveness

With the ability to efficiently and reliably find minimal L2 eviction sets, we now ascertain that the L2 cache is *inclusive*. Otherwise, the L2 cache is *non-inclusive* or *exclusive*. We test for inclusiveness using an L2 eviction set and the distinguisher gadget from Section 4.3. We check if traversing the L2 (shared per-cluster) eviction set leads to L1 (private) cache evictions in another core. That is, we expect to observe cross-core L1 evictions only if the cache is inclusive.

Indeed, we find that the L2 cache of the Apple M1 is inclusive of the L1 caches. Furthermore, given that the L2 cache line size (128 B) is double that of L1 (64 B), we observe improved eviction rates when accessing both halves of each L2 cache line.

Experimental Setup. Noticing the discrepancy in cache line sizes between the L1 and L2 caches, we design experiments to determine if this affects their inclusiveness. In our first three experiments, we access the whole L2 cache line, the first 64 bytes, and the last 64 bytes respectively during the eviction test, and measure the L1 eviction rate of the victim (which is cached in another core's L1). As a control, our fourth experiment evicts an unrelated L2 cache set, which should leave the victim's L2 cache line intact. Subsequently, we measure the effect of repeated eviction set traversals. We use a spinlock to synchronize the attacker's thread with the victim thread, wherein the attacker's thread shuffles and traverses the eviction set 1, 2, or 3 times before yielding to the victim thread.

Results. We show the cross-core L1 eviction rates in Figure 5. Here, we observe from the first three columns that while accessing half of an L2 cache line occasionally results in evictions, accessing both halves approximately triples the success rate. We also verify that evicting a different L2 cache set does not result in victim evictions. For the experiments with synchronization, we observe that accessing the eviction set more often results in higher eviction rates, with 3 access iterations guaranteeing eviction. Overall, our results strongly indicate that the M1's L2 cache is inclusive.

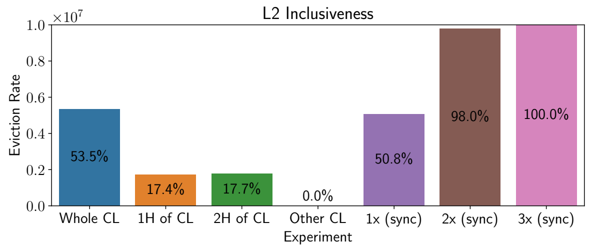


Figure 5: L1 cache eviction rate of a victim when evicting its L2 cache line from another core on the Apple M1 CPU.

4.6 $[\mathcal{P}_6]$: Timerless Spectre Attacks

We recall from Section 2.5 that all major web browsers have degraded their timer resolution in an attempt to harden browsers against side-channel attacks [52, 55, 78]. In fact, browser vendors has gone as far as to restrict the use of `SharedArrayBuffers`, in an effort to stop attacks from building their own timers [14]. While concurrency and shared state continues to be available through web workers and the message passing API [66] due to compatibility reasons, the latency of these interfaces has been artificially increased to be orders of magnitude greater than that needed to distinguish a single cache miss from a hit.

Building on our timerless attack primitives presented earlier, we are able to use the browser's message passing API to cause race conditions, allowing us to execute the cache-distinguisher method from Section 4.3. We then construct a timerless browser-based Spectre v1 gadget, using race conditions rather than cache timing to recover the secrets leaked during speculative execution.

Table 3 shows accuracy and leak rate for a proof-of-concept timerless Spectre-v1 attack using this approach, benchmarked on an Apple M1 laptop running unmodified Safari 16.2, Firefox 108.0, and Tor Browser 12.0.1 (all latest at the time of writing). While the attack's leak rate is relatively low due to noise introduced by the JavaScript environment, we note the attack achieves close-to-perfect accuracy without any explicit or constructed timers.

Metric	Tor Browser	Firefox	Safari
Accuracy	96.15 %	98.79 %	97.46 %
Leak Rate	10.63 b/s	10.76 b/s	8.26 b/s

Table 3: Timerless Spectre-v1 performance across browsers.

5 Attacking the Safari Browser

We now present iLeakage, a JavaScript-based transient-execution attack that recovers secret information from the Safari browser. In addition to bypassing standard side-channel countermeasures (such as a low-resolution timers) deployed by all browser vendors, iLeakage overcomes several Safari- and Apple-specific challenges.

$[\mathcal{C}_1]$ Site Isolation. Just like Chrome and Firefox, modern versions of Safari attempt to prevent mutually-distrusting webpages from using the same rendering process, compartmentalizing different websites into different address spaces.

$[\mathcal{C}_2]$ 35-bit Pointers and Value Poisoning. In addition to site isolation, Safari also limits the attacker's ability to craft and dereference arbitrary 64-bit pointers. All indexing in JavaScript objects is 32-bits and most object accesses entail 35-bit compressed pointers, while 64-bit doubles are poisoned. Thus, an attacker must bypass Apple's isolation countermeasures to retrieve sensitive information.

$[\mathcal{C}_3]$ Obtaining Deep Speculation. To overcome $[\mathcal{C}_2]$, we use speculative type confusion that requires the CPU to speculate past multiple condition checks. Thus, we need a consistent method to ensure that the CPU does not revert the speculation before the attack completes executing transiently and leaks the sensitive data.

$[\mathcal{C}_4]$ Reliability. Finally, we design a reliable exploit that can be used to leak data multiple times. In particular, we need to architecturally hide all the type confusion events to prevent the browser from raising exceptions or falling back to the JavaScript interpreter.

Attack Overview. We first resolve $[\mathcal{C}_1]$ by using the `window.open` JavaScript function, observing that it brings the target website's data into the attacker's address space in Safari. For $[\mathcal{C}_2]$, we forge 64-bit pointers around value poisoning countermeasures via Safari's performance optimization within its mitigations for architectural type confusion. We show this mitigation is insufficient for speculative type confusion, with the CPU transiently dereferencing our forged pointer. For $[\mathcal{C}_3]$, we inspect Safari's memory allocator and data structures for JavaScript objects to selectively evict the type of our attacker object from the L2 cache, but keep the rest of the object in the L1-D cache. Finally, for $[\mathcal{C}_4]$, we hide the type mismatch events from Safari with another layer of speculation.

