SoK: In Search of Lost Time: A Review of JavaScript Timers in Browsers

Thomas Rokicki, Clémentine Maurice, Pierre Laperdrix

SoK: In Search of Lost Time: A Review of JavaScript Timers in Browsers

Thomas Rokicki
Univ Rennes, CNRS, IRISA
Rennes, France

Clémentine Maurice
Univ Lille, CNRS, Inria
Lille, France

Pierre Laperdrix
Univ Lille, CNRS, Inria
Lille, France

***Abstract*—JavaScript-based timing attacks have been greatly explored over the last few years. They rely on subtle timing differences to infer information that should not be available inside of the JavaScript sandbox. In reaction to these attacks, the W3C and browser vendors have implemented several countermeasures, with an important focus on JavaScript timers. However, as these attacks multiplied in the last years, so did the countermeasures, in a cat-and-mouse game fashion.**

In this paper, we present the evolution and current situation of timing attacks in browsers, as well as statistical tools to characterize available timers. Our goal is to present a clear view of the attack surface and understand: what are the main prerequisites and classes of browser-based timing attacks and what are the main countermeasures. We focus on determining to what extent the changes on timing-based countermeasures impact browser security. In particular, we show that the shift in protecting against transient execution attacks has re-enabled other attacks such as microarchitectural side-channel attacks with a higher bandwidth than what was possible just two years ago.

1. Introduction

Subtle variations of computation time can reveal information about the state of a system. Research has developed a variety of side and covert channels, allowing potential attackers to extract secrets or track user behavior. Timing attacks can aim at different components of the microarchitecture, e.g., cache, DRAM, and are purely software-based. These attacks have two common prerequisites: they run code on the victim's hardware, and they rely on high-resolution timers that can distinguish small timing variations in the order of $100\,\mathrm{ns}$. Most of the timing attacks are implemented in native code, allowing the attacker to have great control over the memory and cycle-accurate timers.

In contrast, JavaScript is a high-level object-oriented interpreted scripting language, following the ECMAscript standard [30]. Contrary to native code, it is much easier to run JavaScript code on a victim's system as it is a major component of the web. Almost all websites use JavaScript to execute code on the client side and by visiting a page, a client can download and execute dozens of different scripts. For security purposes, JavaScript code runs inside a sandboxed environment, restricting access to local files, virtual or physical memory addresses and native instructions. These restrictions make it harder to implement microarchitectural attacks. However, Oren et al. [45] implemented a fully JavaScript-based cache attack, running entirely in the browser. Based on Prime+Probe [42], it allows an attacker to track user behavior and recover information belonging to other processes running on the same system.

To try and mitigate JavaScript-based timing attacks, browser vendors have developed countermeasures specifically targeting timers. Notably, they decreased timers' resolution to make them less precise and introduced jitter to add noise in measurements. Other security features like site isolation [48] were added to reinforce the security of the browser and act as a novel line of defense against timing attacks. Amid all these changes, it can be hard to keep track of all the different evolutions that browsers underwent. Particularly, it is unclear how the attacks described in the literature are impacted by current countermeasures.

In this SoK, we aim to provide a clear view of the vulnerability of browsers to timing attacks. In the first part, we take a broad look at the research done in the area to systematize it. We provide a taxonomy of attacks with their prerequisites and classify the countermeasures based on the resources they target. In the second part, our goal is to assess the true efficiency of timing-based countermeasures after seeing how much they changed over the years. In order to gain proper insight, we implemented our own `performance.rdtsc()` high-precision timer into Chromium and Firefox so that we can deconstruct studied timers into their most basic blocks. With its help, we identified that recent countermeasures present great advances against timing attacks but they also present noticeable steps back. For example, with the introduction of COOP/COEP, Firefox 79 gained a robust resource isolation mechanism but lost a lot with regards to timing attacks. Before, an attacker needed several minutes to build an eviction set to conduct an attack. Now, with the resolution changed from $1\,\mathrm{ms}$ to $20\,\mu\mathrm{s}$, we show that it can be setup in just a single second. Another problem is the recent reintroduction of `SharedArrayBuffer` after it was deactivated due to the disclosure of Spectre [38]. Its presence introduces a real security risk because a malicious script can abuse it by creating a very powerful timer that has an incredibly high resolution with very low overhead. By lowering timing-based countermeasures, all the prerequisites for large classes of timing attacks are met, meaning that these attacks can theoretically be implemented.

Contributions. This paper makes the following contributions:

- We provide a classification of prerequisites for timing attacks and present the most notable classes of timing attacks (Section 2).
- We classify countermeasures based on the resources they target (Section 3).
- We present tools to analyze timers and the threat they pose for timing attacks. Notably, we detail our custom timer and automatic tests that measure the resolution and measurement overhead of several built-in timers, which can easily be reproduced for other systems and future browser versions (Section 4).
- We present a longitudinal study of browsers' timing-based countermeasures. (Section 5).

2. Timing attacks in browsers

Timing attacks in browsers are a large class of attacks exploiting timing differences in computations in order to infer private information. As they are mainly based on JavaScript, the attacker code will, by design, always be executed on the victim's hardware. In this section, we aim at systematizing timing attacks in browsers, by classifying prerequisites for different classes of attacks.

2.1. Attack prerequisites

We systematize the major timing attacks prerequisites in order to classify them and better understand the outline of attacks. We have identified the following prerequisites:

P1: High-resolution timers,
P2: Shared hardware resources,
P3: Transient execution,
P4: Shared system resources,
P5: Shared browser features.

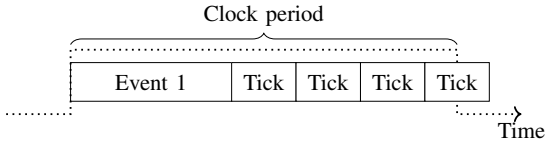
2.1.1. P1: High-resolution timers

Definition. In this work, we call a timer, or a clock, a tool that allows to differentiate two events based on their respective timings. To do so, a timer relies on an operation that needs to:

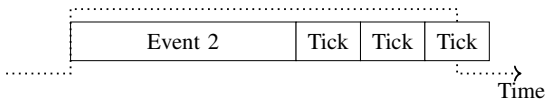
- Be constant over time so that it provides a reliable non-varying unit of time.
- Be free running so that it allows the computation of time differences without blocking the program execution.

We call clock edge the moment where a constant time operation ends and the following starts. A timer can differentiate two events if each of them crosses a different number of clock edges. The duration of the constant time operation is the smallest amount of time this timer can measure, *i.e.*, its resolution.

Timer interpolation. By definition, a timer cannot measure an event shorter than its resolution. Yet, it is possible to bypass this limitation by using *interpolation* to retrieve an even finer-grained timer. The idea behind it is straight forward: one counts the number of times a shorter non-free running operation can be executed. We refer to such an operation as a tick. This tick can simply be writing repeatedly some data to memory or increasing a custom counter by one.



(a) The interpolated time is 4 ticks.



(b) The interpolated time is 3 ticks.

Figure 1: Clock interpolation: Counting the number of ticks between the end of the event and end of the clock period. Even if both events are shorter than the clock period, we can distinguish them by their interpolated time: event 1 has an interpolated time of 4 ticks where event 2 has only 3 ticks.

Figure 1 provides an example of how timer interpolation works. Events 1 and 2 have a shorter execution time than the clock period *i.e.*, the resolution. Interpolation is then needed to differentiate them based on their timing. By running events at the beginning of a clock period and counting ticks when they are finished, we can conclude how fast each of them is when the next clock edge is reached. The more ticks are counted, the faster the event is. In Figure 1, Event 1 is faster than Event 2 as it has 4 ticks against 3. It should be noted that the interpolated timer is equivalent to a timer with a resolution equals to the duration of a tick.

`performance.now()`. The JavaScript High Resolution Time API [32] offers access to the `performance.now()` method, which returns a high-resolution timestamp. The timestamp value represents the time elapsed since the beginning of the current document lifetime in ms, originally with a resolution in the order of 1 ns.

For security reasons, `performance.now()`'s resolution has evolved over time. However, in 2017, Schwarz et al. [53] demonstrated that it is still possible to recover high-resolution timers regardless of the base resolution, by using a variable increment as a tick.

