iLeakage: Browser-based Speculative Execution Attacks on Apple Devices

Jason Kim
Georgia Tech

Stephan van Schaik
University of Michigan

Daniel Genkin
Georgia Tech

Yuval Yarom
University of Adelaide

Abstract

Over the past few years, the high-end CPU market is undergoing a transformational change. Moving away from using x86 as the sole architecture for high performance devices, in recent years we witness the introduction of heavy-weight Arm processors in highly-performant mobile devices. 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, this change is yet to percolate to side-channel research. While significant effort has been invested analyzing and attacking x86-based architectures, the Apple ecosystem remains largely unexplored. This also applies to the Safari web browser, which despite being the only browser engine allowed on iPhone and iPad devices, did not receive much research attention with regards to its side channel resilience.

In this paper, we set out to investigate the resilience of the Apple ecosystem to speculative side-channel attacks. We first establish the basic primitives needed for mounting side-channel attacks, such as the structure and inclusiveness of the cache hierarchy. We then present a novel speculation-based primitive for distinguishing cache misses from cache hits, despite the inability to measure cache latency due to Apple degrading the timer resolution in both native and browser-based code. Finally, we use our distinguishing primitive to construct eviction sets, avoiding the need for an accurate timer.

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 leak any 64-bit address within the address space of the Safari 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

Over the past decades there has been an exponential growth in the complexity of computer systems. Starting from humble origins as "fancy calculators", modern processors have billions of transistors, multiple execution cores, and a complex interconnection structure. While beneficial to performance, this increase in complexity also gave rise to numerous hardware vulnerabilities [1, 11, 21, 28, 29, 33, 37, 39, 44, 66, 67, 68, 76, 81, 82, 84, 89, 95, 96], breaking nearly all hardware-backed security domains.

The ubiquity of computing devices has also 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 [2, 3, 16, 23, 24, 26, 30, 49, 50, 51, 58, 60, 65, 70, 78, 79, 93]. 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 [26, 59, 72, 79, 87]. With the advent of Apple's M-series CPUs, and complete reliance on the Safari browser for iOS devices, we investigate the following main questions in this paper:

How secure are Apple devices against browser-based side-channel attacks? In particular, how can attackers mount transient execution attacks inside the Safari browser?

1.1 Our Contributions

In this paper, we present iLeakage, a speculative type-confusion attack that can extract information from Apple's Safari web browser. In particular, we can defeat Apple's low-resolution timer, compressed 35-bit addressing, and value poisoning countermeasures, allowing us to speculatively read and leak 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 can 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 while Safari is commonly used by MacOS users, 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.

However, as little is known about transient execution attacks on Apple silicon, before constructing we must first construct common side-channel primitives on Apple silicon. Given the growing popularity of Apple hardware, these might be of independent interest.

Investigating Cache Topology. We begin by investigating the topology of the cache hierarchy on Apple CPUs. As this information is not readily available via hardware performance counters, we design a set of experiments that allow us to recover the total size, associativity, cache line size, and inclusiveness of the L1-D, L1-I, and L2 caches.

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 technique that reliably distinguishes hits from misses using both the native 42 ns timer and even Safari's 1 ms timer.

Constructing Eviction Sets. We then shift to constructing cache eviction sets. Here, we improve prior work by Vila et al. [88], presenting a new group testing and backtracking approach which allows the algorithm to converge within seconds even without cycle-accurate timers. As degrading the timer resolution is often seen as a countermeasure to eviction set construction, this technique might be of independent interest for mounting browser-based cache attacks.

Mounting Transient Execution Attacks in Safari. Using the side-channel primitive we have constructed, 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 open arbitrary victim pages that become rendered in its address space, simply by opening them using the JavaScript `window.open` API. We then construct in-browser eviction sets, despite Safari's 1 ms timer. 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.

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.

Summary of Contributions. In this paper we make the following contributions:

- We study the cache topology on Apple CPUs (Section 4.3).
- We present a new speculative-execution technique to distinguish cache hits from misses even in the case of low resolution timers (Section 4.4).
- We tackle the problem of constructing eviction sets in the case of low resolution timers, adapting prior approaches to work in this setting (Section 4.5).
- 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 have identified contacts within Apple's product security team and plan to share our findings with them after submission. 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 overcome 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. In this work we primarily focus on Apple CPUs with a L1-D cache per core and a L2 cache shared within a 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, with the most notable being FLUSH+RELOAD [31, 32, 96] and PRIME+PROBE [19, 41, 43, 56, 65, 67, 69, 83, 88].

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 outcome and speculatively executes instructions accordingly. 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 [49] and Meltdown [54], demonstrated 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 [8, 10, 12, 13, 22, 25, 37, 42, 44, 52, 54, 55, 57, 58, 71, 80, 81, 82, 84, 85, 86, 91, 92].

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. [79] demonstrate cache capacity attacks on M1 processors while Leaky.page [26] and Hetterich et al. [36] established the feasibility of Spectre v1 exploits on these machines.

More recently, the m1racles [59] attack demonstrated a cross-process covert channel due to insufficient access control to system registers. Next, Augury [87] demonstrated the existence of data dependent memory prefetching (DMP) on M1 processors, allowing attackers to bypass some Spectre countermeasures as well as derandomize the kernel's ASLR. Finally, PacMan [72] shows how attackers can forge kernel pointer authentication codes 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 [24], monitoring user activity [65, 78, 79], and even to cause Rowhammer [16, 23, 30]. Next, the data dependency of floating point instructions was also exploited in the browser, in the form of pixel stealing attacks [3, 50, 51]. Finally, early transient execution vulnerabilities have also been demonstrated in browsers [49, 58, 60], prompting large mitigation efforts.

Transient Execution Attacks in Chrome. With the demonstration of the original Spectre [49] attack in JavaScript, Google attempted to harden Chrome against transient execution attacks. This effort led to site isolation, where websites originating from different eTLD+1 domains are rendered in different address spaces (see Appendix A for a deeper discussion). More recently, Leaky.page [26] 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 [2] overcomes this 4 GB restriction, allowing attackers to read browser secrets assuming the attacker and victim pages share the same eTLD+1 domains.

Transient Execution Attacks in Firefox. Transient-execution attacks have also been demonstrated within modern versions of the Firefox web browser. Ragab et al. [70] 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 [93] exploits mispredictions in the return stack buffer (RSB) to again demonstrate type-confusion attacks on Firefox.

2.5 Timer-Friendly Covert Channel

Transient-execution attacks typically leak values obtained during incorrect speculation via microarchitectural covert channels that persist even after the CPU reverts its speculatively updated state. However, as browser vendors typically restrict the timer resolution available to JavaScript code to around 100 μs [90], we require a "timer-friendly" covert channel, which can reliably transmit values from the speculative domain even in the case of degraded timers.

Early works have resorted to timing techniques based on the clock edge [27, 50, 77] to overcome this challenge. More recent work by Google [26] abuses the pseudo least recently used (pLRU) eviction strategy of L1-D caches to construct a stable covert channel. Appendix C documents this approach, adapting it to Apple's Arm CPUs with a transmission rate of 663.33 B/s using a 1 ms timer (see Figure 2 for more details).

For the remainder of this paper, we abstract the pLRU covert channel as providing three primitives `plru.init()`, `plru.transmit()`, and `plru.receive()` which (respectively) initialize the covert channel, transmit a fixed value via some microarchitectural state, and receive the transmitted value using only a low-resolution timer.

3 Threat Model

In this paper we focus mainly on Apple hardware, based on both Intel and Arm CPUs. For the experiments presented in Sections 5 and 6, we assume that the target has been fully updated with Apple's MacOS 12.5 and iOS 15.6 (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:

[$\mathcal{P}_1$] Controlling CPU Affinity. As Apple's recent Arm-based CPUs feature both performance and efficiency cores, we need to control which core is running our experiments.

[$\mathcal{P}_2$] Cache Organization. In addition, we need to determine the organization of the cache, as our micro-architecutral 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.

[$\mathcal{P}_3$] Distinguishing L2 Hits/Misses. Tackling Apple's degradation of timer resolution on Arm cores, we show a method to use speculation in order to distinguish cache misses from cache hits using only low resolution timers.

[$\mathcal{P}_4$] Minimal Eviction Sets. We then use **[$\mathcal{P}_3$]** to develop a primitive to reliably find minimal eviction sets. We improve on the group testing algorithm [88] in ways that add resiliency, such that we can integrate **[$\mathcal{P}_3$]**.

[$\mathcal{P}_5$] L2 Inclusiveness. Finally, before we can use **[$\mathcal{P}_4$]**, we present a collection of primitives that help determine whether the L2 cache is inclusive of the L1 caches.

4.1 [$\mathcal{P}_1$]: Controlling CPU Affinity

Since most of Apple's Arm-based CPU designs (even preceding the M series) feature both P-cores aiming for performance and E-cores aiming for energy efficiency, we must ascertain that our experiments are running on the appropriate CPU cores. To that aim, we use the `pthread_set_qos_class_self_np()` function with `QOS_CLASS_USER_INTERACTIVE` and `QOS_CLASS_BACKGROUND` parameters in order to tell MacOS and iOS that we want to run our process on the P-core and E-core, respectively.

4.2 Timers on Native and Browser

A common requirement for mounting microarchitectural attacks is the capability of measuring the execution time of different instructions. We now survey the state of time measuring capabilities in the Apple ecosystem.