5.1 $[\mathcal{C}_1]$: Bypassing Process Isolation

To mount a speculative execution attack, the attacker must coerce the target webpage into its address space. Recognizing this, both Chrome and Firefox recently implemented a Site Isolation paradigm [53, 65] to ensure that different rendering processes handle pages with different effective top-level domain plus one sub-domain (eTLD+1). See Appendix A for a more complete discussion.

Safari's Isolation Model. Taking this approach a step further, Safari follows a simple one process per tab model, where two webpages are never consolidated into the same rendering process, even under high memory pressure and even if they share an eTLD+1 in their URLs. Instead, Safari spawns a new rendering process for each tab until the system runs out of memory. Empirically confirming this, we used a MacBook Air with the M1 CPU and 16 GB of RAM and opened 177 tabs. While this had the effect of Safari refusing to open additional tabs, it never consolidated webpages into rendering processes, maintaining its one process per tab model. Similar results were obtained on an iPad Pro (12.9-inch, 5th generation).

Abusing `window.open` to Achieve Consolidation. We found that we can reliably render a target page inside the address space of an attacker's page by using the `window.open` JavaScript API. In particular, `attacker.com` can call `window.open` to open a pop-up window rendering `target.com`. Crucially, while both websites appear in different windows, Safari uses a single rendering process for both pages, causing both websites to end up in the same address space. Notably, websites cannot refuse `window.open` to render them in a separate window, making this technique applicable for any target website. Finally, while we also observe cross-origin iframes consolidation, WebKit's Intelligent Tracking Prevention countermeasure prevents the delivery of cookies to cross-origin iframes [82], precluding them from rendering secrets.

Consolidation, Cross-origin Isolation, and Timers. We recall from Section 2.5 that the `SharedArrayBuffer` API can be used to build high-resolution timers. However, to mitigate side channels, all major browsers now limit its availability to cross-origin isolated pages. However, we observe that cross-origin isolated webpages fail to consolidate in Safari with any other webpage using either technique. In turn, this implies we cannot use `SharedArrayBuffer` to craft a high-resolution timer for our attack. Accordingly, we use the pLRU covert channel described in Section 2.6.

5.2 [$\mathcal{C}_2$]: Speculative Type Confusion

As outlined in Appendix B, despite being a 64-bit application, Safari uses 32-bit array indices and 35-bit pointers to the underlying storage of most objects, partitioning them into 32 GB compartments named Gigacages. As secrets in a webpage's DOM are located outside this Gigacage, they remain out of reach for naïve Spectre attackers that can only corrupt 32-bit indices or 35-bit pointers.

In this section we overcome this countermeasure, building a speculative type confusion attack which can read arbitrary 64-bit addresses in the page's rendering process. While we acknowledge prior type confusion techniques against Chrome [1, 26, 51] and the Linux kernel [36], to the best of our knowledge this is the first application of speculative type confusion to Apple's ecosystem in general and the Safari web browser in particular.

Locating 64-bit Pointers. We begin our investigation of speculative type confusion attacks in Safari by inspecting the memory layout of common JavaScript objects, paying special attention to 64-bit pointers. A particularity convenient object containing a 64-bit pointer is JavaScript's `string` object, whose memory layout we show in Figure 6. As can be seen in Figure 6 (bottom), WebKit writes characters in a separate C-style array, and dereferences a 64-bit C-array pointer to access it (Figure 6 (middle), StringImpl).

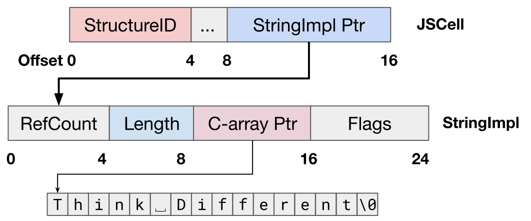


Figure 6: Memory layout of JavaScript strings in WebKit. The `JSCell` class is common to all objects, while the `StringImpl` class is the backing store specific to strings.

Performing a String Indexing Operation. More specifically, Listing 3 contains pseudocode of JavaScript's string indexing operation, when compiled by WebKit's JIT compiler. The code takes as input a variable `inp`, a WebKit metadata object called `JSCell`, common to all JavaScript objects. Having verified that the type inside `inp` indeed corresponds to a string (Line 3), Listing 3 proceeds to retrieve the corresponding `StringImpl` structure (Line 6), a WebKit internal structure holding the string's length and a 64-bit pointer to the string's actual content. After checking that the index being referenced is smaller than the string's actual length (Line 8), Listing 3 retrieves the corresponding character by dereferencing the 64-bit `cArrayPtr` (Line 11).

```
1   String::operator[] (inp, index) {
2       // Check StructureID first
3       if (inp.StructureID != String::StructureID)
4           exitToInterpreter();
5       // Dereference ptr to get StringImpl
6       StringImpl impl = *(inp.ptr);
7       // String length check
8       if (impl.length <= index)
9           exitToInterpreter();
10      // Dereference ptr to C-array
11      return *(impl.cArrayPtr + index);
12  }
```

Listing 3: JIT-compiled WebKit string indexing operation.

Attacker Object Setup. We assume the attacker would like to dereference some 64-bit address `addr`. Then, the attacker has to create an object whose memory layout resembles that of a JavaScript string. As objects must support heterogeneous property types, WebKit implements this by making every property a `JSValue` class. Conveniently, each `JSValue` is 64 bits wide. However, in order to craft a `JSValue` whose value holds `addr`, the attacker needs control over all 64 bits. This is not trivial, as some of `JSValue`'s 64 bits are patterned by WebKit to indicate which property type it holds. Aside from references to other JavaScript objects (which are always the attacker's own data), an attacker can only fill `JSValue` with integers, floats, `True`, `False`, `Undefined`, and `null` values. Figure 7 indicates the bit pattern corresponding to each value.