`SharedArrayBuffer`-based clock. Initially a single-threaded language, JavaScript has evolved into a multi-threaded paradigm. ECMA2017 introduced the `SharedArrayBuffer` API [29] in order to accelerate computations between threads. It allows creating an array shared between the main thread and a sub-thread, or web worker. First implemented in Firefox 46 and Chrome 60, they were used by Schwarz et al. [53] to build a high-resolution timer.

Figure 2 illustrates a simple `SharedArrayBuffer`-based clock. The attacker creates a *clock* web worker, and shares a `SharedArrayBuffer` between the threads. The clock thread is an infinite loop, perpetually incrementing a value in the `SharedArrayBuffer`. This value can be consulted at any time by the attacker, and represents a timestamp. The resolution of this timer is very high as it is

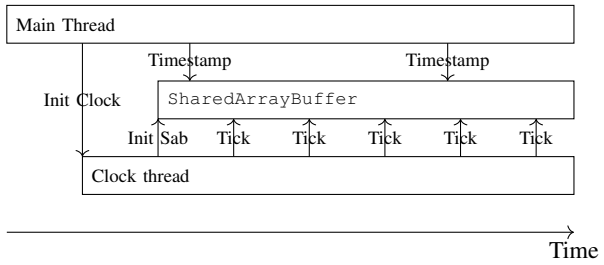


Figure 2: Simple `SharedArrayBuffer` based clock: To time an event, the main thread evaluates the shared value before and after its event.

of the order of the computation time of a shared operation, such as incrementing or reading a shared variable.

Other JavaScript timers. The W3C standards describe other API timers accessible in JavaScript, such as `Date.now()` [23], `Window.requestAnimationFrame()` [26] or `Window.setTimeout()` [27], but they offer a resolution lower than the one of `performance.now()`. Schwarz et al. [53] also designed other auxiliary timers. They presented clocks based on CSS animations. An attacker can run code JavaScript code at each screen refresh, *i.e.*, every 16.66 ms for a typical refresh rate of 60 Hz, thus this can be used for interpolation as well. This is equivalent to using `Window.requestAnimationFrame()` directly in JavaScript.

Most of these timers are less efficient than the one previously developed and are thus not in the scope of this paper.

2.1.2. P2: Shared hardware resources

Modern CPUs present major microarchitectural optimizations in order to compute rapidly and efficiently, often solely created with performance in mind. Since hardware is situated below software security rings and components are often shared between processes, contention at the microarchitectural level can be the root cause of software-based side-channel attacks. By design, these attacks are also less fixable than software.

2.1.3. P3: Transient execution

Modern CPUs use out-of-order execution and speculative execution in order to optimize computations and make the best use of resources.

Out-of-Order execution. Instead of executing instructions in a linear order, CPUs execute them out-of-order, depending on input and execution units availability. The re-order buffer (ROB) then commits the instructions in-order to the architectural states. In case of interruptions occurring during Out-of-Order, the CPU flushes the instructions out of the ROB. Such computed but not committed micro-instructions are called *transient instructions*.

Speculative execution. During code execution, conditional branches and data dependencies would force the CPU to wait until a certain dependency or branch is resolved before continuing. Such stalling decreases the maximum performance a CPU can achieve. To address this, CPUs deploy strategies to speculate on the outcome of a condition and continue executing the predicted path, saving the results in the re-order buffer. After the resolution of the branch or dependency, if the prediction was correct, the pre-computed instructions are committed from the ROB. However, if the prediction was incorrect, the CPU deletes all the transient instructions from the buffer and roll-back to the last correct state.

Table 1: Comparison of attacks prerequisites. All attacks require P1, but with different ranges of resolutions.

	Required resolution (P1)	P2	P3	P4	P5
Spy in the sandbox [45]	100 ns	●	○	○	○
Rowhammer.js [34]	100 ns	●	○	○	○
Website fingerprinting [54]	10 − 100 ms	●	○	○	○
DRAM covert channel [53]	10 ns	●	○	○	○
Breaking ASLR [31]	100 ns	●	○	○	○
Spectre [38]	100 ns	●	●	●	○
ret2spec [43][1]	100 ns	●	●	●	○
RIDL [61]	100 ns	●	●	●	○
Store-to-Leak [15]	100 ns	●	●	●	○
Memory deduplication [33]	1 µs	○	○	●	○
Loophole [62][2]	100 µs	○	○	○	●
Extension side channel [59]	100 ns	○	○	○	●

2.1.4. P4: Shared system resources

In a browser, system resources are often shared between contexts. Memory in particular can be shared between all code running in the same browser process.

2.1.5. P5: Shared browser features

A lot of purely software features are shared between contexts or tabs in browsers. This is the case for history, cookies, event loops or renderers for example. The difference in timings of such features' operation can leak information between tabs and inform about a user's behavior.

2.2. Attack classes

We now give an overview of the major classes of timing attacks in browsers, along with their major characteristics and prerequisites. Table 1 illustrates the prerequisites for a sample of state-of-the-art timing attacks in browsers.

2.2.1. Hardware-based side-channel attacks

Hardware-based side-channel attacks exploit hardware components shared by all processes in the system. By measuring the computation time of specific operations, an attacker can infer the state of certain components. These attacks share two major prerequisites:

P1: High-resolution timers,
P2: Shared hardware components, e.g., cache, DRAM.

The resolution of required timers vary in function of the attacked hardware, but typically spans from $10-100$ ns in order to potentially distinguish different states in the hardware.

Cache-based attacks. A cache is a small memory that sits close to the CPU cores. Modern CPUs employ multiple levels of caches, the smallest being the fastest, and the largest being the slowest. Cache attacks exploit the difference of

1. ret2spec can still break ASLR with site isolation.
2. Since each process has its own event loop, the attack is not implementable on recent Chrome versions.

timing between a cache hit (*i.e.*, memory is served from the cache) and a cache miss (*i.e.*, memory is served from the DRAM). Typically, such attacks require timers with a resolution in the order of 100 ns. By monitoring a specific cache section, an attacker can retrieve information about the actions of other processes. Two main variants exist. First, Flush+Reload [35], [67] can monitor a single cache line, but requires shared memory between an attacker and the victim. Second, Prime+Probe [42], [47] can monitor a cache set, by continuously accessing an eviction set, *i.e.*, a set of addresses sharing this same cache set, and measuring the time the Probe step takes. An attacker learns whether another process has loaded a line in the same cache set between two Probe steps. This attack does not require any shared memory, nor any specific instruction to evict cache lines from a cache set. It can therefore be implemented in JavaScript.

Oren et al. [45] were the first to implement Prime+Probe in JavaScript, therefore extending its attack model drastically. They demonstrate a covert channel and side-channel attacks to track user behavior. Gruss et al. [34] used Prime+Probe in order to evict specific cache sets and allow the implementation of the DRAM fault attack Rowhammer [37] in JavaScript. Shusterman et al. [54] monitored the whole last-level cache in order to fingerprint websites opened in other browser tabs, without requiring high-resolution timers.

Other hardware side-channel attacks. Gras et al. [31] demonstrated that it is possible to de-randomize virtual addresses from ASLR from the JavaScript sandboxed environment. Andrysco et al. [9] exploited floating point computation timing differences to perform history sniffing. In a paper studying auxiliary time sources, Schwarz et al. [53] introduced a DRAM covert channel that is purely based on JavaScript. Sanchez-Rola et al. [50] showed a method relying on computer internal clock imperfections to fingerprint unique machines. Schwarz et al. [51] presented JavaScript template attacks to create fingerprints and retrieved the instruction-set architecture and the used memory allocator.

2.2.2. Transient execution attacks

With Spectre [38], Kocher et al. paved the way to a new class of attacks, exploiting transient execution to leak protected data. Some of these attacks can be implemented in browsers in JavaScript, including Spectre-PHT [16]. This class of attacks is very wide, but shares some prerequisites:

P1+P2: A hardware covert channel to extract leaked data,
P3: Transient execution,
P4: Shared system resources—finding secrets to leak.