Hardware. Just like their PC counterparts, Intel based x86 Macs provide the `rdtsc` and `rdtscp` instructions which expose cycle-accurate timers to micro-architectural attacks using native code. For Arm-based hardware however, finding a good timer source turns out to be challenging. Even though the ARMv8-A architecture defines a standardized Performance Monitor Unit (PMU) interface [7], Apple CPUs provide a custom PMU implementation instead. Moreover, the PMU can only be configured from supervisor mode (`EL1`) and higher, thereby preventing its use for micro-architectural attacks from unprivileged native code.

Function Calls. Both MacOS and iOS provide a monotonically increasing counter as well as an interface to retrieve the current time and date (i.e., wall-clock time) via `clock_gettime()` and `mach_absolute_time()`. While both of these aim to retrieve the current time with an accuracy of nanoseconds, when attempting to use these timers to differentiate between cache hits and misses we discovered that the timers report similar times for both events. While we are unable to precisely point to the reason for this, we conjecture that Apple had deliberately attempted to make these timers unusable for micro-architectural side channels.

Establishing a (Privileged) High Resolution Timing Source. However, we observe that both macOS and iOS implement the kperf API for instrumentation purposes [6, 53], which allows privileged users to configure and read the PMU counters on Apple silicon. Using this interface, we were able to get a cycle-accurate timer which is accessible via native code running at an elevated privilege level (i.e., `root`).

Unprivileged Timer Resolution. To establish the resolution of various unprivileged timers, we collect a timestamp from the timer we would like to measure, use the wall-clock time to wait for one second, and then collect another timestamp. By subtracting the timestamps, we then get the amount of ticks that passed in a second, i.e. the true resolution of the measured timer. We found that the system calls provided by MacOS such as `mach_absolute_time()` run at a resolution of 42 ns (24 MHz), while the privileged cycle counters on the PMUs run at a frequency of up to 3.2 GHz on the Apple M1, depending on the CPU frequency.

Timer Resolution in 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 [61, 64, 90]. Here, Google Chrome provides a timer resolution of 100 μs, while Mozilla's Firefox and Apple's Safari provide a 1 ms timer.

In older versions of all major browsers, it was possible to craft a timer with 5 ns resolution [77] 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` [17].

Yet, `SharedArrayBuffer` was conditionally re-enabled in all major browsers by December 2021 for webpages served with cross-origin isolation [17]. Cross-origin isolation is a set of HTTP response headers that prevents a webpage from loading cross-origin content not explicitly allowed by the server that provided the page, and was purposefully designed as a Spectre countermeasure for the web [48]. For additional depth, we discuss the implications of this condition on the Safari browser in Section 5.1.

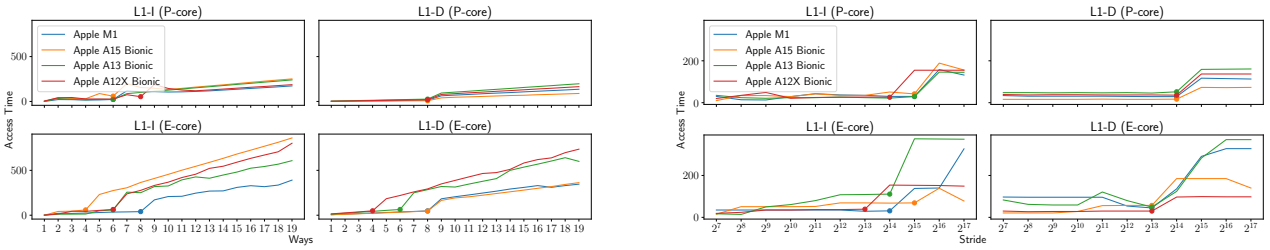

Figure 1: The access times when accessing the same offset in a loop for a varying number of pages to determine the ways (on the left) and the stride (on the right), and thus the size, of the L1-D cache set and the L1-I cache set on the P-cores and E-cores of various Apple CPUs. The round markers indicate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

4.3 [$\mathcal{P}_2$]: Cache Organization

We now proceed to reverse engineer the cache topology of Arm-based Apple CPUs. While writing a MacOS kernel extension that reads the `CLIDR_EL1`, `CSSELR_EL1` and `CCSIDR_EL1` registers present in the ARMv8-A ISA is possible, accessing these registers on iOS devices (especially non-jailbroken) is rather challenging. In addition, these registers do not disclose whether caches are inclusive or not, i.e., whether the L2 cache includes cache lines allocated in lower levels of the cache hierarchy. Thus, we now describe our empirical method to determine the cache topology.

Determining the Associativity of the L1 Cache. To find the associativity of L1 instruction and data caches, we first note that L1 caches typically use the page offset bits of the virtual address to select the cache set. Thus, by allocating a number of pages, and accessing the same page offset in each page, we can access the same L1 cache set. To determine the associativity, we simply access a number of elements in a loop, while measuring its execution time. When the number of elements accessed exceeds the size of the cache set, we will notice a significant slowdown during the loop's execution. Thus, by increasing the number of elements accessed by the loop one at a time, we can determine the size of the L1-D cache set and, therefore, its associativity.

For the instruction cache (i.e., L1-I), instead of accessing the cache lines directly, we use chains of branch instructions to jump from one instruction to the next, making sure that all instructions are placed at the same page offset. We then measure the time it takes to execute this chain, expecting an increase in execution time when the amount of jumps exceeds the cache's associativity. Figure 1 (left) summarizes our findings, showing the cache associativity of the L1 caches on various Apple CPUs.

L1 Cache Size. To facilitate virtual to physical address translation in parallel to cache lookups, L1 caches typically use the page offset bits of the address to select the cache set. With Apple CPUs using 16KiB pages, we can upper-bound the size of the L1 caches by cache_size $\leq 16\,\text{KiB} \cdot$ cache_associativity. Next, cache designs often try to avoid self-eviction of blocks of continuous memory. Thus, we first allocate a memory block of size $16\,\text{KiB} \cdot$ cache_associativity, and repeatedly access cache_associativity $+ 1$ elements from it using a given stride. Having obtained an accurate average access time, we double the stride and measure the average access time again. Once self-evictions begin to occur we know that the current stride is too large to map cache_associativity $+$ 1 elements into different sets, resulting in an increased access time due to self-evictions.

Figure 1 (right) summarizes our findings, presenting the access time across different strides. As the largest stride with fast access times still allows cache_associativity $+ 1$ elements to map to different sets, this stride in fact corresponds to the capacity of a single cache way. Thus, the side of the entire cache can be computed via multiplying the largest stride with fast access times via the cache's associativity.

L1 Cache Line Size. To determine the cache line size, we again start from the fact that accessing cache_associativity $+ 1$ elements with a distance of page size bytes in between them results in a slowdown. If we offset half of these elements by the cache line size, we will observe a speed up as half of the elements will map to the adjacent cache line and thus to the adjacent cache set. We simply try adding increasingly large offsets to half of the elements until we find some offset such that results in a speed up. From this we can then derive that for the lowest offset, for which we observe the speed up, will equal the cache line size. Finally, once we know the cache size, the cache line size, and the cache's associativity, the number of sets can be calculated by number_of_sets = cache_size/(cache_line_size $\cdot$ ways).

L2 Cache Organization. To determine the cache size and the cache line size of the L2 cache, we use the `sysctl` interface which provides some information about the kernel configuration, the hardware configuration, etc. including the cache sizes as well as the L2 cache line size on both macOS and iOS. However, as we cannot retrieve the associativity of the L2 cache or the number of sets, we instead have to rely on the ability to find minimal eviction sets that we will further elaborate in Section 4.5. As the size of a minimal eviction set for the L2 cache is equal to its associativity, we use this algorithm to determine the associativity.

Finally, from the cache size, the cache line size, and the cache's associativity we compute the number of cache sets by number_of_sets = cache_size/(cache_line_size · ways).

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.

4.4 [$\mathcal{P}_3$]: Observing L2 Hits and Misses

In the previous section, we could use low-resolution timers such as `mach_absolute_time()` for our measurements, as we were measuring the execution time of a loop taking long enough for it to be distinguishable using such a timer. While this method was sufficient to reverse engineer the structure of L1 caches, we cannot apply it to the L2 cache, as it uses bits from the physical address for cache set selection. Furthermore, in a non-privileged environment, such as iOS, we cannot view the virtual to physical mapping, nor can we rely on high-resolution timers to distinguish a single L2 cache hit from a single miss for individual addresses.

Thus, in this section we develop a technique for distinguishing L2 cache hits from misses for individual addresses, which we employ for cache reverse engineering in subsequent sections. This technique is also suitable for other environments with restricted timer resolution, such as in Safari (see Section 5), and is thus of independent interest.

A Timer-Friendly pLRU-Based Covert Channel. We first recall the pLRU-based channel outlined in Section 2.5. We implement a covert channel on the Apple M1 that transmits a 1 by accessing the same cache set and a 0 by not accessing the cache set and use the pLRU gadget to monitor the L1 cache set. We transfer 10K bits over the covert channel, running each experiment five times. Figure 2 demonstrates that this achieves a transmission rate of 663 B/s despite a severely degraded timer resolution of 1 ms.

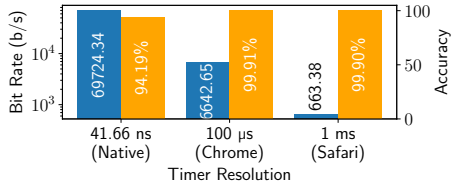

Figure 2: Bit rate (blue) and accuracy (yellow) of our pLRU-based covert channel across different timer resolutions.

Distinguishing L2 Hits From Misses Using Speculation. Recall that in case of a conditional branch instruction that depends on data that is not yet available, the CPU tries to execute subsequent computations speculatively. Even more so, the number of instructions that the CPU speculatively executes depends on where the data is located, with further away structures resulting in a longer speculation window.