Value Poisoning via NaN-boxing. Using bit patterns to indicate different property types within a fixed-width datatype is a technique known as NaN-boxing. Remarkably, in WebKit's implementation of NaN-boxing, bits 63-48 are non-zero for integers. Furthermore, WebKit uses the IEEE 754 double-precision format [29] to represent floats. While this format requires control over all 64 bits, WebKit adds a poison value of 2^{49} to the encoded bits, making bit 49 or

Integer	0b 1111111111111110...xxxxxxxxxxxxxxxx
Float	0b xxxxxxxxxxxxxx1x...xxxxxxxxxxxxxxxx
True	0b 0000000000000000...0000000000000111
False	0b 0000000000000000...0000000000000110
Undefined	0b 0000000000000000...0000000000001010
Null	0b 0000000000000000...0000000000000010
	Bit 63 48........15 0

Figure 7: NaN-boxing in WebKit `JSValues` to discern data types. The bit patterns that indicate each data type are colored in red, while the attacker-controllable bits are marked by `x`.

higher always set in memory. In both cases, the 64 bits comprising the `JSValue` can never represent canonical virtual addresses. For all other property types, most bits are zeroed out such that they cannot be used to read arbitrary locations in the address space. Hence, these bit patterns not only serve as delineating sequences for property types, but also as a countermeasure against forging pointer values for architectural type confusion.

Bypassing Value Poisoning. Inspecting WebKit's NaN-boxing implementation, we found a performance optimization for floats. While adding 2^{49} prevents architectural type confusion, it requires performing arithmetic every time when computing on a float for poisoning and unpoisoning. Therefore, if an object contains an array of floats, WebKit marks this information in the object's type instead of poisoning each float. Hence, the attacker must create an object with an array of floats to avoid poisoning, where they must compute the inverse of the IEEE 754 encoding to convert the bits of `addr` to a valid float. Likewise, the attacker must provide a valid number for a string's length variable, which overlaps with the lower 32 bits of the indexable property at index 0. See Figure 8 (top) for an initialization example, where the `ieee754-inv` function performs this conversion (i.e., from bits to floats).

Architecturally, WebKit's type checking mechanism prevents unpoisoned floats from being interpreted as pointers when this optimization is applied. However, when the CPU speculates past type checks, we show that this security guarantee fails to hold.

Attacker Object Memory Layout. We show the memory layout of the `attackerObj` class after running the code in Figure 8 (top) in Figure 8 (bottom), along with the memory layout of a string. At the `JSCells` for the two objects, we note the pointers to the underlying storages (`StringImpl Ptr` and `Butterfly Ptr`) are at an offset of eight bytes in both cases. In the underlying storage containing an array of floats, Line 3 of Figure 8 (top) puts the value `0xffff` at offset 4 (after IEEE 754 conversion) where the string length resides, and line 5 puts the target address `addr` at offset 8, which contains the C-array pointer in the string.

Dereferencing Arbitrary 64-bit Pointers via Speculative Type Confusion. To dereference the 64-bit pointer, we consider the case where the code in Listing 3 is executed on the `attackerObj` created in Figure 8(top) with `index=0`. Before this code executes, we evict the `StructureID` of our object from the cache to delay the resolution of the branch at Line 3. This essentially forces the CPU to speculate forward, derefencing the object's butterfly pointer and treating the resulting butterfly as a `StringImpl` structure (Line 6). Next, the CPU incorrectly uses the data in `attackerObj[0]` as the string length (Line 9). As the string length is `0xffff`, which is larger than `index`, the CPU bypasses the branch and incorrectly uses the data in `attackerObj[1]` as the string's `cArrayPtr`, derefences it and returns the resulting value under speculation.

```
1  let attackerObj = {};
2  // [0] partially overlaps with length
3  attackerObj[0] = ieee754-inv(0xffff);
4  // [1] overlaps with C-array ptr
5  attackerObj[1] = ieee754-inv(addr);
```

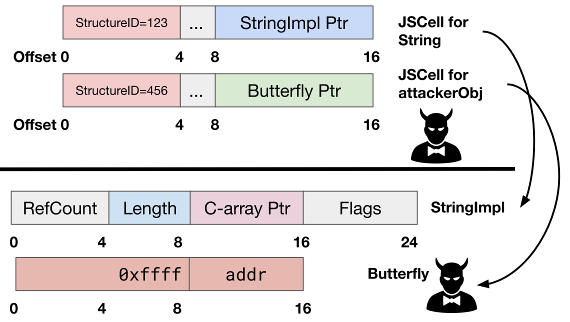


Figure 8: (Top) Pseudocode to construct a malicious object to obtain a transient dereference of the target address `addr`. (Bottom) A comparison with the memory layout of a `string` object.

Type Eviction. To ensure the type confusion attack succeeds, we delay the resolution of the branch in Line 3 as much as possible by evicting the cache line holding the `StructureID`. However, this poses two challenges. First, the attacker must be able to construct eviction sets for a given address, overcoming Safari's 1 ms timer. Next, the attacker must keep the rest of the data of the `attackerObj` cached to prevent the CPU's pipeline from stalling. Overall, to meet both requirements, we must force WebKit's internal memory allocator to place the `attackerObj` across cache line boundaries.

5.3 [$\mathcal{C}_3$]: Partial Object Eviction

As outlined above, our type confusion primitives requires placing an `attackerObj` across cache line boundaries, such that we can evict its `StructureID` variable while retaining the rest of the object in the CPU's cache. Next, to evict the `StructureID`, we must find a methodology for evicting specific addresses from the CPU's cache. This implies bypassing Safari's 1 ms timer, in order to distinguish cache misses from cache hits.

Understanding Safari's Memory Allocator. We depict the ideal scenario from above in Figure 9. Unfortunately, we observe that WebKit's memory allocator always places JavaScript objects, such as `attackerObj`, at 16-byte boundaries. Next, we note that most Intel and AMD CPUs use 64-byte cache lines while recent Apple CPUs use 128-byte cache lines in their L2 cache. Thus, we cannot evict the `attackerObj`'s `StructureID` without also evicting its butterfly pointer, preventing our attack.

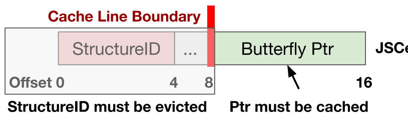


Figure 9: Ideal placement of the `JSCell`, with a new cache line between `StructureID` and `Butterfly Ptr`.

Constructing Split Objects. Inspecting WebKit's memory allocator, we find a performance optimization where for certain built-in objects in the JavaScript API, the first 8 instances for each object type are 'half-aligned', using 8-byte alignment boundaries.