State-of-the-art attacks. P3 is a wide prerequisite: the Spectre attack alone possesses many variants, each targeting different optimizations [16]. Most crucially, different attacks target different secrets to leak (P4), *i.e.*, in the same address space, or across address spaces. Maisuradze et al. [43] demonstrated how a JavaScript attacker could read outside of the sandboxed environment by exploiting return stack buffers misspeculations. Van Schaik et al. introduced RIDL [61], a class of transient attacks able to leak *in-flight* data with unprivileged code, including JavaScript. Canella et al. [15] introduced Store-to-Leak, and break KASLR and the ASLR of the browser in JavaScript.

Most of these attacks need a covert channel to extract the leaked data (P1+P2). To this purpose, they mainly use Prime+Probe, hence require the same timer resolution as cache-based attacks, around 100 ns.

2.2.3. Attacks based on system resources

By exploiting shared system resources, an attacker can also retrieve information using a side-channel attack. These attacks share the following prerequisites:

P1: High-resolution timers,
P4: Shared system resources.

State-of-the-art attacks. Gruss et al. [33] exploited memory deduplication to let an attacker leak data from the sandboxed environment. Lipp et al. [41] managed to time keystrokes from JavaScript by detecting the number of interruptions during a fixed time frame.

2.2.4. Attacks based on browser resources

In order to have a uniform and efficient browsing experience, most browsers share information between tabs or processes. This includes, among others, browsing history, browser extensions, or event loops. However, this sharing of information, even if not directly reachable in JavaScript, can leak private information to a malicious site. This class of attacks shares two prerequisites:

P1: High-resolution timers,
P5: Shared browser resources.

State-of-the-art attacks. Mowery et al. [44] implemented JavaScript-based fingerprinting in 2011, by exploiting engine characteristics, as well as showing the inefficiency (if not threat) of privacy extensions such as NoScript [1]. Van Goethem et al. [60] developed various timing attacks against software implementations to determine the size of external resources, thus allowing an attacker to retrieve private information about a user browsing social networks, such as the user gender, age, or contacts. Van Goethem and Joosen [59] also exploited timing attacks to connect browsing contexts that are supposed to be isolated. Sanchez-Rola et al. [49] and Van Goethem and Joosen [59] exploited timing differences on browser extensions operation, allowing an attacker to enumerate the extensions or link to different browsing contexts. Vila and Köpf [62] exploited the browser shared event loop to monitor the behavior of other tabs in the same browser. Stone et al. [55] exploited the timings of various rendering tasks to retrieve personal information, e.g., browsing history.

3. Countermeasures in browsers

In this section, we systematize the different countermeasures proposed by academics or browser vendors. We have categorized such countermeasures in three classes:

C1: Isolation-based countermeasures,
C2: Timing-based countermeasures,
C3: Browser resources based.

3.1. C1: Isolation

Isolation is a staple of web security. Separating resources and communication between contexts reduce the threat surface drastically.

Same-origin policy. The same-origin policy [21] is a major security feature of the web. An origin is defined by the tuple (Protocol, Port, Host). The same-origin policy restricts read access to resources loaded from a different origin. This means that data from one website, e.g., authentication cookies, are not accessible from another website loaded in another tab. This countermeasure aims at mitigating, among others, attacks using browser resources by removing P5.

However, the same-origin policy has no impact on other type of timing attacks. In particular, Spectre [38] was also implemented in JavaScript, therefore demonstrating the limitations of the same-origin policy.

Site isolation. In response to transient execution attacks, Chrome developed a new security measure called site isolation [48], which forces each website, defined by the tuple (Protocol, Host), to run in a specific process, not shared with other websites. This prevents an attacker to access the mapped memory space of another website and mitigates the effect of some JavaScript-based transient execution attacks by preventing the access—including transient—to out of bound information. This feature was deployed in Chrome 67, and is currently being deployed in Firefox under the code name "Project Fission" [66]. At the time of writing, it is deployed selectively on a subset of the users of Firefox Nightly 84 [58]. It is likely to reach the stable version of Firefox shortly.

COOP/COEP. Site isolation was extended with the introduction of Cross-Origin Embedder Policy (COEP) and Cross-Origin Opener Policy (COOP) [10]. COOP ensures that a top-level window is isolated from other documents by putting them in a different browsing context group. For instance, if a website opens a pop-up whose origin is different from the website, the browser under COOP will put the pop-up in a separate process, similarly to site isolation, and will prevent direct interaction with the main document. COEP complements COOP by forcing the browser to only load trusted resources. If a resource has no explicit permission to be loaded, the browser will do nothing if COEP is enabled. Both COOP and COEP rely on policies defined through HTTP headers and they were both added in Firefox 79 and Chrome 83 [18], [19]. When they are enabled on a document, they guarantee a unique context group for the site and a safe loading of trusted resources.

JIT mitigations. Other countermeasures were studied by browser vendors to mitigate Spectre at JIT level. V8 developers have studied the impact of implementing retpolines [57] at the JIT level [56]. However, they found that these mitigations had serious impact on performance (a 2 or 3 times slowdown) and this countermeasure was not released.

Instead of preventing transient execution at the JIT level, they prevent the transient execution to read secret data [56], hence mitigating P4 instead of P3. They do so by reserving a register, tracking whether code is executed in a misspeculated branch. If so, they replace all loaded values by the value of said register, hence destroying potential out of bound information.

Conclusion. While these countermeasures were mostly implemented to prevent transient execution attacks, especially Spectre-PHT [38], they do not prevent other timing attacks, e.g., microarchitectural side channels or attacks exploiting browser features as, by design, they are not meant to mitigate P1, P2 or P3. As site isolation isolates the process' attack space, they also do not mitigate P4 for transient execution attacks targeting cross-address-space data, such as RIDL [61].

3.2. C2: Timers

The common prerequisite for timing attacks is access to timers. By removing timers, or lowering their resolution, most timing attacks would theoretically be mitigated. While the needed resolution depends on the attack, most hardware and transient attacks require high-resolution timers (P1), around 10 to $100\,\mathrm{ns}$. Removing such high-resolution timers would theoretically mitigate these attacks.

However, even by reducing timers' resolution, interpolation is still possible as long as attackers have access to timers with a constant time free running operation. To mitigate clock interpolation in browsers, adding a random jitter to API timers was proposed [39], [53]. Adding jitter to a measurement is equivalent to having clock periods of different time (with an average around the real clock period). This means that the interpolated time would vary significantly between clock periods for the same event, hence reducing the precision of clock interpolation.

Mitigations on `performance.now()`. After the first JavaScript timing attack [45] in 2015, the W3C advised that the resolution should be reduced to $5\,\mu\mathrm{s}$ to mitigate timing attacks, and browsers followed [36], [46].

However, in 2017 Schwarz et al. [53] demonstrated that it was still possible to recover high-resolution timers with the clamped resolution by using interpolation. This allowed an attacker to reimplement most timing attacks. In particular, they presented a clock interpolation-based timer with a resolution of $500\,\mathrm{ns}$.

After the disclosure of Spectre [38], browser vendors went to greater lengths to mitigate timing attacks. Figure 3 illustrates the changes that happened in a short time. Mozilla first clamped `performance.now()` to a resolution of $20\,\mu\mathrm{s}$ in Firefox 57.0.4 [64] then furthermore to $2\,\mathrm{ms}$ in Firefox 59 [11]. Browser vendors then introduced jitter to `performance.now()` in order to prevent clock interpolation: Firefox 60 set the resolution to $1\,\mathrm{ms}$ and added a jitter of $1\,\mathrm{ms}$ range [12], [25], while Chrome 64 set the resolution to $100\,\mathrm{ms}$ with a jitter of $100\,\mathrm{ms}$ [40]. The jitter being higher than the needed resolution for timing attacks, it is no longer possible to use `performance.now()` interpolation.

However, such drastic countermeasures also had an impact on web development [13]. JavaScript-based animation or video games often require sub-millisecond timers, which were no longer available. These countermeasures were however temporary, until other countermeasures were developed. Indeed, because of the security added by site isolation, most browsers have re-allowed high-resolution timers under certain conditions. Since Chrome 72, `performance.now()` has a resolution of $5\,\mu\mathrm{s}$ with a jitter of $5\,\mu\mathrm{s}$ under site isolation (which is present by default), whereas, since Firefox 79, it has a resolution of $20 \pm 20\,\mu\mathrm{s}$ without jitter, only when COOP/COEP is activated [7], [14].