Applying this observation to the cache hierarchy, we notice that it is quicker to resolve a speculated branch if the data is present in the L1 caches, compared when the data is in a higher-level cache or the main memory. Thus, if we place a covert channel transmission routine just at the end of a long instruction stream that barely fits in a speculation window created by the mispredicted branch, we observe a covert channel transmission only if the branch's condition indeed misses the L2 cache. This is because when the branch's data is closer, the CPU will revert the speculation before it can execute the covert channel transmission routine, resulting in no signal being transmitted. Finally, by executing the covert channel receive function after the speculation, we can architecturally determine if a transmission happened, leaking the length of the speculation window and consequently the location of the branch's data (i.e., cache hit or miss).

Gadget Overview. We present the pseudocode of our extended pLRU gadget in Listing 1. After initializing the pLRU channel (Line 1), the gadget branches on the value of the address to which `target` points (Line 3). This causes the CPU to speculate if the address, to which `target` points, is not available at the CPU core. Within the if-body (lines 4 – 10), we cause the CPU to execute a sequence of junk instructions[1] to delay the pLRU transmission routine until the end of the speculation window. Finally, in Line 13 we measure the state of the pLRU channel, outputting the `status` of the address to which `target` points.

Note that if the value to which `target` points is located too close to the CPU core (e.g., a cache hit), the speculation window will be too small for the CPU to reach the `plru.transmit(1)` routine. However, if the value to which `target` points has to be served from a higher-level cache or the main memory, the speculation window will be large enough that the CPU will actually reach and execute the `plru.transmit(1)` routine. Next, we use the `plru.receive()` routine to architecturally determine whether speculation reached `plru.transmit(1)`, giving us an indication of how far the CPU speculated. Finally, we can use the information about the length of the CPU's speculation to distinguish a L2 hit from a L2 miss, which allows us to measure the latency of individual memory accesses even in the case of severely degraded timers.

Experimental Results. Demonstrating this, Figure 3 presents the accuracy of our pLRU-based speculation gadget in determining L2 hits and L2 misses across 3,000 trials for different timer resolutions. As can be seen, our gadget is able to distinguish L2 hits and misses for individual variables using both the 42 ns clock available to unprivileged native code running on Arm-based Apple devices and the 1 ms clock available to JavaScript code in the Safari web browser.

[1] In practice, our slide is not a series of `nop`s, but different instructions as `nop`s maybe re-ordered or optimized-out by the CPU. We use `nop`s here for illustration purposes.

Name	CPU	L1-I Cache W	L1-I Cache S	L1-I Cache Size	L1-I Cache CL	L1-D Cache W	L1-D Cache S	L1-D Cache Size	L1-D Cache CL	L2 Cache W	L2 Cache S	L2 Cache Size	L2 Cache CL
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 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*	4096*	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 [34], 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.

```
plru.init()
// Speculates taken if target is not cached.
if (*target != 0) {
    nop;
    nop;
    // Speculation ends here if target is not evicted from the cache.
    ...
    nop;
    plru.transmit(1);
    // Speculation reaches here only if target is evicted from the cache.
}
status = plru.receive();
```

Listing 1: Our speculation-based eviction testing method.

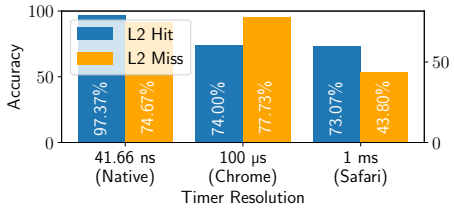

Figure 3: Accuracy of distinguishing L2 cache hits and misses using our speculation-based eviction testing method across different timer resolutions.

4.5 [$\mathcal{P}_4$]: Finding L2 Eviction Sets

Given an extended speculation gadget that can distinguish L2 hits from L2 misses, we can use this gadget to find L2 minimal eviction sets from an unprivileged user, even in environments that only provide low-resolution timers. As browser-based code does not have access to any `clflush` instructions, finding minimal eviction sets in timer-restricted environments is a common side-channel task, making the technique presented in this section of independent interest for browser applications.

Previous Works. The idea behind finding a minimal eviction set to evict a specific address first starts with an inflate step, where we keep allocating elements until accessing these elements evicts the victim address. The output of the inflate step is called a conflict set, which is a set of elements that is capable of evicting the victim address, but not necessarily a minimal one. Next, in order to retrieve the minimal eviction set out of the conflict set, we then perform a reduce step. Here, prior works [19, 56, 65, 83] removed elements from the conflict set one at a time, and then test if the set of elements is still able to evict the victim address.

Group Testing Approach of [88]. Vila et al. [88] use group testing to improve this approach in the case that the cache's associativity is known. The improved reduction step divides the conflict set in ways + 1 equally-sized bins, removes one bin at a time and tests if the remaining bins are still able to evict the victim address. In the case that a certain bin is found to be redundant, i.e., where the remaining bins still evict the victim address, the algorithm removes this bin and immediately proceeds with the next reduction step by redistributing the remaining elements of the conflict set across new ways + 1 equally-sized bins.

Vila et al. [88] argue that such a redundant bin can always be found, as the minimal eviction set must contain ways elements, while the algorithm always maintains ways + 1 bins. Thus, there is always one bin that is not required to evict the victim address, making it redundant. Overall, this technique discards $n/(\text{ways}+1)$ elements from the eviction set during each iteration step, until we are left with a minimal eviction set of exactly ways elements.

Improving the Reduction. We improve this technique by generalizing it to operate on some upper bound k on the cache's associativity, rather than the exact number of ways. Then, during each reduction step, we divide the conflict set into k equally-sized bins and proceed to test each bin. As before, we test each bin for redundancy, i.e. if the remaining bins are still able to evict the victim address. Next, if a bin is found to be redundant, we remove the bin's elements from the conflict set. However, here we deviate from [88] and instead of stopping the test and redistributing the elements, we keep testing the remaining bins for redundancy. Thus, at the end of each reduction step we are left with the minimal set of bins required to evict the victim address. Only then do we redistribute the remaining elements into k new bins, and proceed to the next reduction step.

We argue that this approach is more resilient, and thus more effective when used with our extended pLRU gadget, as in case k is large (e.g., 64), most bins are not necessary on average in order to evict the victim. Thus, our testing approach filters out about $k-$ways bins on average per reduction step, instead of just a single bin. Overall, we empirically observe that this significantly improves the resiliency of finding eviction sets, because we perform additional tests for redundant bins at every iteration.

Improving the Backtracking. Noise sometimes causes the reduction step to remove bins that are essential to evict the victim address. Vila et al. [88] remediate this by performing a backtracking step that adds the last removed bin back to the eviction set. We found that this backtracking approach is susceptible to noise, especially when used with the speculation-based gadget from Section 4.4, which often results in not converging to a minimal eviction set.

We improve the backtracking strategy by testing if the conflict set that we retrieved after the reduction step can still evict the victim address. If it cannot, we re-allocate as many of the previous elements until we double the number of elements in our conflict set. We repeat this until our set of elements can evict the victim address again, which generally only requires a few iterations. However, as our reduction step is more aggressive than the backtracking, we find that this converges very quickly despite the additional noise introduced by the speculation-based gadget from Section 4.4.

Eviction Testing. To guarantee proper eviction of the L2 cache, we rely on the pointer chasing technique as described by [83, 94] by embedding a bidirectional linked list within the cache lines of the eviction set. Then we traverse the linked list using a forward, backward, forward then backward pattern. Here, we employ two pointer chases with the second pointer lagging behind by eight elements.

Empirical Results. Establishing a baseline, we first implement both the group testing algorithm of [88] and our algorithm without the speculative gadget from Section 4.4, using a privileged cycle-accurate kperf timer from Section 4.2 for eviction testing. We run our experiment 20 times, and find that the group testing algorithm converges in 0.79s, whereas our algorithm converges in 2.11s. Furthermore, both algorithms find a minimal L2 eviction set that achieves an eviction rate of 100.00% for these runs.

Next, we integrate the gadget from Section 4.4, switch to using the 24 MHz timer and run this experiment 20 times. We find the group testing algorithm of [88] converges within 0.49s, with an eviction rate of 0.00%. In contrast, our approach converges in 3.08s on average and finds an a minimal L2 eviction set with an eviction rate of 100.00%.

4.6 [$\mathcal{P}_5$]: L2 Inclusiveness

With the ability to efficiently and reliably find minimal L2 eviction sets, we now ascertain that the L2 cache is *inclusive*. More specifically, a L2 cache *inclusive* of the L1 caches guarantees that any cache line present in the L1 is also present in the L2 cache. Otherwise, the L2 cache is *non-inclusive* or *exclusive*. We find that L2 cache of the Apple M1 is inclusive of the L1 caches, with improved eviction rates when accessing both halves of each L2 cache line. We discuss additional experiments regarding inclusiveness in Appendix D.

We use the various techniques described to determine the cache organization of the L1 and L2 caches on various Arm-based Apple CPUs, and present the results in Table 1.

5 Attacking the Safari Browser

In this section we 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 timer) 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 in order 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 for ensuring that the CPU does not revert the speculation early, before the attack has the opportunity to fully execute transiently and leak the sensitive data.

[$\mathcal{C}_4$] Reliability. Finally, we aim to design a reliable exploit which 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.