While user-defined `attackerObj` do not qualify for this optimization, `Intl.Locale` objects do qualify for such placement. See Figure 10. Next, although `Intl.Locale` stores region-specific information (e.g., fonts, numbering system, calendar, etc), JavaScript allows us to set its indexed properties to floats, facilitating type confusion. Finally, when WebKit places such objects on 8-bytes, the `StructureID` and butterfly pointer land on adjacent caches line for some of these objects, as the `JSCell` structures are 168 bytes.

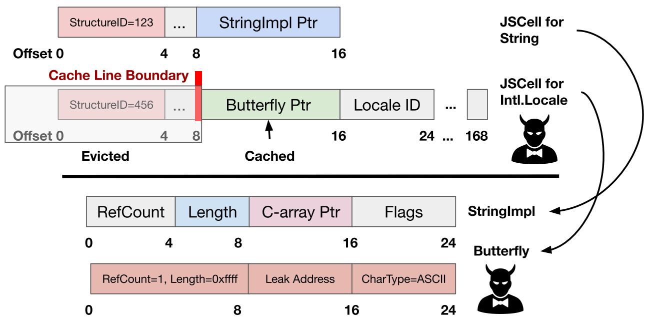


Figure 10: Comparison of memory offsets between a string object and an `Intl.Locale` object whose `JSCell` class is split across cache lines, and whose indexed properties mimic the backing store of a string object. We use the latter for speculative type confusion.

Achieving Cache Evictions. A final prerequisite to mount our type confusion attack is the ability to evict the address holding the `StructureID` of an `Intl.Locale` object. While Safari's 1 ms timer prevents the attacker from distinguishing cache hits from misses, thus constructing eviction sets, we recall the technique from Section 4.4 for constructing eviction sets using only low-resolution timers. Porting a similar approach to Safari, we can evict `StructureID`s of these objects, thereby facilitating our attack.

5.4 [$\mathcal{C}_4$]: Speculative Suppression

Having demonstrated how to achieve type confusion between JavaScript's `Intl.Locale` and `String` objects while making the `Intl.Locale`'s `JSCell` object straddle two cache lines, we now describe our memory read primitive end-to-end in Listing 4.

```
1  let malObj = new Intl.Locale("en-US");
2
3  for (let i = 0; i < 10000; i++)
4      gadget(0, 0, "training");
5
6  const junk = malObj[1];
7  malObj = [ieee754-inv(0xffff), ieee754-inv(addr)];
8  evict_type(malObj);
9  plru.init();
10 gadget(0xffff, index, malObj);
11 return plru.receive();
12
13 function gadget(condVar, index, confusionObj) {
14     if (condVar < confusionObj.length)
15         let val = confusionObj[index];
16         plru.transmit(val);
17 }
```

Listing 4: Our speculative type confusion primitive.

Setup. As our attack relies on speculatively performing type confusion between `Intl.Locale` and `String` objects, Line 1 allocates the `malObj` of type `Intl.Locale`, whose `JSCell` is split between cache lines as outlined in Figure 10.

Training. We call `gadget()` 10,000 times (Lines 3 – 5) with `condVar` and `index` set to 0, and `confusionObj` set to "training" to train the branch predictor. As `condVar` is less than 8, the CPU executes the branch, which sets `val` to '`t`' and transmits '`t`' over the pLRU channel (Line 14 – 16). By passing a string to the `gadget` function repeatedly, Safari consequently specializes the access at Line 15 to use the JIT-compiled code from Listing 3 instead of processing Line 15 with its JavaScript interpreter.

Attack Phase. We then ensure that the CPU cache contains our malicious `Intl.Locale` object (Line 6). We set the two indexed properties of `malObj` to be a fake string length and the address of the contents we wish to leak (Line 7).[1] As `malObj` is split between two cache lines, we can evict its `StructureID` while keeping its butterfly pointer cached (See Section 5.3), which we do on Line 8. We then proceed to call `gadget()` on `malObj`.

Speculative Type Confusion. With the `StructureID` evicted, the CPU fails to retrieve the `length` property (Line 14), leading it to speculate the `if` as a consequence of mistraining. As the code at Line 15 is specialized to use Listing 3, calling `gadget()` on `malObj` results in a type confusion where the CPU dereferences and returns the value of `addr`. We then transmit (Line 16) this value over the pLRU channel.

Speculative Suppression. As Line 15 was executed speculatively, JavaScript cannot retrieve the `length` property of `malObj` while `StructureID` is evicted. Once the value of `malObj.length` is architecturally available however, the CPU rolls back the incorrect speculation at Lines 15 – 16. Thus, speculative suppression prevents Safari from de-specializing the memory access at Line 15 or producing a type error event, allowing us to keep on repeating the attack across multiple addresses.

Value Recovery. Finally, we have to retrieve the value stored at `addr`, which was transmitted through the pLRU channel (Line 16 of Listing 4). Here, we simply rely on the pLRU channel for this (Line 11), calibrating it for Safari's 1 ms timer, see Section 2.6.

5.5 End-to-End Attack Evaluation

Having described the collection of techniques to build our attack, we now proceed to evaluate its leakage rate and accuracy across a variety of Apple devices. More specifically, we run iLeakage on unmodified Safari and try to read a known 512-bit string, averaging accuracy over 10 trials. Table 4 contains a summary of our findings, showing that our attack can read arbitrary address at a rate of 24 to 34 bits per second, with an accuracy of 90% to 99%.

6 Weaponizing iLeakage

We now turn our attention to the implications of iLeakage on the security of the Safari web browser.

Experimental Setup. We use a MacBook Air (model A2337) with the M1 CPU and 16 GB of RAM for all attack experiments. We run Safari in an out-of-the-box configuration, with all side-channel countermeasures enabled.

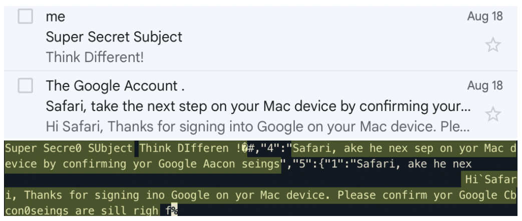
[1] Recall that these values must be encoded as IEEE-754 floats, as explained in Figure 8.