The implementation of jitter differs across browsers. On Firefox 81, the jitter is computed using the SHA-256 hash function as follows [7]: the high-resolution timestamp is clamped to the lowest millisecond. The browser then uses SHA-256 on a tuple composed of the clamped time, a context-dependent seed, and a secret seed. It returns a random midpoint between the clamped time and the next millisecond. Depending on whether the precise timestamp is under or above this midpoint, the returned timestamp will be the lowest or highest millisecond, uniformly distributing clock edges between 0 and 2 ms. On Chromium, the jittered value is computed similarly, using murmur3 as the hash function [2].

Jitter requires a methodology change to evaluate auxiliary timers: as such, the work of Schwarz et al. [53] is not applicable anymore.

`SharedArrayBuffer`. Due to the powerful threat this `SharedArrayBuffer`-based clocks create, `SharedArrayBuffer` were disabled by default after the publication of Spectre [38] in Firefox 57.0.4 and all Chrome versions from 60. Once again, this measure is very restrictive for web developers and was only meant to be temporary. With the introduction of COOP/COEP and site isolation, browsers have re-enabled `SharedArrayBuffer`, claiming that access to a high-resolution timer is not a major threat in a strict isolation context. For instance, Chrome now supports `SharedArrayBuffer` since version 68 claiming that site isolation [48] is a sufficient defense, and as of Firefox 79, `SharedArrayBuffer` are available by default only under COOP/COEP.

Other timers. Many other timers exist in JavaScript. API timers, such as `Date.now()` or `Window.requestAnimationFrame()`, are subject to the same jitter as `performance.now()`. Furthermore, they have a resolution equal or worse than `performance.now()`, hence a higher measurement overhead. In the next sections of this paper, we focus on the two most potent timers: `performance.now()` and `SharedArrayBuffer`.

3.3. C3: Browser resources

When some timing leakage in the browser are not fixed by a more general approach like C1, there is a need to issue patches that specifically target them. For example, the history sniffing attack detailed by Paul Stone in 2013 [55] was only fixed in Firefox in 2020 by issuing repaints on both visited and unvisited elements [5], [6]. Regarding extension fingerprinting, the timing attacks detailed by Sanchez-Rola et al. [49] and Van Goethem and Joosen [59] were fixed by the Chromium team by changing how the checks for web accessible resources were made to prevent early-out exits [3], [4].

3.4. State of browser countermeasures

Figure 3 illustrates the evolution of browsers' main countermeasures and significant state-of-the-art attacks. P2 and P3 are not mitigated in browsers. Site isolation and COOP/COEP are important security updates on browsers, but do not mitigate hardware side-channel attacks and transient execution attacks other than Spectre-PHT. Furthermore, V8 developers claim that software fixes for transient execution attacks are "an unsustainable path" [56] as fixes on P3 are too performance consuming, and fixes on P4 only apply to specific attacks. Mitigation for P3 must be implemented at least at the OS level, if not at the hardware level.

The common prerequisites of timing attacks is P1, access to high resolution timers. This means that C2, timing based countermeasures, are the only generic defense for timing attacks, including future timing attacks.

Changes on timer-based countermeasures were motivated by a compromise between security—less effective timers means less threatening attacks—and usability. Indeed, countermeasures had major implications on web development and were meant to be temporary. `SharedArrayBuffer` are a powerful tool to build more complex websites, and access to high-resolution timers is necessary for some fields of web-design, such as animation or monitoring performances. There is a clear trade-off between strengthening the timers for security, and weakening then for easier development. However, we have found no quantitative study of the impact of the changes of values, especially regarding API timers resolution and jitter. Finding when and why each countermeasure was deployed often requires a deep dive in browser bug trackers.

With all the changes brought to countermeasures, especially timing-based, it is not clear to what extent P1 is mitigated, *i.e.*, to what extent browser are vulnerable to most timing attacks, and whether an optimal value for resolution and jitter exists. In the following sections, we evaluate the efficiency of timing-based countermeasures.

4. Evaluation tools

In this section, we present our threat model, and the properties of timers we are interested in: their resolution and measurement overhead. We focus on three timers or variants: `performance.now()` interpolated, `performance.now()` interpolated and amplified, and `SharedArrayBuffer`.

4.1. Threat model

We evaluate two popular browsers that are Mozilla Firefox and Google Chrome, letting aside Safari, Edge, or Tor Browser. It should be noted that we study the desktop version of these browsers, as all timing attacks we look at were not performed on mobile devices.

JavaScript-based timing attacks can be used in several threat models, each offering a different range of possibilities. First party attacks, where a user visits a malicious website, offer the most possibilities to the attacker. As the attacker can setup COOP/COEP as she wishes, she can freely use the unrestricted timers, based either on `performance.now()` or `SharedArrayBuffer`. In this model with Firefox 81, an attacker has access to `performance.now()` with 20 µs resolution and no jitter and to `SharedArrayBuffer`. However, the attacker has to redirect users to her malicious website.

With third party attacks, a user visits a legitimate website which has, e.g., a malicious advertisement controlled by the attacker. This allows the JavaScript code of the ad to be run on the user's machine. As opposed to the first party

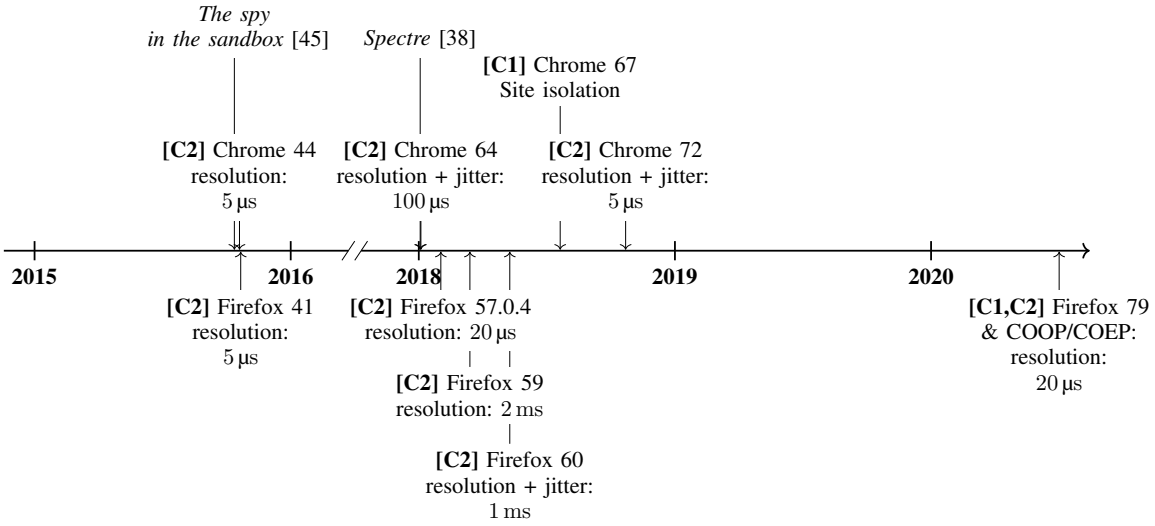


Figure 3: Timeline of timing attacks and browser countermeasures. Items in italics are significant attacks that caused the changes. Items preceded by **[C1]** are isolation-based countermeasures, and **[C2]** are timing-based countermeasures

model, the attacker does not control the top-level page. The attacker therefore cannot set up COOP/COEP. Without these flags, an attacker on Firefox 81 only has access to `performance.now()` clamped and jittered at 1 ms and no access to `SharedArrayBuffer`. This model presents a major threat surface, as it can be implemented in massively visited websites, e.g., Twitter or Facebook.

On Chrome 84, timer-based countermeasures are independent of COOP/COEP. This means that the first and third party models offer the same timing possibilities, *i.e.*, access to `SharedArrayBuffer` and `performance.now()` with a 5 µs resolution and jitter. However, Chrome 84 implements site isolation, which prevents certain transient execution attacks.