5.1 [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 [62, 74] thereby ensuring that pages with different effective top-level domain plus one sub-domain (eTLD+1) are handled by different rendering processes. 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 simply spawns a new rendering process for each new tab until the system runs out of memory. Empirically confirming this, we used a Macbook Air equipped with the M1 CPU and 16 GB of RAM and opened 177 tabs using MacOS 12.5 and Safari 15.6 (both latest at the time of writing). 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) running iOS 15.6 (also latest at the time of writing).

Achieving Consolidation via iFrames. Despite Safari's one process per tab model, we have observed that when a page contains an iframe, the iframe is rendered using the rendering process which handles the entire page. Thus, `attacker.com` can embed an (invisible) iframe with content from `target.com`, resulting in Safari rendering `target.com` in the same address space as `attacker.com`. While effective, we note that many popular webpages often refuse to be rendered inside embedded iframes by setting the `X-Frame-Options` HTTP response header to deny. Setting this option is in fact a recommended security measure employed by many websites, as many attacks exploit weaknesses in iframes [38, 75].

Abusing `window.open` to Achieve Consolidation. In addition to iframes, we have also observed 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 execute a `window.open` call with `target.com` as a parameter, resulting in a pop-up window rendering `target.com` appearing. Crucially, while `attacker.com` and `target.com` appear in different windows, Safari uses a single rendering process for both pages, resulting in `attacker.com` and `target.com` being placed in the same address space. Finally, unlike the case of iframes, websites have no way of refusing being rendered in separate windows following a call to `window.open`, making this technique applicable for any target website.

Consolidation, Cross-origin Isolation, and Timers. Recall from Section 4.2 that the `SharedArrayBuffer` API in JavaScript can be used for building high-resolution timers, and recent versions of all major browsers 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 rely on `SharedArrayBuffer` and craft a high-resolution timer for our attack. Accordingly, we use the pLRU covert channel described in Section 2.5.

5.2 [C_2]: Speculative Type Confusion

As outlined in Appendix B, despite being a 64-bit application, Safari uses 32-bit values for array indices and 35-bit values for pointers to the underlying storage of most objects, partitioning most JavaScript objects into 32 GB compartments internally 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 pointers. While we acknowledge prior type confusion techniques against various versions of Chrome [2, 35, 60] and the Linux kernel [45], to the best of our knowledge this is the first application of such techniques to the Apple ecosystem and to the Safari web browser.

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 4. As can be seen in Figure 4 (bottom), WebKit writes characters in a separate C-style array, and dereferences a 64-bit C-array pointer to access it (Figure 4 (middle), StringImpl).

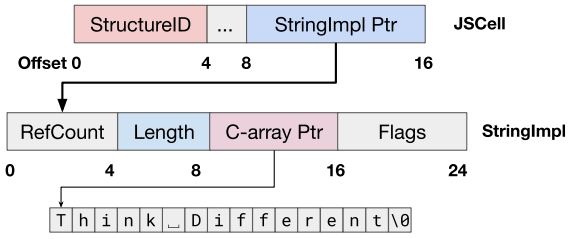

Figure 4: Memory layout of JavaScript strings in WebKit. The `JSCell` class is common to all objects, while the `StringImpl` class is the backing store specific to strings.

Performing a String Indexing Operation. More specifically, Listing 2 contains a pseudocode of JavaScript's string indexing operation, when it is compiled by WebKit's JIT compiler. The code takes as input a variable `inp`, which is a WebKit metadata object called `JSCell` that is common to all JavaScript objects. Having verified that the type inside `inp` indeed corresponds to a string (Line 3), Listing 2 then proceeds to retrieve the corresponding `StringImpl` structure (Line 7), which is 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 indeed smaller than the string's actual length (Line 9), Listing 2 retrieves the corresponding character by dereferencing the 64-bit `cArrayPtr` (Line 13).

```
1  String::operator[] (inp, index) {
2      // Check StructureID first
3      if (inp.StructureID != String::StructureID) {
4          exitToInterpreter();
5      }
6      // Dereference ptr to get StringImpl
7      StringImpl impl = *(inp.ptr);
8      // String length check
9      if (impl.length <= index) {
10         exitToInterpreter();
11     }
12     // Dereference ptr to C-array
13     return *(impl.cArrayPtr + index);
14 }
```

Listing 2: JIT-compiled WebKit string indexing operation.

A Naïve Approach to Speculative Type Confusion. It is appealing for an attacker to construct their own JavaScript object, hoping some variable will be at the same offset in memory as the C-array pointer in the JavaScript strings that we target, as we show in Figure 4 (middle). Subsequently, we show the memory layout for JavaScript objects in Figure 5. More specifically, all built-in and user-defined objects in JavaScript, can have named properties such as `obj.price` and indexed properties such as `obj[0]`. A backing store called the Butterfly stores these properties, where indexed properties reside at positive offsets relative to the Butterfly pointer and named properties at negative offsets.

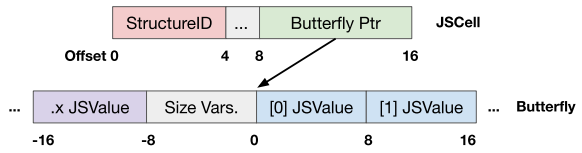

Figure 5: Memory layout of JavaScript objects in WebKit.

Inspecting Figure 4 (middle), we note that the 64-bit `C-array Ptr` is located at an 8-byte offset from the start of the string's `StringImpl` structure. Observing Figure 5 (bottom), `obj[1]` is located at the same offset from the start of the object's `butterfly` structure. See Figure 6.

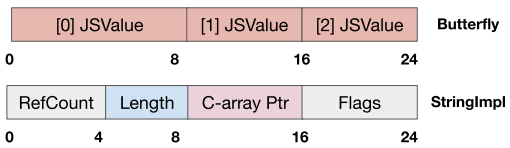

Figure 6: Side-by-side memory layout of the backing stores of a JavaScript object (top) and string (bottom).

Therefore, an attacker might attempt to control the value of `obj[1]` in the hope of achieving speculative type confusion by tricking the CPU to speculate past the type check in Line 3 of Listing 2. This will allow the attacker to dereference `obj[1]` instead of `cArrayPtr`, pointing it to arbitrary and attacker-chosen locations in Safari's address space.

NaN-Boxing. Attempting to execute this idea, an attacker needs to place some `JSValue` object inside the field corresponding to `obj[1]`. Observing Safari's code, we discover that users can set `JSValues` only to Integers, Floats, `True`, `False`, `Undefined` and `null`. Next, inspecting the bit patterns corresponding to these values, we discover Apple's countermeasure known as NaN-boxing, which ensures that values of these types can never be used as pointers. More specifically, both integers and floats have bits 63—48 set to non-zero, making them point to non-canonical unmapped virtual addresses. The other values have most of the bits zeroed out, making them unsuitable for reading arbitrary locations in Safari's address space. See Figure 19.

Bypassing NaN-Boxing. Inspecting Safari's NaN-boxing implementation, we discovered an edge case where Safari does not modify the bit patterns of `JSValues`. That is, we discovered that Safari will not modify bit patterns of objects that have no user-defined named properties, but have several indexed properties that are all floats.

More specifically, consider the `noPoison` object constructed in Figure 7 (top). As it only has three indexed properties (Lines 2-4), all containing floats, Safari will directly translate it to the memory layout described in Figure 7 (bottom) without any value poisoning.

```
1 let noPoison = {};
2 noPoison[0] = 3.14159;
3 noPoison[1] = 2.71828;
4 noPoison[2] = 1.41421;
```

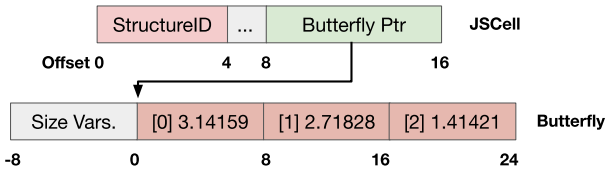

Figure 7: Initializing a WebKit object without poisoning (top) and resulting memory layout (bottom). Notice that each floating-point number is stored as-is.

In particular, Safari does not modify the value in `noPoison[1]`, which has the same offset from the beginning of the butterfly as the `cArrayPtr` field in the `StringImpl` object. See Figure 4 and Figure 7 (bottom).

Attacker Object Setup. Assume the attacker would like to dereference some 64-bit address `addr`. Recall from Figure 7 that all user-defined properties of the object must be floating-point numbers to avoid poisoning. As WebKit uses

the IEEE 754 double-precision format [40] for float encoding, the attacker must compute the inverse encoding in order to convert the bits of `addr` to a valid floating point number. Likewise, the attacker must provide a valid number for the length variable of a string, which overlaps with the lower 32 bits of the indexable property at index 0. See Figure 6 for memory layout and Figure 8 (top) for an initialization example, where the `ieee754-invert` function performs the reverse IEEE 754 conversion (i.e., from bits to floats).

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, Line 3 of Figure 8 (top) puts the value `0xffff` at offset 4 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.

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

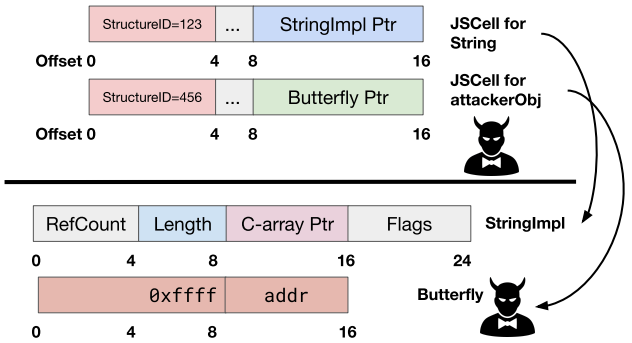

Figure 8: A malicious object to obtain a transient dereference of the target address `addr` (top), and a comparison with the memory layout of a `string` object.