Device	Apple CPU	Leak Rate	Accuracy
iPad Pro 11" 2nd Gen.	A12Z Bionic	23.22 b/s	97.36 %
iPhone 11	A13 Bionic	28.89 b/s	99.18 %
iPhone 12	A14 Bionic	33.13 b/s	98.96 %
iPad Pro 12.9" 5th Gen.	M1	34.78 b/s	99.57 %
iPhone 13	A15 Bionic	26.29 b/s	95.92 %
MacBook Air (M1, 2020)	M1	32.97 b/s	98.48 %
MacBook Pro (14", 2021)	M1 Pro	33.28 b/s	90.96 %
MacBook Pro (14", 2021)	M1 Max	24.46 b/s	93.79 %
MacBook Air (M2, 2022)	M2	29.14 b/s	97.05 %

Table 4: End-to-end performance of iLeakage.

Bringing Targets to Attackers. We begin by recalling that while Safari generally follows a strict process-per-tab model, pages opened by the `window.open` function share a rendering process with the parent page. Thus, we created an attacker page that binds `window.open` to an `onmouseover` event listener, allowing us to open any webpage in our address space whenever the target has their mouse cursor on the page. We note that even if the target closes the opened page, the contents in memory are not scrubbed immediately, allowing our attack to continue disclosing secrets. Finally, as `window.open` performs consolidation regardless of the origins of both the parent and opened webpages, we host our attacker's page on a non-publicly accessible webserver, while using `window.open` to consolidate pages from other domains.

Attacking Gmail. With Google being one of the world's largest email providers, it is highly likely for a target to be signed in with their personal account. By having the event listener inside the attacker's page access execute `window.open(gmail.com)`, we can consolidate the target's inbox view into the attacker's address space. We then leak the contents of the target's inbox, see Figure 11.

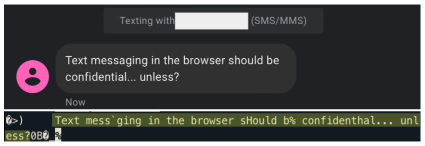

me Aug 18
Super Secret Subject
Think Different!

The Google Account . Aug 18
Safari, take the next step on your Mac device by confirming your...
Hi Safari, Thanks for signing into Google on your Mac device. Ple...

```
Super Secre0 SUbject Think DIfferen !�#,"4":"Safari, ake he nex sep on yor Mac d
evice by confirming yor Google Aacon seings","5":{"1":"Safari, ake he nex
                                                                        Hi`Safar
i, Thanks for signing ino Google on yor Mac device. Please confirm yor Google Cb
con0seings are sill righ f%
```

Figure 11: (Top) An email displayed in Gmail's web view. (Bottom) Recovered sender address, subject, and content.

Recovering Android Text Messages. Android users can send and receive text messages from a browser window by pairing their phone with Google's Messages platform. Thus, by opening Google Messages using `window.open()`, we can recover a target's text messages without attacking their mobile phone itself. See Figure 12.


Texting with [redacted] (SMS/MMS)

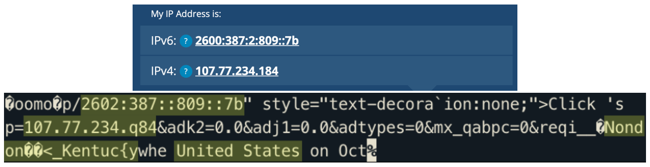
Text messaging in the browser should be confidential... unless?

Now

```
�>)     Text mess`ging in the browser sHould b% confidenthal... unl
ess?0B�_%
```

Figure 12: (Top) Text message sent to an Android phone which has been paired to the Google Messages webpage. (Bottom) Recovered text message in highlights.

IP Address and Geolocation. Finally, an attacker might decide to open a website the target does not normally visit, in order to learn more information about the targeted user. For example, in case the attacker does not have access to server logs for their malicious page (e.g., due to hosting on third party servers), the attacker can open an IP address geolocation page and subsequently recover the target's location, IP and ISP details. See Figure 13.


My IP Address is:

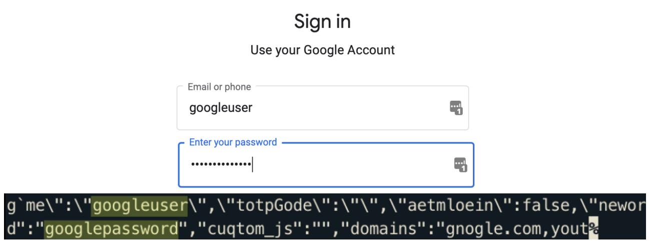
IPv6: 2600:387:2:809::7b

IPv4: 107.77.234.184

```
�oomo�p/2602:387::809::7b" style="text-decora`ion:none;">Click 's
p=107.77.234.q84&adk2=0.0&adj1=0.0&adtypes=0&mx_qabpc=0&reqi__�Nond
on��<_Kentuc{ywhe United States on Oct%
```

Figure 13: (Top) Website displaying the target's IPv4 and IPv6 addresses. (Bottom) Recovered information from the website's DOM in highlights, including geolocation.

Attacking Password Managers. Going beyond reading website content, the popularity of password managers also allows us to go after login credentials of popular websites. More specifically, we installed LastPass version 4.107.1 (the latest at the time of writing) on our Safari browser. Next, while LastPass requires user interaction when autofilling credentials for the first time on a webpage, it automatically and permanently fills them in on subsequent logins without any interaction. Thus, by opening the login pages of popular websites (prompting LastPass to autofill credentials), we can recover the target's username and password. See Figure 14.


Sign in

Use your Google Account

Email or phone
googleuser

Enter your password
•••••••••••••••

```
g`me\":\"googleuser\",\"totpGode\":\"\",\"aetmloein\":false,\"newor
d":"googlepassword","cuqtom_js":"","domains":"gnogle.com,yout%
```

Figure 14: (Top) Google's accounts page autofilled by LastPass, where the password is `googlepassword`. (Bottom) Leaked page data with credentials highlighted.

Forced Logout to Induce Login Page. To steal credentials, an attacker must first prompt a credential manager to autofill the target's password. Thus, when the target is already authenticated for a service, we must somehow log the target out to access the service's login screen, thereby triggering password autofills.