4.2. Measurement tools

One challenge is that measuring properties of high-resolution timers requires timers with an even higher resolution and precision, which are not available in standard browser releases. To solve this issue, we built a custom version of Firefox and Chromium that adds a new JavaScript method we call `performance.rdtsc()`, which executes the RDTSC instruction. The RDTSC instruction reads a CPU timestamp counter and returns it. As RDTSC is a cycle-accurate timer, we can use it to evaluate real-world auxiliary timers. Specifically, we use it to measure the resolution of both `performance.now()` interpolation and `SharedArrayBuffer`, as well as their measurement overhead.

We implemented our custom RDTSC builds in Firefox 81 and Chromium 84, built from sources on the same system. We disabled debug flags to get a version as close to the release version as possible. We modified files of the `performance` API [24] to add a new public method, `performance.rdtsc()`, which executes the native RDTSC and returns its value. However, RDTSC can be reordered by the out of order execution, therefore two calls to `performance.rdtsc()` would often be executed together, hence not timing an event. For instance, two calls to `performance.rdtsc()` with a call to `performance.now()` between would return the same timing difference than two subsequent calls to `performance.rdtsc()`. We fixed this by adding memory fences (`mfence` instructions) in our `performance.rdtsc()` method, before and after the call to RDTSC. Memory fences force the CPU to compute all operations preceding the fence before executing operations succeeding the fence. This prevents out-of-order execution of parts of code where the order of execution is critical. Appendix A illustrates our implementation of `performance.rdtsc()` for Firefox 81.

The `mfence` instructions add a constant overhead to the returned timestamp. Overhead is also added by the browser handling of the native code. On our system, the total overhead is around 1000 cycles. In all our following measurements, we measured the cycle difference between two events, and then the cycle difference between two subsequent `performance.rdtsc()`. This allows us to estimate the overhead for the current state of the CPU, depending on noise or core frequency. By removing this overhead, we retrieve a more reliable estimation of the event time in cycles.

4.3. Resolution

The resolution of a timer is the smallest value that it can measure. The smaller the resolution, the more precise the timer is. However, precisely evaluating the resolution of a JavaScript timer is a challenge as this requires an already precise timer.

`performance.now()` interpolation. To evaluate the resolution of `performance.now()`, we evaluate the maximum number of times we can increment a variable between two clock edges. By dividing the clock edge duration by this number of ticks, or measuring the time it takes to increment, we learn the shortest event that we can measure. However, because of the jitter, this value alone is not representative. It can vary significantly between measurements. Hence the importance of the standard deviation of the resolution.

It is also important to note that the duration of a "tick", an increment, can vary from a browser version to another. The main factor of this varying duration is due to the implementation of clock interpolation: after every increment, we check if `performance.now()` timestamp has changed. As the computation of the pseudo-random jitter often uses a hash function, it significantly increases the `performance.now()` computation time. Logically, the number of possible increments during the same duration changes according to this computation time.

`performance.now()` amplification. As the resolution is downgraded by pseudo-random values, we also study the impact of amplification. By repeating the measurement, an attacker can average the results and reduce the impact of jitter. This amplification allows an attacker to recover a higher resolution. As each repetition can increase the resolution, granting an absolute resolution for an amplification clock is illogical. The attacker has to compromise between the resolution and the number of repetitions she can afford.

In the following sections, we call error rate for cache hit/miss discrimination, the ratio of false hits, *i.e.*, misses computed as hits, and false misses, *i.e.*, hits computed as misses over the total number of experiments. This error rate allows to evaluate the efficiency of a timer in a real-world cache timing attack. We assume that a timer with a 5% error rate has a resolution sufficient to implement cache timing attacks, hence a resolution of around $10-100\,\mathrm{ns}$. This rate will be used as a standard for amplification in the following sections. We compute this rate by causing ourselves cache hits by calling a variable repeatedly and cache misses by calling a variable after evicting it from the cache by browsing through a large array.

Amplifying the results by repeating measurements is, however, not adapted to all attacks. Specifically, it only works when the attacker controls the event that creates the timing difference. For instance, in the case of a covert channel, the attacker can recreate the conditions for the same measurement several times. Some monitoring attacks do not tolerate repetitions, as the attacker cannot recreate the same conditions for the measurement. For instance, RIDL [61] steals in-flight data, and cannot select which data is in flight.

`SharedArrayBuffer`. For `SharedArrayBuffer`, the resolution is the time of a shared increment. The faster the increment, the higher the resolution. Other factors can impact the resolution of the timer, such as the computation time of reading a value from the array or potential concurrent accesses. Multithreading is handled differently in different browsers and this can be a potential source of randomness on the timestamp.

4.4. Measurement overhead

While the resolution indicates how precise a timer can be, an overhead is always incurred during a time measurement. In our study, we consider the following ones:

- Setting up a timer and handling it (starting it, stopping it, retrieving the results) always add an overhead. For `performance.now()`, it is very low as we rely on a built-in API in the browser. For `SharedArrayBuffer`, it is also comparatively low as starting a worker is very fast and communication with it is immediate.
- The repetition of ticks in timer interpolation (see Figure 1) is an overhead. Indeed, even if we measure the time of a short event, we still need to wait for the next clock edge of the timer that we rely on to retrieve the results. This is why, for `performance.now()` interpolation, the resolution can be very low but the overhead can be large as we are bound by `performance.now()` resolution.

To evaluate the measurement overhead of a given method, we use our custom `performance.rdtsc()` to measure the time difference between the start and the end of the measurement, without any event in between. A high measurement overhead does not necessarily prevent the attack but it plays a major role in the attack severity.

Ideal bit rate. A covert channel created with a high measurement overhead will yield a lower bit rate than one with a low measurement overhead, *i.e.*, the time to receive one bit will be higher with a high measurement overhead. Assuming each measure of time corresponds to a bit of information, and that the time the operation takes t_{op} is insignificant with respect to the overhead t_{oh}, we can define the ideal bit rate of such a timer to $\frac{1}{t_{oh}}\ \mathrm{bit/s}$.

Eviction set. Measurement time, and therefore measurement overhead, also impacts other attacks. For instance, building an eviction set in JavaScript [63], has a complexity in time measurements of $\mathcal{O}(C)$ where C is the size of the cache. Standard L3 caches have a size of several megabytes. In practice, on our system, an attacker must measure time around $100\,000$ times in order to build an eviction set. We use the computation of an eviction set as a standard as it is a critical step of attacks based on Prime+Probe. A high measurement time therefore leads to huge eviction set computation times, which may not be available to an attacker in a real-world scenario where a user only spends a few seconds or minutes on a web page.

5. Results

In this section, we present the results of our longitudinal studies of `performance.now()` and `SharedArrayBuffer` timers over many browser versions, as well as the impact of changes in countermeasures on state-of-the-art attacks. Table 2 presents the results of our comparative study.

5.1. Experimental setup

We ran measurements on a machine with an Intel CPU i5-8365U (Whiskey Lake generation) with $1.60\,\mathrm{GHz}$ frequency, under Fedora 31.

We performed experiments using every major release versions of Firefox from 53 (2017) to 80 (2020) and Chrome 48 (2016) to 84 (2020)[4]. We used Selenium WebDriver [8] and Python to automate tests. New versions of said browsers can easily be included in the test routine

4. Some releases of Chrome (versions 65 to 6) were unavailable on our system. We replaced them by the equivalent Chromium version. To our knowledge, the timer implementations are the same on both browsers.

Table 2: Comparison of timers' resolution, measurement overhead and ideal bit rate for an error rate of 5%. We used a frequency of 1.60 GHz and the resolution is displayed in ns. We can observe that the `SharedArrayBuffer`-based timer is by far the most efficient, as they offer a better resolution and a lower measurement overhead than timers based on `performance.now()` on all browsers. Timers based on `performance.now()` clocks are still highly effective in Chrome 84 and Firefox 81 with COOP/COEP.