Dereferencing Arbitrary 64-bit Pointers via Speculative Type Confusion. Next, in order to dereference the 64-bit pointer, consider the case where the code in Listing 2 is executed on the `attackerObj` created in Figure 8 (top) with `index` = 0. The main idea of our attack is to delay the resolution of the branch in Line 3 of Listing 2, by evicting the `StructureID` of the `attackerObj` outside the CPU's cache.

This forces the CPU to speculate forward, dereferencing `attackerObj`'s butterfly pointer and treating the resulting butterfly as a `StringImpl` structure (Line 7 of Listing 2). Next, the CPU then incorrectly uses the data in `attackerObj[0]` for string length (Line 9 of Listing 2), ensuring that `0xffff` is larger than `index`. Because we have set `index` to zero, the CPU incorrectly uses the data in `attackerObj[1]` for the string's `cArrayPtr`, dereferences it, and returns the resulting value under speculation.

Type Eviction. For the type confusion attack described above to operate, it is crucial that we delay the resolution of the branch in Line 3 of Listing 2 by as much as possible. As mentioned above, we achieve this by evicting the cache line holding the `StructureID` of the `attackerObj` outside the CPU's cache. This approach however, creates two main 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 inside the `attackerObj` inside the cache, in order to prevent the CPU's pipeline from stalling. This requires understanding WebKit's internal memory allocator, in order to place `attackerObjs` across cache line boundaries.

5.3 [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, in order 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 lowest-level cache. Thus, we cannot evict the `attackerObj`'s `StructureID` without also evicting its butterfly pointer, preventing our attack.

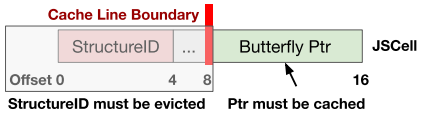

Figure 9: Ideal placement of the `JSCell`, with a new cache line between `StructureID` and `Butterfly Ptr`.

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

While user-defined `attackerObj` do not qualify for this optimization, we have discovered a suitable object in the form of `Intl.Locale`, whose memory layout is presented in Figure 10. Remarkably, although `Intl.Locale` is intended for region-specific information (e.g., fonts, numbering system, calendar, etc), JavaScript allows to initialize its indexed

properties to contain floating point numbers, in a way that facilitates type confusion. Finally, as the `JSCell` structures for `Intl.Locale` objects are 168 bytes long, when WebKit places `Intl.Locale` objects on 8-byte boundaries, it is the case that the object's `StructureID` and butterfly pointer land on different and adjacent cache lines for some of the `Intl.Locale` objects.

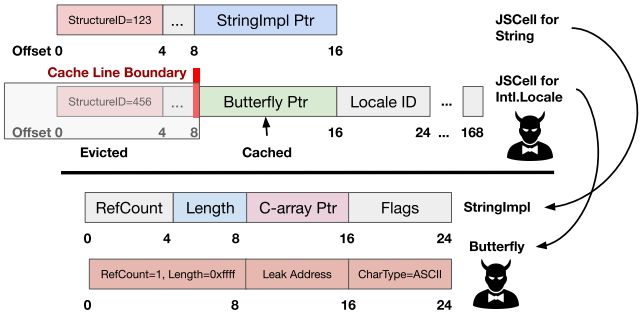

Figure 10: Comparison of memory offsets between a string object and an `Intl.Locale` object whose `JSCell` class is split across cache lines and whose index properties are initialized to create speculative type confusion.

Achieving Cache Evictions. A final prerequisite for mounting our type confusion attack is the ability to evict the `StructureID` of an `Intl.Locale` object from the cache. While Safari's 1 ms timer aims to prevent eviction set construction by hindering the attacker's ability to distinguish cache hits from cache misses, we recall the technique from Section 4.5 for constructing eviction sets using only low resolution timers. Porting a similar approach to Safari, we are able to evict `StructureID`s of `Intl.Locale` objects, thereby facilitating our attack.

5.4 Overcoming [$\mathcal{C}_4$]: 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. More specifically, Listing 3 is a pseudocode for our iLeakage attack, which we now describe in this subsection.

Setup. As our attack relies on speculatively performing type confusion between `Intl.Locale` and `String` objects, the setup phase prepares all the necessary parameters for mounting our attack. That is, Line 2 allocates parameters needed for branch predictor training while Line 4 allocates an `malObj` of type `Intl.Locale`, whose `JSCell` is split between cache lines as outlined in Figure 10.

Training Phase. Having successfully initialized our attack parameters, we now proceed to train the CPU's branch predictor. That is, the loop in Line 7 of Listing 3 proceeds to execute our attack's `gadget` function 10,000 times. During each execution of the `gadget` function (Lines 27-36), we have the

```
1  // Setup
2  let params = [0, 0, "training"];
3  // Allocate new Intl.Locale object (split
   ↪ across cache lines)
4  let malObj = new Intl.Locale("en-US");
5  // Training phase
6  // Run the gadget on valid arguments to
   ↪ mistrain the branch predictors.
7  for (let i = 0; i < 10000; i++) gadget();
8  // Attack phase
9  // Cache the malicious Intl.Locale.
10 const junk = malObj[1];
11 // Change params to malicious ones.
12 params[0] = 0xffff;
13 params[1] = index;
14 params[2] = malObj;
15 // Fake string length
16 malObj[0] = ieee754-invert(0xffff)
17 // Address to read
18 malObj[1] = ieee754-invert(addr)
19 // Evict StructureID and setup pLRU
20 evict_type(malObj);
21 plru.init();
22 // Run attack gadget on malicious params
23 gadget(params);
24 // Value Retrieval
25 return plru.receive();
26
27 // Attack Gadget
28 function gadget(params) {
29     const condVar = params[0];
30     const index = params[1];
31     // During training, this variable is a
       ↪ string. In the attack, it is a
       ↪ malicious Intl.Locale.
32     const confusionObj = params[2];
33     // The if-statement is true during
       ↪ training, false during the attack.
34     if (condVar < confusionObj.length)
35         let val = confusionObj[index];
36         plru.transmit(val);
37 }
```

Listing 3: Our speculative type confusion primitive.

variables `condVar` and `index` set to 0, while `confusionObj` holds the string "`training`" of length 8. Next, as the value of `condVar` is less than 8, the `if` statement in Line 34 of Listing 3 evaluates to `true`, resulting in the value of `val` (Line 35) being equal to '`t`'. Finally, the gadget terminates by transmitting '`t`' over the pLRU covert channel (Line 36).

Having executed the `gadget` function 10,000 times, Safari's JavaScript observes that the memory access in Line 35 of Listing 3 was performed using `string` objects. Consequently, it specializes the access in Line 35 to use the code presented in Listing 2. Moreover, the CPU's branch predictor had observed that the `if` in Line 34 was always taken, and updates its future prediction to `taken`.

Attack Phase. We begin by ensuring that our malicious `Intl.Locale` object is present in the CPU's cache (Line 10). Next, in Lines 12-14 we proceed to initialize the parameters

to our `gadget` function, pointing it to use `malObj`. We then proceed to initialize the `malObj`, setting its two indexed properties to be a fake string length (Line 16) and the address the contents of which we wish to leak (Line 18), respectively.[2] Having initialized `malObj`, we recall that this object is split between two cache lines (see Figure 10), and that it is possible to evict its `StructureID` while keeping its butterfly pointer. See Section 5.3. Having done so in Line 20 of Listing 3, we proceed to execute our `gadget` function on `malObj`.

Speculative Type Confusion. The core of our type confusion attack occurs at this phase, when `confusionObj` loads our `malObj` of type `Intl.Locale` (Line 32 of Listing 3). As we have evicted the `StructureID` of `malObj` in Line 20, when the CPU attempts to execute Line 34, it is unable to retrieve the `length` property, leading it to speculate taken in Line 34as a consequence of training.

Recall that the training phase also specialized the code for performing Line 35 of Listing 3 to use Listing 2. As outlined in Section 5.2 and Figure 10, executing Listing 2 on `malObj` results in a type confusion, causing the CPU to dereference the value of `addr` and return the value stored in this address. We then transmit (Line 36) this value over the pLRU covert channel described in Section 2.5.

Speculative Suppression. Having performed type confusion by executing the code in Listing 2 on non-string objects, the type check in Line 3 of Listing 2 will eventually realize the mismatch, triggering Safari to de-specialize the code handling the memory access in Line 35 of Listing 3 and prompting a type error. Luckily however, Line 35 is executed under speculation, due to JavaScript's inability to retrieve the `length` property of `malObj` while its `StructureID` is not cached. Once the value of `malObj.length` is architecturally retrieved, the CPU discovers that the condition in Line 34 of Listing 3 is in fact false and rolls back the incorrect speculative execution in Lines 35-36, which effectively suppresses the type error event from Safari. This preserves Safari's optimized code for string indexing (i.e., Listing 2), allowing for subsequent attack iterations.

Value Recovery. A final task of the attacker is to actually retrieve the value stored at `addr`, which was transmitted using the pLRU covert channel (Line 36 of Listing 3). We perform this task in Line 25, utilizing the fact that the pLRU covert channel is able to retrieve values even in the presence of Safari's 1 ms timer. See Section 2.5.