Here, we observe that logout operations on most services do not require knowing any user-specific information. Thus, an attacker can recover credentials from Amazon, GitHub, and Google by by first sending an AJAX request in JavaScript to the logout endpoints of each service, then opening the login pages via our `window.open` technique.

7 Discussion

Observing the practicality of our end-to-end attack, we now discuss countermeasures to iLeakage, including Apple's efforts during the responsible disclosure process. We also state iLeakage's limitations and high-level takeaways from our attack primitives, some of which are applicable to other browser engines and vendors.

7.1 Countermeasures for Safari

Throughout Section 5, our attack leveraged facets in Safari's multi-process architecture and performance optimizations. We now discuss several countermeasure designs for closing these loopholes.

Preventing Consolidation via Site Isolation. Recall WebKit's process model (Section 5.1), where pages launched with `window.open` are consolidated into the address space of their parent's rendering process. While this is practical since the rendered pages must have a reference back to their parent page in such cases, this design was what made our attack possible.

Following our disclosure, Apple has revised Safari's site isolation strategy, developing a new inter-process communication API that makes spawning new processes for pages launched with `window.open` possible, in addition to pages opened in new tabs. We have empirically verified this mitigates our attack by preventing consolidation of domains across security boundaries. That is, while the speculative JavaScript sandbox escape is still possible, an attacker becomes limited to reading their own address space and therefore their own data. Finally, while Apple's patch is publicly available on the WebKit codebase [57], Apple does not currently have a timeline for the deployment of this mitigation.

Preventing Speculation Past Type Checks. In Section 5.2 we constructed a 64-bit read primitive by confusing the CPU to assume the input is a string object, while in reality the input is an `Intl.Locale` object. WebKit's JIT compiler can be hardened against type confusion attacks by inserting a fence instruction after every type check to prevent the CPU from speculating past it.

Removing Poisoning Optimizations. Also in Section 5.2, we were able to fake a 64-bit pointer by abusing an optimization where WebKit would not poison floats in an array of floats, instead encoding that information in the array's type information. As our attack demonstrates, this is risky in the face of speculative type confusion. Thus, while poisoning all floating-point numbers would not prevent speculative type confusion itself, it would prevent an attacker from reading the address space by using floats to craft 64-bit pointers. Yet, this comes with performance tradeoffs: every computation on a poisoned variable must be preceded by unpoisoning and succeeded by repoisoning when writing the result back to memory.

7.2 Countermeasures for Websites

As full site isolation in Safari is under active development by Apple, we now discuss a countermeasure available to website administrators for mitigating our attack. Here, web pages can serve cross-origin isolation (COI) HTTP headers with website endpoints containing secrets. We recall from Section 2.5 and Section 5.1 that COI was proposed as a Spectre countermeasure for websites, and pages with COI failed to consolidate in Safari.

COI consists of two HTTP headers: Cross-Origin-Opener-Policy (COOP) set to same-origin, and Cross-Origin-Embedder-Policy (COEP) set to require-corp. COOP states the current webpage must not be able to communicate with other pages, while COEP requires third-party resource to explicitly opt into being loaded on the current webpage [39]. In current release versions of Safari, we identify that the presence of the correct COOP header is alone for WebKit to spawn the webpage in a new process.

Measuring COI Adoption. Accordingly, we measure how widely popular websites have adopted COI or the COOP header. We take the Alexa top-100 websites, and also visit every URL linked from the main page of each website. On each URL, we check the HTTP headers across four user agents: {Safari 16.0, Chrome 105} for {macOS, iOS}. For both browsers, we crawled 3,058 URLs for macOS and 2,551 for iOS (some hyperlinks are not included in mobile versions of webpages) for a total of 5,609 URLs. Finally, our web crawler did not visit webpages that require authentication.

Results and Suggestions. We find that only 113 of 5,609 URLs (2%) serve the COOP header with the correct value. Matching these 113 URLs to their corresponding Alexa top 100 websites, we find that 13 websites serve the COOP header at some page in their domain, thus resulting in a 13% adoption rate. Next, we find the user agent influences whether a server includes either header for some websites. We observed some servers including COOP headers to desktop users but not mobile users, and vice versa. While COI has corner cases with website compatibility which we describe in Appendix C, we recommend websites handling secrets to opt-in.

7.3 Limitations

Leak Rate. As shown in Table 4, our attack recovers data at rate of about 30 bits per second. While our attack's bottleneck is the transmission rate of pLRU covert channel with Safari's 1 ms timer, we do acknowledge our attack's relatively low leakage. Thus, we leave designing high speed covert channels that are robust enough to use low resolution timers to future work.

Inability to Cross Address Spaces. Being a Spectre-style attack, iLeakage cannot read information present in other address spaces. While early versions of Apple hardware were susceptible to Meltdown [4], there is no indication of such vulnerabilities in newer Apple CPUs. We thus leave the task of exploring cross-address space attacks on modern Apple silicon to future work.

7.4 Broader Implications

We now discuss the components of our attack which are transferable to other platforms, and offer general takeaways for designing memory-safe browser engines in the face of speculative execution.

Transferability. As all iOS-based browsers are mandated to use WebKit as their underlying engine by Apple's App Store policy, the end-to-end exploit chain of iLeakage affects all iOS browsers beyond Safari. Furthermore, beyond the Apple ecosystem, some of Samsung's mobile and embedded devices use the Tizen operating system, whose default bundled browser uses WebKit: we project these devices to be exploitable in theory, although we leave the microarchitectural primitives to future work since their CPUs are significantly different from Apple CPUs. Finally, our microarchitectural primitives, such as the timerless cache hit/miss distinguisher gadget using race conditions, are agnostic to browser engine, as we have shown with our timerless Spectre-v1 proof-of-concept (PoC) on Firefox and Tor Browser in Section 4.6.

Memory Safety Under Speculation. Both the Spectre-v1 PoCs and the end-to-end Safari exploit were made possible when assumptions about memory safety that hold true architecturally failed to hold under speculation. While it is well known that assuming variables will be in-bounds following a length check leads to Spectre-v1, we uncover more assumptions and implementation details which warrant re-evaluation. More specifically, while Safari's design is sufficient to prevent architectural type confusion, our work shows that speculative type confusion is still possible. Thus, we argue that speculative memory safety must be considered separately from architectural memory safety during system design.