Browser	Timer	Resolution [cycles]	Converted resolution	Measurement overhead [cycles]	Ideal bit rate [bit/s]
Chrome 84	`SharedArrayBuffer`	20	10 ns	40	1×10^8
	`performance.now()` interpolation [3]	100-1000	100 ns	7.2×10^4	22×10^4
Firefox 81	`SharedArrayBuffer`	20	10 ns	42	1×10^8
	`performance.now()` interpolation	100-1000	100 ns	2.9×10^7	60
	Interpolation with COOP/COEP	100-1000	100 ns	7×10^4	22×10^4

Table 3: Duration of a tick, using `performance.rdtsc()`.

Browser	Average tick duration [cycles]	Standard Deviation [cycles]
Firefox 81	200	10
Unjittered Firefox 81	100	9
Chrome 84	150	9

to see the evolution of timers without a deep dive in documentation and source code. Our scripts access test pages hosted on a local server, each page containing our JavaScript benchmark code. Our scripts also handle browser evolution, e.g., the different flags required by the use of `SharedArrayBuffer`, or the eventual activation of COOP/COEP.

5.2. Longitudinal study of `performance.now()` interpolation

5.2.1. Simple interpolation

Resolution. We seek to measure the resolution of the `performance.now()` method when interpolated. Two factors influence this resolution: (1) the duration of a tick, and (2) the jitter.

The shorter the duration of a tick, the higher the resolution. Table 3 illustrates the duration of a tick on different browsers and versions. It is important to note that while the increment part of the tick has a relatively steady computation time through browser versions, the computation time of the call to `performance.now()` varies a lot depending on the version. For instance, on Firefox 81, a call lasts around 200 `performance.rdtsc()` cycles, while it only takes 100 cycles on an unjittered version. At this scale, the jitter is indeed a time-consuming operation as the browser uses a hash function to compute a random midpoint.

To understand the impact of resolution and jitter on measurements, we measure the number of ticks between two clock edges. This is equivalent to measuring no event with a `performance.now()` interpolation-based clock. Without any jitter, the number of ticks is always the same. Since the standard deviation is low, an accurate timing can be retrieved [53]. With jitter, the story is obviously different. If the resolution of `performance.now()` is low, the time between two clock edges increases, so the number of ticks increases. If the jitter is high, the number of ticks will be spread on a wide range of values between measurements, and therefore the standard deviation increases. Table 4 illustrates this evolution by showing statistics about the average number of ticks between two clock edges. As a reminder, a tick is composed of an increment as well as a call to `performance.now()`. The results were computed using 100 000 samples. We see that jittered versions often have a high variance to average ratio, e.g., on Firefox 81 without COOP/COEP, the standard deviation is half of the average.

Table 4: Comparison of the average number of ticks per browser. The duration of a tick can vary accordingly to the browser or the presence of jitter.

Browser	Average number of ticks	Standard Deviation	Announced resolution	Jitter
Firefox 81-latest + COOP/COEP	300	40	20 µs	○
Firefox 60-latest[5]	3310	1510	1 ms	●
Firefox 41-57[6]	70	10	5 µs	○
Chrome 72 and later	12	7	5 µs	●
Chrome 64-71	400	200	100 µs	●
Chrome 64 and former	10	2.5	20 µs	○

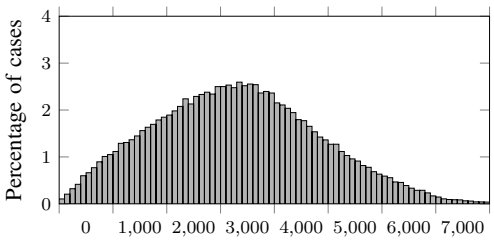
Figure 4a and Figure 4b illustrate the variation of the number of ticks in a single clock period for Firefox 81 and Chrome 84 respectively. The data follows a triangle distribution, with the top value at around 3300 and 12 ticks respectively. A precise timestamp can be clamped to a certain value if the timestamp is smaller than the clamped value but higher than the random midpoint, or if the timestamp is higher than the clamped value but lower than the random midpoint. As both midpoints are computed following a uniform distribution, the result of the sum of these two distributions is a triangle distribution. On the contrary, as can be seen on Figure 4c, the behavior of an unjittered Firefox is very different. The distribution of ticks in a clock period is grouped between 280 and 320 and does not follow a triangle distribution. The difference in the number of ticks stems from the different base resolutions and jitters. Knowing the distribution of the number of ticks in a single clock period means that an attacker can gather more information from a single measurement.

The resolution for the different versions vary drastically between browsers. Figure 5 illustrates the timings for

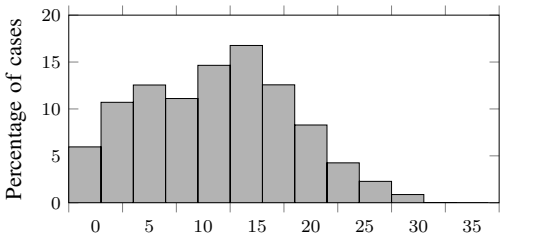
4. COOP/COEP has no impact on timers in Chrome 84.

5. Without COOP/COEP for Firefox 81 and later.

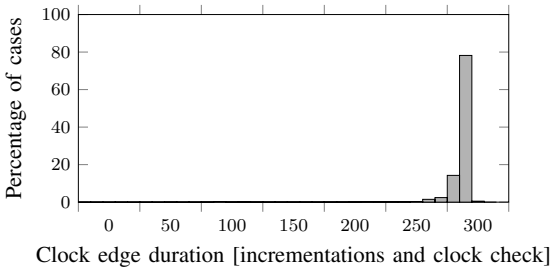
6. Versions 58 and 59 experienced many timer changes and do not appear here.



(a) Firefox 81 without COOP/COEP (jitter).

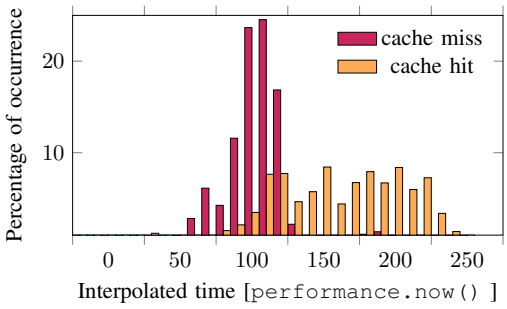


(b) Chrome 84 (jitter).

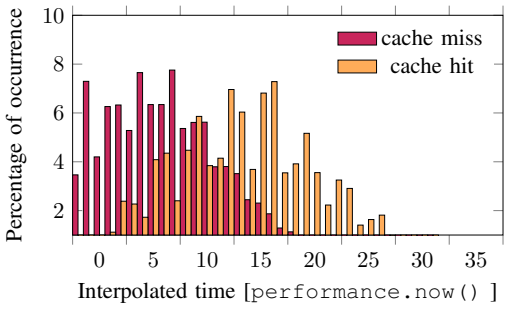


(c) Firefox 81 with COOP/COEP (no jitter).

Figure 4: Distribution of number of ticks in a single clock edge.



(a) Firefox 81 with COOP/COEP.



(b) Chrome 84.

Figure 5: Histogram for cache hits and cache misses.

cache misses and cache hits by using `performance.now()` interpolation on Firefox 81 with COOP/COEP and Chrome 84. Distributions must be differentiated in order to implement cache attacks. Note that, contrary to what we expect, hit timings are higher to miss timings on this graph. This is because `performance.now()` interpolation measures the time between the end of the event and the end of the clock edge. This means that a long event will yield a short interpolated time, whereas a quick event will yield a long interpolated time. The lack of jitter on Firefox has an impact on the histogram, where the difference between cache hits and cache misses is clearer.

On Firefox 81 without COOP/COEP, an attacker has a 30% error rate on distinguishing cache hits from cache misses when targeting a sub-100 ns resolution. On Chrome 84, regardless of COOP/COEP, the error rate stands at 20%. This makes for a highly unreliable clock, and drastically hinders the development of attacks. On Firefox 81 with COOP/COEP, the lack of jitter drops this error rate to only 11%.

The lower the error rate, the more threatening timing attacks with this timer are. This means that, with interpolation alone, timers based on `performance.now()` on Firefox 81 with COOP/COEP offer the best timer in term of resolution, as they allow to implement cache-based timing attacks with a relatively low error rate. Error rates of Chrome 84 and Firefox 81 without COOP/COEP are too error-prone to implement attacks that require high precision.