5.5 End-to-End Attack Evaluation

Having described the collection of techniques required in order to build our attack, we now evaluate its leak rate and accuracy across a variety of Apple devices. More specifically, we run iLeakage on unmodified Safari 15.6 and attempt to read a known 512-bit string, averaging accuracy over 10 trials. Table 2 contains a summary of our findings. As can be seen, our attack is capable of reading arbitrary address at a rate of 24 to 34 bits per second, with an accuracy of 90% to 99%.

Device	Apple CPU	Leak Rate	Accuracy
iPhone 11 (iPhone12,1) A2111	A13 Bionic	27.77 b/s	99.14 %
MacBook Air (M1, 2020) (MacBookAir10,1) A2337	M1	32.97 b/s	98.48 %
iPad Pro 12.9" 5th Gen. (iPad13,9) A2379	M1	34.78 b/s	99.57 %
MacBook Pro (14", 2021) (MacBookPro18,3) A2442	M1 Pro	33.28 b/s	90.96 %
MacBook Pro (14", 2021) (MacBookPro18,4) A2442	M1 Max	24.46 b/s	93.79 %

Table 2: End-to-end performance of iLeakage.

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 an Apple MacBook Air laptop (model A2337) with the M1 CPU and 16 GB of RAM for all attack experiments. This laptop runs macOS Monterey 12.5 and Safari 15.6, the latest versions at the time of writing. We run Safari in an out-of-the-box configuration, and with all side-channel countermeasures enabled.

Bringing Attackers and Targets Together. We begin by recalling that while Safari generally follows a strict process-per-tab model, `iframe` elements and pages opened by the `window.open` function share a rendering process with the parent page (Section 5.1). Next, as most websites refuse to render inside an `iframe`, we focus on the latter case.

More specifically, we created an attacker's page that binds `window.open` to an `onmouseover` event listener. Thus, we can open any targeted page in the attacker's rendering process whenever the user has their mouse cursor over any area of the attacker's page. 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 are able to 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.

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 are able to recover a target's text messages without attacking their mobile phone itself. See Figure 12.

IP Address and Geolocation. Finally, an attacker might

[2] Recall that one must have these values be encoded at IEEE-754 Floats, as explained in Figure 8.

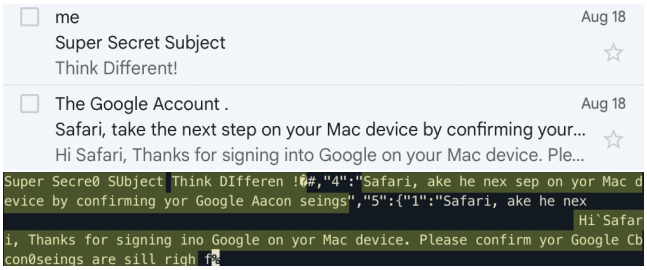

Figure 11: (Top) An email displayed in Gmail's web view. (Bottom) Recovered sender address, subject, and content.

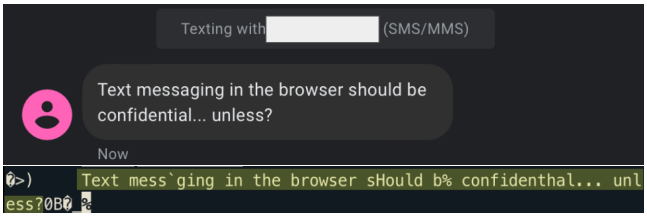

Figure 12: (Top) Text message sent to an Android phone which has been paired to the Google Messages webpage. (Bottom) Recovered text message in highlights.

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.

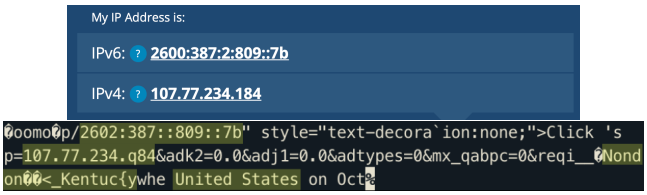

Figure 13: (Top) Website displaying the target's IPv4 and IPv6 addresses. (Bottom) Recovered information in highlights, including geolocation.

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 4.101.1 (the latest at the time of writing) on our Safari browser. Next, while LastPass will require user interaction when autofilling credentials for the first time on a webpage, it will automatically and permanently fill them in on subsequent logins without any interaction. Thus, by opening the login pages of popular websites (prompting LastPass to autofill credentials), we are able to recover the target's username and password. See Figure 14.

Forced Logout to Induce Login Page. In order to steal

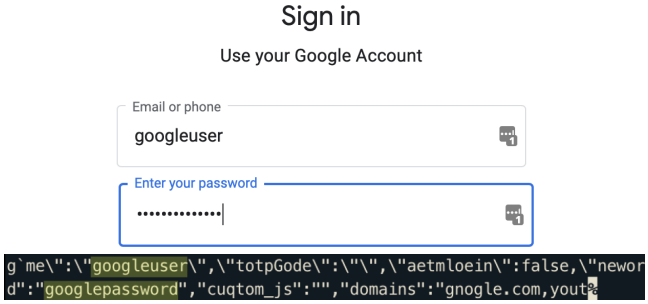

Figure 14: (Top) Google's accounts page autofilled by LastPass, where the password is `googlepassword`. (Bottom) Leaked page data with credentials highlighted.

credentials, an attacker must first prompt a credential manager to autofill the target's password. Thus, in case the target is already authenticated in to a service, we must somehow log the target out to be able 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 Countermeasures and Limitations

Eliminate Consolidation. Recall WebKit's process model (Section 5.1), where `iframe` elements and pages launched with `window.open` are consolidated into the address space of their parent's rendering process. While this is a sensible decision in terms of performance, since the rendered pages must have a reference back to their parent page in such cases, this design was what made our attack possible. Therefore, Apple might consider revising Safari's site isolation strategy, spawning new processes not only for new tabs, but also for `iframes` and pages launched with `window.open`. While this will require adding an inter-process communication API to Safari, and will thus incur overhead, it will mitigate our attacks.

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, therefore preventing the CPU from speculating past it.

Remove 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 some corner cases. 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.

Leak Rate. As shown in Table 2, our attack recovers data at rate of about 30 bits per second. While the bottleneck of our attack is the transmission rate of pLRU covert channel in the presence of Safari's 1 ms timer, we do acknowledge the relatively low leakage rate of our attack. Thus, we leave the task of 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 [5], 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.

8 On the Adoption of Cross-origin Isolation

Recall from Section 4.2 and Section 5.1 that cross-origin isolation was proposed as a Spectre countermeasure for websites, and cross-origin isolated pages failed to consolidate in Safari. Here, we discuss the cross-origin isolation specification in further detail. A webpage is cross-origin isolated if it is served with two HTTP response headers: `Cross-Origin-Opener-Policy` (COOP) must be set to `same-origin`, and `Cross-Origin-Embedder-Policy` (COEP) to `require-corp` [48]. COOP states that the webpage must not share a browsing context group with other pages, i.e., be able to communicate with other pages. Meanwhile, COEP requires third-party resources to opt into being loaded as part of the current webpage [48].

Experimental Setup. We note that COOP is designed to prevent cross-page communication, and does not explicitly mandate anything about the browser's process model or consolidation. However, we observed the presence of the COOP header alone precludes consolidation. We tested this both empirically and by inspecting WebKit's commit log, where Safari spawns a new rendering process if the page has the COOP header set to the correct value [20].

Accordingly, we measure how widely popular websites have adopted cross-origin isolation 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 for the presence of the COOP header and cross-origin isolation across four user agents: Safari 16.0 for macOS and iOS, and Chrome 105 for macOS and iOS. Overall, the number of URLs crawled was 3,058 for both browsers on macOS and 2,551 for both browsers on iOS (since some hyperlinks are not included in mobile versions of webpages). In total, our crawler visited 5,609 URLs.

Results and Suggestions. Out of 5,609 visited URLs, we find that only 113 (2%) serve the COOP header with the correct value. Matching these 113 URLs to their corresponding Alexa top 100 websites, we find that 13 out of the Alexa top 100 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 most websites, with some servers serving COOP headers to desktop users but not mobile, and vice versa. Finally, due to limitations of our web crawler, we did not visit webpages that require authentication.

Adopting Isolation. We note that adopting cross-origin isolation headers differs from our other countermeasures mentioned in Section 7 above. This is since website administrators can locally enable these headers, independently of any Safari modifications performed by Apple. Therefore, for webpages which do not need the capability to communicate across windows, we strongly recommend the widespread adoption of cross-origin isolation headers.

Shortcomings of Cross-origin Isolation. We acknowledge however that cross-origin isolation is not a panacea. The current specification prohibits a window from communicating with any other window or resource. This is problematic since many webpages offload user-specific functionality to other third-party webpages, such as external payment processing and authentication. Prohibiting cross window communication will thus make webpages unable to share user identifiers with external services, breaking the ability of websites to rely on external functionalities. Recognizing the issue, the W3C suggests extending the definition of cross-origin isolation in the HTML standard to allow the current webpage to open popups [47]. At the time of writing however, such an extension has not yet been codified.

Next, cross-origin isolation can introduce problems even when credentials do not need to be communicated. Recall that the COEP header will preclude the current webpage from loading cross-origin resources unless the resource explicitly opts in. Practically, this prohibits news and social media websites from loading images from a content delivery network. To alleviate this, the HTML standard is also being updated with a 'credentialless' mode [46] that allows cross-origin requests which are not login-protected. While only Google Chrome and Microsoft Edge support this update at the time of writing [18], we anticipate other major browsers to implement it in the near future.

References

[1] Onur Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *CSAW*, 2007.

[2] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking chrome strict site isolation via speculative execution. In *IEEE SP*, 2022.