Firstly, assuming objects will be of the correct type following a check may lead to sandbox escapes, even in the presence of standard Spectre countermeasures. Secondly, implementing optimizations or corner cases based on type may also become affected by misspeculations on type checks, even if they guarantee perfect security under architectural execution. Finally, making allocations such that attributes of objects which are essential to memory safety (e.g., type or length) map to a different cache line than the data of objects is risky, as the CPU can speculate past these safety checks when the object is partially evicted.

8 Conclusion

In this paper, we study the side-channel resilience of recent Apple CPUs. Overcoming the lack of high-resolution timers in both native environments and JavaScript in the Safari web browser, we introduce primitives to mount cache attacks on degraded timers and notably without timers, after empirically measuring the cache organization, inclusiveness, and speculation depth. Furthermore, we show these primitives are transferable with Spectre-v1 proof-of-concepts in several browsers, and present algorithmic improvements for finding eviction sets with our new timing primitives. Subsequently, we show that numerous architectural invariants for memory safety in Safari's design and implementation break under speculation, resulting in an end-to-end speculative type confusion primitive with arbitrary 64-bit read capabilities. We demonstrate the practicality and severity of this attack with popular real-world scenarios and targets. Finally, we suggest several mitigations and takeaways for web browser vendors.

Acknowledgments

This research was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; an ARC Discovery Early Career Researcher Award DE200101577; an ARC Discovery Project number DP210102670; the Defense Advanced Research Projects Agency (DARPA) under contract HR00112390029, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972; the National Science Foundation under grant CNS-1954712. and a gift by Qualcomm.

References

[1] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. 2022. Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution. In *IEEE SP*.

[2] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating point and Abnormal Timing. In *IEEE SP*.

[3] Apple. 2011. WebKit2. https://trac.webkit.org/wiki/WebKit2.

[4] Apple. 2018. About speculative execution vulnerabilities in ARM-based and Intel CPUs. https://support.apple.com/en-us/HT208394.

[5] Apple. 2021. `xnu/osfmk/arm/kpc_arm.c`. https://github.com/apple-oss-distributions/xnu/blob/e6231be02a03711ca404e5121a151b24afbff733/osfmk/arm/kpc_arm.c.

[6] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *USENIX Security*.

[7] Eloi Benoist-Vanderbeken and Fabien Perigaud. 2019. WEN ETA JB? A 2 million dollars problem.

[8] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *CCS*.

[9] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*.

[11] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. 2018. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 178–194.

[12] Intel Corporation. 2021. Intel Transactional Synchronization Extensions (Intel TSX) Memory and Performance Monitoring Update for Intel Processors. https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html.

[13] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2021. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In *USENIX Security*.

[14] Alexis Deveria. 2022. Shared Array Buffer | Can I use... Support tables for HTML5, CSS3, etc. https://caniuse.com/sharedarraybuffer.

[15] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime + Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security*.

[16] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS*.

[17] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *IEEE SP*.

[18] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. 2018. Drive-By Key-Extraction Cache Attacks from Portable Code. In *ACNS*.

[19] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative probing: Hacking blind in the Spectre era. In *CCS*.

[20] Google. 2021. Spectre. https://leaky.page.

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

[22] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*.

[23] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: a fast and stealthy cache attack. In *DIMVA*.

[24] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–Bringing access-based cache attacks on AES to practice. In *IEEE SP*.

[25] Gregor Haas, Seetal Potluri, and Aydin Aysu. 2021. iTimed: Cache attacks on the apple a10 fusion SoC. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 80–90.

[26] Noam Hadad and Jonathan Afek. 2018. Overcoming (some) Spectre browser mitigations. https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/.

[27] Lorenz Hetterich and Michael Schwarz. 2022. Branch Different-Spectre Attacks on Apple Silicon. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 116–135.

[28] Jann Horn. 2018. Speculative Execution, Variant 4: Speculative Store Bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[29] IEEE Standard Board. 2008. *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2008. IEEE. https://doi.org/10.1109/IEEESTD.2008.4610935

[30] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing–and its Application to AES. In *IEEE SP*.

[31] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security*.

[32] Dougall Johnson. 2021. Apple M1 microarchitecture research. https://dougallj.github.io/applecpu/firestorm-int.html.

[33] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. 2023. The gates of time: Improving cache attacks with transient execution. In *USENIX Security*.

[34] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2016. A High-Resolution Side-Channel Attack on Last-Level Cache. In *DAC*.

[35] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv preprint arXiv:1807.03757* (2018).

[36] Ofek Kirzner and Adam Morrison. 2021. An Analysis of Speculative Type Confusion Vulnerabilities in the Wild. In *USENIX Security*.

[37] Eiji Kitamura. 2022. Load cross-origin resources without CORP headers using 'COEP: credentialless'. https://developer.chrome.com/blog/coep-credentialless-origin-trial/.