Measurement overhead. We now evaluate the measurement overhead on Firefox 81 and Chrome 84. We use our custom `performance.rdtsc()` to retrieve the timestamp before and after the measurement of an empty event with `performance.now()` interpolated. We found that, on Firefox 81 without COOP/COEP, the measurement overhead averages around 1.8 million cycles. On our 1.60 GHz CPU, this represents around 1.1 ms. This is coherent with the announced resolution, 1 ms, as the measurement always takes at least this time between two clock edges. On Chrome 84, a measurement using interpolation takes 9,000 cycles, corresponding to 5.5 µs with our average frequency. On Firefox 81 with COOP/COEP, a measurement using interpolation takes 35,000 cycles, averaging to 21 µs on our system. The slight difference may come from the potential change in CPU frequency under a lot of calculations and noise.

With interpolation alone, Chrome 84 has the shorter measurement overhead, which allows to implement attacks that run faster. Namely, a cache covert channel built with this timer could ideally retrieve a resolution of 180 kbit/s, against 50 kbit/s for Firefox with COOP/COEP and 900 bit/s for Firefox without COOP/COEP. However, the ideal bit rate does not take into account the different error rates between browsers.

Conclusion. Although Chrome 84 offers the best base resolution with `performance.now()` interpolation, the jitter causes a high error rate for high-resolution attacks. This error rate would slow, if not prevent, the implementation of attacks using interpolation alone. Due to the lack of jitter, Firefox 81, with COOP/COEP enabled, offers the lowest error rate with interpolation. It could be used to

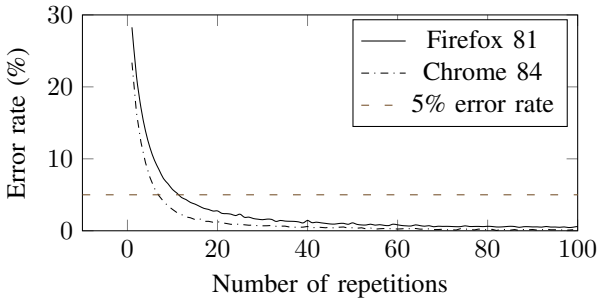


Figure 6: Hit / miss error rate in function of repetitions, using `performance.now()` interpolation on Firefox 81 without COOP/COEP and Chrome 84

built error-tolerant cache attacks, such as a covert channel. Firefox 81 without COOP/COEP has a high error rate (30%) and a high measurement time, rendering it inefficient for timing atttacks.

With interpolation alone, Firefox 81 with COOP/COEP is the only browser where an attacker can build clocks based on `performance.now()` that can efficiently run timing attacks.

5.2.2. Interpolation and amplification

The jitter being longer than most of the events we wish to time, it is not possible to simply use clock interpolation to retrieve a high resolution. To get a more accurate timer, an attacker therefore needs to reduce the impact of the pseudo-random jitter. On both Chrome and Firefox, the jitter is deterministically defined by computing a random midpoint between the highest and lowest clamped values. However, the time of a jittered clock edge follows a triangle distribution around the precise timing value. This means that each measurement has a slight tendency towards the real value. By repeating the measurement several times, an attacker can, therefore, achieve a higher precision.

Resolution. To evaluate the impact of amplification on the resolution, we repeated the measurements and computed each time the average error rate on cache hit/miss discrimination. Figure 6 illustrates the results for Firefox 81 without COOP/COEP and Chrome 84. The error rate follows a logarithmic decrease with repetitions for both browsers. An attacker can reach a 5% error rate by repeating the measurements 15 times on Firefox 81 without COOP/COEP and 8 times for Chrome 84. On Firefox 81 with COOP, a mere 2 repetitions grant a 4% error rate. Comparatively, on an older version of Chrome, namely version 71, an attacker needed 12 repetitions in order to reach the 5% error rate.

Measurement overhead. A single measurement without amplification takes at least the duration of a clock edge. That is, on average, $1\,\mathrm{ms}$ for Firefox 81 and $5\,\mu\mathrm{s}$ for Chrome 84. In the best scenario, repeating the measurement n times will increase the measurement time by n.

Specifically, repeating the measurement 15 times on Firefox 81 without COOP/COEP takes 29 million cycles on average. This represents a measurement overhead of $18\,\mathrm{ms}$ on our $1.60\,\mathrm{GHz}$ CPU. This means that building a covert channel using this timer as a receiver yields, at best, an ideal bit rate of $60\,\mathrm{bit/s}$.

On Chrome 84, repeating the measurement 8 times grant a measurement overhead of $72\,000$ custom `performance.rdtsc()` cycles, or around $45\,\mu\mathrm{s}$. This would grant an ideal bit rate of $22\,\mathrm{kbit/s}$.

On Firefox 81 with COOP/COEP, with amplification, the measurement time would be $70\,000$ custom `performance.rdtsc()` cycles, granting as well an ideal bit rate of $22\,\mathrm{kbit/s}$.

Conclusion. The introduction of non timer-based countermeasures, specifically site isolation for Chrome and COOP/COEP for Firefox[7] have led browser vendors to take a step backward on timer-based countermeasures. Lowering the resolution and the jitter has an impact on the measurement overhead, hence on real-world attack time. Starting from Chrome 72, the resolution of `performance.now()` has been set to $5\,\mu\mathrm{s}$ with jitter. Starting from Firefox 79, when COOP/COEP are set, the resolution of `performance.now()` is set to $20\,\mu\mathrm{s}$ without jitter.

A cache covert channel based on `performance.now()` amplification on Firefox 81 with COOP/COEP has an ideal bit rate 360 times higher than without COOP/COEP. Similarly, an attacker with COOP/COEP can build an eviction set in around a few seconds, where an attacker without COOP/COEP would need several minutes. The same goes with Chrome: on version 71, with a resolution of $100\,\mu\mathrm{s}$, we obtain, with amplification, an ideal bit rate of $800\,\mathrm{bit/s}$, 30 times lower than with the new Chrome 84 timer values. An attacker using Chrome 84 can build an eviction set in a matter of seconds, as opposed to several hundred seconds for Chrome 71.

The efficiency of the jitter is also highlighted by these results: between Chrome 84, with jitter, and Firefox 81 with COOP/COEP, without jitter, the measurement overhead is similar, when Chrome offers a better base resolution.

These changes have a massive impact on a browser-based threat model, where a script running with high performance for several minutes is highly suspicious, and can be detected by browsers. In a first-party scenario, an attacker can easily setup COOP/COEP and use a powerful timer granted by this change of resolution and timer. For both browsers, the impact of this change in resolution is a massive increase in the threat of timing attacks, as it means that P1 is less mitigated than on older versions. Changing the timer resolution does not prevent attacks, but impacts massively the time needed to execute them, which is an important factor in web security, as the user has to stay on the malicious web page for the duration of the attack. Setting a lower resolution and lower jitter has a double impact on the measurement overhead: as each clock edge is shorter, each measurement is faster. The attacker therefore needs fewer repetitions to eliminate the jitter, furthermore accelerating attacks.

5.3. Longitudinal study of `SharedArrayBuffer`-based clocks

Resolution. The resolution of `SharedArrayBuffer` only depends on the computation time of an increment of the shared value. The smaller the value, the higher the resolution. We used `performance.rdtsc()` before and

7. Implementations of site isolation for firefox are developed. COOP/COEP are implemented in Chrome but have no impact on timers.

after incrementing the array buffer and tested two different types of increments: a simple increment (`value++`) and the built-in method `Atomics.add` [28]. The Atomics API offers methods to safely use shared memory and handle conflicts.

We observed that a simple increment takes in the order of 20 custom cycles on both Chrome 84 and Firefox 81 whereas using Atomics takes 100. Logically, handling concurrent accesses introduces an overhead. We also noted that the first increment is slower, by an order of magnitude, probably from a cache impact. As the sub-thread systematically increases the same value, we excluded it and focused on the following values. `SharedArrayBuffer` offers a resolution in the order of 20 CPU cycles, or 10 ns on our CPU.