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

[4] Apple. Webkit2. https://trac.webkit.org/wiki/WebKit2, 2011.

[5] Apple. About speculative execution vulnerabilities in arm-based and intel cpus. https://support.apple.com/en-us/HT208394, 2018.

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

[7] Arm Ltd. Architecture reference manual: Armv8 for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/ga, 2017.

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

[9] Eloi Benoist-Vanderbeken and Fabien Perigaud. WEN ETA JB? a 2 million dollars problem, 2019.

[10] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, 2019.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

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

[13] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, 2019.

[14] MDN Contributors. Window.opener - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Window/opener, 2022.

[15] MDN Contributors. Window.parent - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Window/parent, 2022.

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

[17] Alexis Deveria. Shared array buffer | can i use... support tables for html5, css3, etc.

[18] Alexis Deveria. headers http header: Cross-origin-embedder-policy: credentialless | can i use... support tables for html5, css3, etc. https://caniuse.com/mdn-http_headers_cross-origin-embedder-policy_credentialless, 2022.

[19] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime + Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *USENIX Security*, 2017.

[20] Chris Dumez. Move cross-origin-opener-policy handling to the networkprocess. https://github.com/WebKit/WebKit/commit/a199c9c906f0310fab825f4b0f8d1bf4894a0665, 2021.

[21] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, 2016.

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

[23] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the GPU. In *IEEE SP*, 2018.

[24] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ACNS*, 2018.

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

[26] Google. Spectre. https://leaky.page, 2021.

[27] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[28] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[29] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, 2015.

[30] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *DIMVA*, 2016.

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

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

[33] Berk Gülmezoglu, Andreas Zankl, M. Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining user privacy on mobile devices using AI. In *AsiaCCS*, 2019.

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

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

[36] Lorenz Hetterich and Michael Schwarz. Branch different-spectre attacks on apple silicon. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 116–135. Springer, 2022.

[37] Jann Horn. Speculative execution, variant 4: Speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[38] Lin-Shung Huang, Alex Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *USENIX Security*, 2012.

[39] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE SP*, 2013.

[40] IEEE Standard Board. IEEE standard for floating-point arithmetic. IEEE Std 754-2008, IEEE, 2008.

[41] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *IEEE SP*, 2015.

[42] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative load hazards boost Rowhammer and cache attacks. In *USENIX Security*, 2019.

[43] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *DAC*, 2016.

[44] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[45] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX Security*, 2021.

[46] Eiji Kitamura. Load cross-origin resources without corp headers using ‘coep: credentialless‘.

[47] Eiji Kitamura. Making your website "cross-origin isolated" using coop and coep. https://web.dev/coop-coep/, 2022.

[48] Eiji Kitamura and Domenic Denicola. Why you need "cross-origin isolated" for powerful features. https://web.dev/why-coop-coep/, 2021.

[49] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, 2019.

[50] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security*, 2016.

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

[52] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT*, 2018.

[53] Daniel Lemire. 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/, 2021.

[54] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[55] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of AMD's cache way predictors. In *Asia CCS*, 2020.

[56] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, 2015.

[57] Andrei Lutas and Dan Lutas. Security implications of speculatively executing segmentation related instructions on Intel CPUs. https://businessresources.bitdefender.com/hubfs/noindex/Bitdefender-WhitePaper-INTEL-CPUs.pdf, Aug 2019.

[58] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, 2018.

[59] Hector Martin. M1ssing register access controls leak EL0 state. https://m1racles.com/, 2021.

[60] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. CoRR abs/1902.05178, 2019.

[61] MDN Contributors. performance.now() - web APIs - MDN. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now#reduced_time_precision, 2022.

[62] Mozilla. Project Fission. https://wiki.mozilla.org/Project_Fission, 2021.

[63] Nick Nguyen. The best Firefox ever. https://blog.mozilla.org/blog/2017/06/13/faster-better-firefox/, 2017.

[64] Ryosuke Niwa. Reduce the precision of "high" resolution time to 1ms. https://github.com/WebKit/WebKit/commit/25e575313d12e97a9e6c2b1d9b6ddd1d510e01a9, 2018.

[65] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, 2015.

[66] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, 2006.

[67] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[68] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security*, 2016.

[69] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *CCS*, 2021.

[70] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, 2021.

[71] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE SP*, 2021.

[72] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. PACMAN: Attacking Arm pointer authentication with speculative execution. In *ISCA*, 2022.

[73] Charles Reis, Adam Barth, and Carlos Pizano. Browser security: Lessons from Google Chrome: Google Chrome developers focused on three key problems to shield the browser from attacks. *Queue*, 7(5): 3–8, 2009.

[74] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: process separation for web sites within the browser. In *USENIX Security*, 2019.

[75] Justin Ludwig Rezos. Cross frame scripting. https://owasp.org/www-community/attacks/Cross_Frame_Scripting, 2021.

[76] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[77] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *Financial Cryptography and Data Security*, 2017.

[78] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security*, 2019.

[79] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime + Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *USENIX Security*, 2021.

[80] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[81] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[82] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE SP*, 2020.

[83] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why stopping cache attacks in software is harder than you think. In *USENIX Security*, 2018.

[84] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Rogue in-flight data load. In *IEEE SP*, 2019.

[85] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxe.com/files/SGAxe.pdf, 2020.

[86] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *USENIX Security*, 2021.

[87] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. In *IEEE SP*, 2022.

[88] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *IEEE SP*, 2019.

[89] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, 2017.

[90] Yoav Weiss and Eiji Kitamura. Aligning timers with cross origin isolation restrictions. https://developer.chrome.com/blog/cross-origin-isolated-hr-timers/, 2021.

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

[92] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *USENIX Security*, 2022.

[93] Johannes Wikner, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Spring: Spectre returning in the browser with speculative load queuing and deep stacks. In *WOOT*, 2022.

[94] Henry Wong. Intel Ivy Bridge: Cache replacement policy. https://blog.stuffedcow.net/2013/01/ivb-cache-replacement/, 2013.

[95] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE SP*, 2016.

[96] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[97] Andy Zeigler. 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, 2008.

A Multi-Process Web Browsers

The ever-increasing use of the web browser has resulted in users often keeping many websites open simultaneously, with each website housed in its own dedicated browser tab. While early browser designs used a uni-process model, with one process responsible for rendering all the open windows and tabs, modern browsers adopted a multi-process architecture with multiple unprivileged rendering processes rendering open websites [4, 63, 73, 97]. These effectively compartmentalize the browser, limiting the damage done in case of an exploit inside the browser's rendering engine.

Site Isolation. The next step in the evolution of browser security architecture aims to improve the grouping of pages into rendering processes. Instead of grouping pages arbitrarily, site isolation [74] uses the URL serving the page to group them, aiming to separate domains from each other. Spectre's discovery [49] further accelerated the deployment of site isolation, in an attempt to create address space separation between data from mutually-distrusting domains.

eTLD+1 Consolidation. Both Chrome's and Firefox's implementation of strict site isolation groups websites intro rendering processes based on their effective top-level domain plus one sub-domain (eTLD+1). More specifically, any given rendering process will only handle websites that share their eTLD+1, while pages served from different eTLD+1's will be rendered by different processes. 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

With Apple going as far as eliminating 32-bit support from MacOS Catalina (10.15), all modern versions of Safari are 64-bit applications running on 64-bit hardware. Despite this, Safari uses 32-bits to represent array indices, and 35-bits to represent a compressed pointer to the Butterfly, which is the underlying storage holding the array data. The underlying storage for nearly all JavaScript objects (including arrays) is allocated in an isolated 32 GB region of the rendering process's address space named the Gigacage. To index into an array, Safari views indices as unsigned 32-bit offsets which are added to the array's Butterfly pointer. The outcome is then further added to a 64-bit immediate holding the address of the beginning of the Gigacage. Finally, data is fetched by dereferencing the resulting pointer, returning the array element to the JavaScript application. See Figure 15.

The idea is that even in case of 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 [9]. Indeed, inspecting Safari's memory layout using a debugger, we find that secrets from the DOM of the target webpage do not reside in the Gigacage, effectively putting them out of reach for 35-bit attackers.

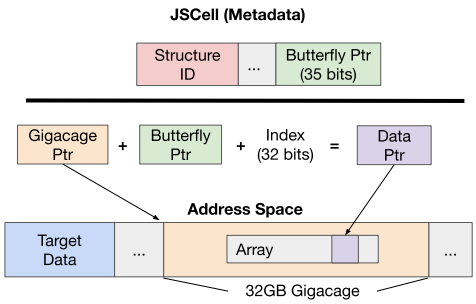

Figure 15: Layout of the `JSCell` class holding the metadata for JavaScript objects, and address computation for array indexing in Safari. The resulting address is confined to the Gigacage.

C Pseudo Least Recently Used

Since the available timer sources in macOS, iOS and Safari are all low-resolution timers or require privileged access, we cannot use these to measure the execution time of a single memory access, as is typically required when performing cache attacks. While previous works have resorted to timing techniques based on the clock edge [27, 50, 77] to overcome this challenge, we instead rely on an amplification technique observed by Google [26] that abuses the pLRU eviction strategy of L1-D caches.