[38] Eiji Kitamura. 2022. Making your website "cross-origin isolated" using COOP and COEP. https://web.dev/coop-coep/.
[39] Eiji Kitamura and Domenic Denicola. 2021. Why you need "cross-origin isolated" for powerful features. https://web.dev/why-coop-coep/.
[40] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE SP*.
[41] David Kohlbrenner and Hovav Shacham. 2016. Trusted browsers for uncertain times. In *USENIX Security*.
[42] David Kohlbrenner and Hovav Shacham. 2017. On the Effectiveness of Mitigations Against Floating-Point Timing Channels. In *USENIX Security*.
[43] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. In *WOOT*.
[44] Daniel Lemire. 2021. Counting Cycles and Instructions on the Apple M1 Processor. https://lemire.me/blog/2021/03/24/counting-cycles-and-instructions-on-the-apple-m1-processor/.
[45] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take a way: Exploring the security implications of AMD's cache way predictors. In *Asia CCS*.
[46] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*.
[47] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE SP*.
[48] Andrei Lutas and Dan Lutas. 2019. Security Implications of Speculatively Executing Segmentation Related Instructions on Intel CPUs. https://businessresources.bitdefender.com/hubfs/noindex/Bitdefender-WhitePaper-INTEL-CPUs.pdf.
[49] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *CCS*.
[50] Hector Martin. 2021. M1ssing Register Access Controls Leak EL0 State. https://m1racles.com/.
[51] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. CoRR abs/1902.05178.
[52] MDN Contributors. 2022. performance.now() - Web APIs - MDN. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now#reduced_time_precision.
[53] Mozilla. 2021. Project Fission. https://wiki.mozilla.org/Project_Fission.
[54] Nick Nguyen. 2017. The Best Firefox Ever. https://blog.mozilla.org/blog/2017/06/13/faster-better-firefox/.
[55] Ryosuke Niwa. 2018. Reduce the precision of "high" resolution time to 1ms. https://github.com/WebKit/WebKit/commit/25e575313d12e97a9e6c2b1d9b6ddd1d510e01a9.
[56] 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 *CCS*.
[57] John Pascoe. 2023. Process swap on cross-site window.open behind a flag. https://github.com/WebKit/WebKit/pull/10169.
[58] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan*.
[59] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. 2023. ShowTime: Amplifying Arbitrary CPU Timing Side Channels. In *ACM AsiaCCS 2023*.
[60] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *CCS*.
[61] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. 2021. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security*.
[62] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. Crosstalk: Speculative data leaks across cores are real. In *IEEE SP*.
[63] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: Attacking Arm Pointer Authentication with Speculative Execution. In *ISCA*.
[64] Charles Reis, Adam Barth, and Carlos Pizano. 2009. Browser Security: Lessons from Google Chrome: Google Chrome developers focused on three key problems to shield the browser from attacks. *Queue* 7, 5 (2009), 3–8.
[65] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site isolation: process separation for web sites within the browser. In *USENIX Security*.
[66] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security*.
[67] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. Prime + Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *USENIX Security*.
[68] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security*.
[69] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv preprint arXiv:1806.07480* (2018).
[70] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.
[71] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE SP*.
[72] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security*.
[73] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX fails in practice. https://sgaxe.com/files/SGAxe.pdf.
[74] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. Rogue In-flight Data Load. In *IEEE SP*.
[75] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *USENIX Security*.
[76] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. 2022. Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest. In *IEEE SP*.
[77] Pepe Vila, Boris Köpf, and José F Morales. 2019. Theory and practice of finding eviction sets. In *IEEE SP*.
[78] Yoav Weiss and Eiji Kitamura. 2021. Aligning timers with cross origin isolation restrictions. https://developer.chrome.com/blog/cross-origin-isolated-hr-timers/.
[79] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-order Execution. https://foreshadowattack.eu/foreshadow-NG.pdf.
[80] Johannes Wikner, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2022. Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks. In *WOOT*.
[81] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *USENIX Security*.
[82] John Wilander. 2017. Intelligent Tracking Prevention. https://webkit.org/blog/7675/intelligent-tracking-prevention/.
[83] Haocheng Xiao and Sam Ainsworth. 2022. Hacky Racers: Exploiting Instruction-Level Parallelism to Generate Stealthy Fine-Grained Timers. *arXiv preprint arXiv:2211.14647* (2022).
[84] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.
[85] Andy Zeigler. 2008. SharedArrayBuffer updates in Android Chrome 88 and Desktop Chrome 91. https://docs.microsoft.com/en-us/archive/blogs/ie/ie8-and-loosely-coupled-ie-lcie.
[86] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. 2023. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In *USENIX Security*.

A Multi-Process Web Browsers

Modern browsers employ a multi-process architecture with unprivileged rendering processes for each webpage [3, 54, 64, 85]. These effectively compartmentalize the browser, limiting the damage done in case of an exploit inside the browser's rendering engine. Instead of grouping pages arbitrarily, Site Isolation [65] uses the page's URL, aiming to separate domains from each other. Spectre's discovery [40] further accelerated the deployment of site isolation, in an attempt to create address space separation between data from mutually-distrusting domains.

eTLD+1 Consolidation. Site isolation in Chrome and Firefox groups websites into rendering processes based on their effective top-level domain plus one subdomain (eTLD+1). More specifically, a rendering process will only handle websites that share their eTLD+1. For example, `example.com` and `example.net` will be housed in different rendering processes, as their top-level-domains,

`.net` and `.com`, are different. Likewise, `target.com` and `attacker.com` are also separated, as their first sub-domains (example and attacker) are different. Finally, `store.bigbiz.com` and `accounts.bigbiz.com` might share a rendering process, since they both share the same eTLD+1, `bigbiz.com`.

B Safari's Object Layout

Safari is a 64-bit application running on 64-bit hardware. Despite this, Safari uses 32-bits to represent array indices, and 35-bits (which can address 32 GB) to represent a compressed pointer to the Butterfly, which is the underlying storage holding the array data. The underlying storage for most JavaScript objects is allocated in an isolated 32 GB region of the rendering process's address space named the Gigacage. For example, array indexing happens by taking a 64-bit constant holding the base address of the Gigacage, adding the 35-bit butterfly pointer to it to obtain the array's base address, and then adding the 32-bit offset. See Figure 15. Even in a memory corruption attack, as all variables in the address computation are at most 35-bits, an attacker cannot escape the Gigacage, leaving the rest of the address space of the rendering process out of reach [7]. Indeed, we find that DOM secrets do not reside in the Gigacage, putting them out of reach for Spectre-v1 attackers.

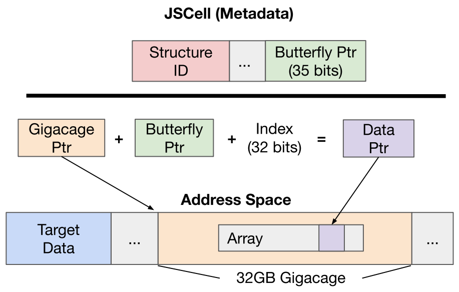


Figure 15: Layout of the `JSCell` class holding the metadata for JavaScript objects, and address computation for array indexing in Safari. The resulting address is confined to the Gigacage.

C Shortcomings of Cross-origin Isolation

COOP is problematic for many webpages offloading user-specific functionality to third parties, such as federated logins or payment processing, as it prohibits cross-window communication. To address this, W3C has suggested allowing COI webpages to open popups [38], though such an extension has not yet been codified. Also, COEP prohibits news and media websites from loading images from a CDN, as the image must opt into every such site. In response, COI is being updated with a 'credentialless' mode [37] to allow unauthenticated cross-origin requests. However, only Google Chrome and Microsoft Edge currently support this.