To better understand how the pLRU algorithm works, we first consider a binary tree as shown in Figure 16. Each node in the tree has a one-bit flag where 0 and 1 denote that we

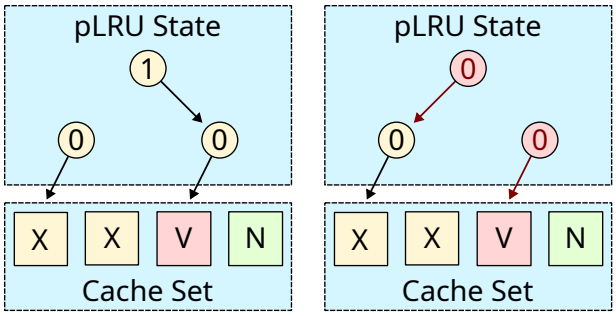

Figure 16: On the bottom we have a 4-way cache set with V being the victim element, N being the direct neighbor of the victim and X being unrelated elements. On the top we have the pLRU state that points to the element that should be evicted next. On the left the pLRU state points to V, whereas on the right the pLRU tree points to N.

should go left and right respectively to find the next node to evict. For instance, in the left tree, if we follow the nodes, we will end up at V being the eviction candidate.

In addition, whenever we access an element that is present in the cache set, we have to update the pLRU tree to point to the approximately least-recently used element. This is achieved by updating the nodes along the path to that element by setting the bit flag of each node opposite of the direction that was taken. If we access element N, for instance, the root node that used to point to the right will now be updated to point to the left, while the next node will still point to V as we accessed N. This is also illustrated by the transition of the figure on the left to the right with the updated nodes along the path to N marked in red.

Abusing the pLRU Algorithm. Note that whenever the pLRU state points to the victim element, we can divert the arrow to point away from the victim instead simply by accessing the neighboring victim. In fact, by repeatedly accessing the two elements X and the neighbor we can keep the victim element in the cache set for an indefinite amount of time. More specifically, using this knowledge we can perform a PRIME+PROBE attack: First, we reset and prime the cache set. Then, we run the victim code. Finally, we probe the cache set while keeping the victim alive.

Resetting/Massaging the pLRU State. After successive runs of the attack, the cache set may still contain any elements we used during the attack. Thus, the attacker first has to reset the pLRU state of the cache set by filling it with as many unrelated elements as there are ways in the cache set. Then we fill the cache set with our own elements except for one spot that we leave empty for the victim. In addition, we keep track of the element that we accessed after accessing half the elements of our set, as this particular element will land at the neighboring spot of the victim.

Keeping the Target Alive. After priming the cache set of choice, we can run the victim code that we want to monitor. If the victim ends up using the same cache set, it loads the victim element where we want it while evicting one of our elements. Otherwise, if the victim ends up using a different cache set, the elements in the cache set remains untouched.

To keep the victim element alive in the cache set, we can now access our elements while accessing our neighboring element every so often as the pLRU state is about to point to the victim. More specifically, we have to access our neighboring element after at most $\log_2(\text{ways})$ elements. At the same time, we measure the execution time of our accesses in our probe step. If the victim element ended up in the same cache set, our elements will no longer fit simultaneously, and as the victim element never gets evicted, each element will repeatedly evict another element in our set. Thus, all accesses to our own elements will be slow. On the other hand, if the victim element ends up in a different cache set, all our accesses will be fast, as our elements perfectly fit within the cache set.

Amplifying the Signal. One property of this attack primitive is that we can decide how often we want to access our elements in sequence to amplify the signal. By simply increasing the number of iterations, we increase the number of accesses to the cache set, which makes the timing difference more apparent. Given that the number of iterations is sufficiently large, we can use this to monitor L1-D cache sets with lower resolution timers such as `mach_absolute_time()` and `performance.now()`.

Effectiveness of pLRU. To measure the effectiveness of the pLRU attack primitive, we implement the pLRU primitive using different timing APIs as discussed before. We then measure the time it takes to execute the pLRU primitive when the target is in the same and in a different cache set to find out when we can actually distinguish these cases. The number of pLRU rounds indicates how often we iterate over the fill set. As the results in Figure 17 show, the KPC and `mach_absolute_time()` APIs are able to distinguish these cases starting from 40 pLRU rounds, whereas `performance.now()` is able to distinguish them starting from 40,000 pLRU rounds.

pLRU as a Covert Channel. Finally, another application of the pLRU primitive is that it can be used as a covert channel to signal a bit of information. In fact, as the state of the cache is not reverted when the CPU returns from speculative execution back to normal execution, we can even use this covert channel in a transient execution attack, where we propagate some information in a speculative gadget by accessing the same or a different cache set to signal a 1 or 0 respectively. This makes the pLRU technique especially interesting in environments that only offer low-resolution timers, such as web browsers.

D L2 Inclusiveness

With the ability to efficiently and reliably find minimal L2 eviction sets, we want to make sure that the L2 cache is actually *inclusive*. More specifically, a L2 cache is *inclusive* of the L1 caches, if any cache line present in the L1 is also guaranteed to be present in the L2. Otherwise, a L2 cache

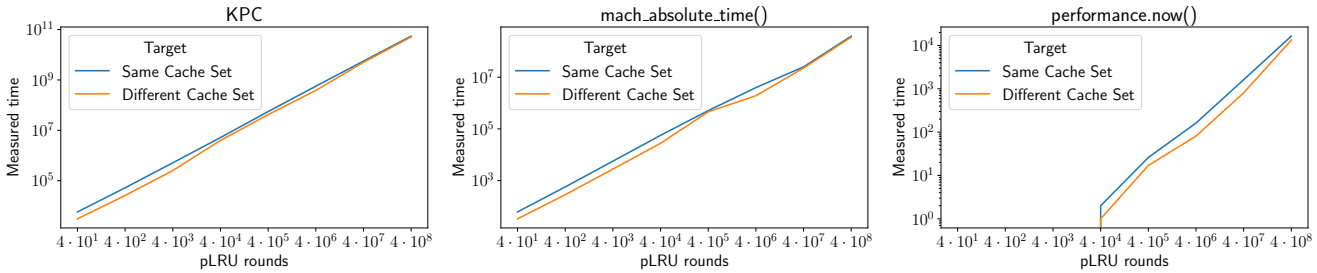

Figure 17: The time it takes to execute pLRU attack primitive when the target is in the same and a different L1-D cache set using the KPC, `mach_absolute_time()` and `performance.now()` timers. The pLRU rounds (x-axis) indicate how often we iterate over the fill set.

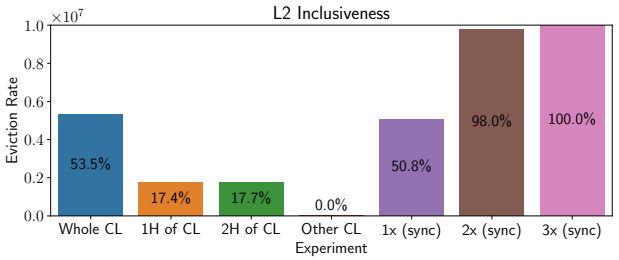

Figure 18: Eviction rate when evicting a L2 cache line from different cores on the M1 CPU. The first three experiments show the effects of 64B vs. 128B accesses, while the fourth experiment shows that accessing a different L2 cache set has no effect. The last three experiments show how effectiveness improves the more often the eviction set is being accessed.

is *non-inclusive* or *exclusive* depending on whether the L2 cache may contain cache lines of the L1 caches or not, respectively. Hence if the L2 cache is inclusive of the L1 caches, evicting an entry from the L2 cache will also evict it from all the L1 caches. We can use this property to test if a L2 cache is inclusive of the L1 caches by trying to evict a L2 cache line used by one CPU core from another core, as doing so requires only a minimal L2 eviction set.

64B vs. 128B Cachelines. Before testing inclusiveness, we begin by noting the discrepancy in line sizes between the L1 and the L2 caches, which may affect the policy. More specifically, as the L1 cache line size is 64B while the L2 cache line size is 128B, it is possible that the L2 cache is only truly inclusive when we access both 64B halves of each L2 cache line. As such, our first three experiments focus on testing the behavior of accessing the L2 cache lines at a 64B granularity. That is, they access both the first half and the second half of each L2 cache line, respectively, during the eviction test. To further confirm that we are indeed evicting the victim's L2 cache line, we perform a fourth experiment that accesses a different unrelated L2 set instead, as that should not evict the victim's L2 cache line. Figure 18 shows that, while accessing only half of the L2 cache line does evict the victim's L2 cache line, it is actually much more effective to access both halves. We conjecture that the L2 cache deploys an eviction strategy that prioritizes the eviction of cache lines that do not fully utilize the 128B available.

Synchronization. To ascertain the L2 cache is indeed inclusive, we want to study the effects of accessing the eviction set a few times before testing if the victim's cache line has been evicted. To implement this, we resort to atomic instructions to implement a spinlock such that both threads can run in turns, while keeping the CPU fully occupied while they wait for the other thread to complete their turn.

We check the CPU utilization through the `powermetrics` command, and confirm that both threads are indeed running simultaneously. During the attacker's turn, the attacker shuffles and accesses the eviction set a number of times before yielding to the victim. As shown in Figure 18, accessing the eviction set more often results in a higher eviction rate, with it being close to 100% when accessing the eviction set three times more than the baseline. This strongly indicates that the L2 cache is inclusive of the L1 caches on the Apple M1.

E Safari's NaN-boxing

Integer	0b	1111111111111110...xxxxxxxxxxxxxxxx
Float	0b	xxxxxxxxxxxxx1x...xxxxxxxxxxxxxxxx
True	0b	0000000000000000...0000000000000111
False	0b	0000000000000000...0000000000000110
Undefined	0b	0000000000000000...0000000000001010
Null	0b	0000000000000000...0000000000000010
	Bit	63 48.....15 0

Figure 19: NaN-boxing in WebKit `JSValues` to differentiate data types. The bit patterns that indicate each data type are colored in red.