Measurement overhead. The measurement overhead for `SharedArrayBuffer` is roughly the time it takes to read a value in the shared array twice: before and after the event. We again used our custom `performance.rdtsc()` to measure this time, and we tested two methods: standard array access and `Atomics.load`. On Firefox 81, a standard access lasts in the order of 42 custom cycles, opposed to 160 with Atomics API. As the lowest measurement overhead is preferable, we used the standard access. The measurement overhead on Chrome 84 with the standard access is 40 `performance.rdtsc()` cycles, or 20 ns on a 1.60 GHz CPU. It is similar on Firefox 81.

Conclusions. `SharedArrayBuffer` have been disabled by default in Chrome 60 and Firefox 57.0.4 to mitigate Spectre. With the introduction of mitigations to transient execution attacks, they have been reimplemented. They are available by default in Firefox 79 with COOP/COEP, and by default in Chrome 68.

`SharedArrayBuffer` based timers are, by far, the most powerful timer available in browsers. Table 2 illustrates the resolution and measurement time for `SharedArrayBuffer`-based clocks. They offer a resolution of 20 cycles and a measurement overhead of 40 cycles, equivalent to 10 ns and 20 ns respectively on a 1.60 GHz CPU. The offered resolution is sufficient to implement all known timing attacks. In addition, they have a very low measurement overhead and do not need amplification. An attacker using `SharedArrayBuffer` to build a covert channel can achieve an ideal bit rate of 50 Mbit/sec on both browsers. This is 800 000 times higher than with `performance.now()` on Firefox 81 without COOP/COEP, and 2000 times higher than Chrome 84 and Firefox 81 with COOP/COEP.

An attacker could theoretically create an eviction set in less tenth of milliseconds by using this method. However, the algorithm required to create an eviction set have other sources of heavy computation, and still ran within a few hundred milliseconds on our system.

Free access to such a powerful timer shows that P1 is not mitigated. Excluding Spectre-PHT, most of state-of-the-art attacks are theoretically possible under the current state of Chrome and Firefox, as P1, P2 and P3 are not mitigated, and P4 only partially mitigated. Other transient execution attacks, such as RIDL [61] or ret2spec [43] are not prevented by COOP/COEP or site isolation, and are still implementable under certain conditions as the shared system resources are still accessible. In Firefox, where `SharedArrayBuffer` are restricted to sites with COOP/COEP, an attacker can still use them in a first party scenario. Logically, countermeasures on `performance.now()` or other timers are secondary when `SharedArrayBuffer` are available, because they allow the creation of way more potent timers.

6. Discussion

In this section, we discuss the current state of timers in browsers and the challenges surrounding them.

Usability vs security and the lack of proper mitigations. In Section 3, we have seen that the behavior of API timers like `performance.now()` has varied over the years in response to newly discovered attacks. Timing-based countermeasures (C2) have been widely implemented, but to various degrees of strength. On one hand, browser vendors decreased timer's resolution and added jitter to provide a more secure environment for their users. But on the other hand, their decisions appear arbitrary in retrospect as changes to timers were made without any concrete evidence of their effectiveness. For example, despite the study of Oren et al. in 2015 [45], it was not until 2018 that we saw the first implementation of jitter in web browsers. In that time frame, the decrease in timer resolution could simply be bypassed through interpolation [53]. Moreover, since some genuine applications are directly impacted by the lack of real-time precision provided by the affected timers [13], each browser vendor has tried to balance security with usability: Chrome never went above 100 µs, Firefox hovered around the 1 ms mark and got down recently to 20 µs while the Tor Browser has kept a 100 ms resolution since April 2015. The same goes for `SharedArrayBuffer`: they are enabled by default on Chrome, Edge and Opera, enabled under COOP/COEP for Firefox, and disabled on Safari and Tor.

This lack of consensus between vendors highlights how uncertain the industry is with the provided fixes. The results provided in this paper show that the current timer-based countermeasures (C2) are a good first step towards protecting users but they still fall short of fully protecting them against a large range of timing attacks, as P1 is a shared prerequisites between most of the timing attacks, and allegedly future timing attacks.

An alternative that could be considered by vendors is to put access to a high-resolution timer, based on `performance.now()` or `SharedArrayBuffer`, behind a permission. This way, when a developer needs it for an application or a game, she would need to ask the user for an explicit permission. This would prevent stealthy usage of API timers for timing attacks and all vendors could adopt the exact same very low resolution by default as it would not break pages not needing it.

The false sense of security created by resource isolation. Recent trends on the development of the web as a platform have focused a lot on controlling what is running on a web page. Isolation-based countermeasures (C1) are at the core of browser security. Mechanisms like SRI [22], CORP [20], COOP/COEP [10], site isolation [48] or even a proposal for better cookies [65] are all pushing the web forward in strengthening security. Yet, when it comes to

timing attacks, all these new barriers create a false sense of security even though some attacks are definitely now much harder to pull off than before. By design, these countermeasures are not meant to mitigate P2 nor P3, and only a subset of P4. They are a great step forward in term of security, but are not sufficient alone to mitigate the vast majority of timing attacks. While they greatly limit the possibility of a third-party attack when everything is set up properly on a web page, an attacker can still host her own malicious domain and conduct the attack from there. Moreover, the recent increase of timer resolution coupled with the reactivation of `SharedArrayBuffer` represents a massive step back in security where some attacks can be run in similar conditions to the ones in 2015. Our study highlights the dangers of coming back to such a state, and we hope browser vendors will recognize their mistakes by considering stronger mitigations to P1.

The need to mitigate timing attacks at the OS or hardware level. Since timers can represent such an important threat to the security in a web browser, one can wonder if it would be possible to have a browser without timers. One approach is to try and make the browser deterministic as prototyped by Cao et al. with DeterFox [17]. By transforming a physical clock into a logical one, they change the behavior of known timers so that they do not return the time a request takes but return the number of operations that are being executed. While promising, this approach presents several shortcomings. First, they have to patch each known clock individually to instill this new behavior. A deep re-engineering of the browser would have to be made to possibly cover all implicit clocks. The second problem is that a logical clock loses all its meaning in the context of a real-time application. Some programs need to access the actual physical clock of the system to function properly and having a deterministic clock would present a lot of hurdles for developers.

In the end, we believe that everything running in a web browser has the potential to be a timer. This means that fully mitigating P1 seems out of reach. While browsers in 1995 mainly rendered static pages, the web has kept growing since then and it is now this rich and dynamic platform that can not only render pages but it is also the home of real-time communications and virtual reality, to name but a few. While some alternatives like putting access to high-resolution timers behind a permission can improve security, we simply have to learn to live with timers as they are such an intricate but integral part of the web.

As a consequence, mitigating timing attacks at the browser level is not the only solution as we can develop solutions at both the OS and hardware level to provide stronger security against such threats, particularly mitigating P2 and P3. Software mitigations to transient execution still have, at this point in time, a high cost in performance. Browser vendors claim that mitigation must come from a lower level than software [56] as hardware and transient execution attacks originate from micro-architectural optimizations.

7. Related Work

Other leads have been studied to try and prevent JavaScript based timing attacks.

Kohlbrenner et al. [39] presented Fuzzyfox, a Firefox fork where timer resolution is degraded and jittered. This includes `performance.now()` as well as other existing JavaScript auxiliary timers, such as `Window.requestAnimationFrame()`. Fuzzing techniques are now implemented in most browsers, and include auxiliary clocks.

Schwarz et al. [52] have developed JavaScript Zero, a dynamic permission model. It aims at reducing threats by controlling which JavaScript features are available in a specific context. The authors claim fixing most JavaScript timing attacks (pre-Spectre) as well as any JavaScript based 0-days. They implemented their model in a modified version of Google Chrome, with only a small overhead. To prevent timing attacks, in their most secure settings, they reduce `performance.now()` resolution to 100 ms with jitter. They also replace `SharedArrayBuffer` by a single threaded polyfill, mitigating the auxiliary clock threat. These measures are efficient against timing attacks, however they do not go in the current browser's direction. Indeed, these restrictions are particularly constraining for web developers. For instance, it is impossible to implement 60 Hz animations with such a resolution.

Cao et al. [17] proposed an other approach to mitigate timing attacks. They introduce DeterFox, a fork of Firefox which implements deterministic computing. This means that a potential attacker will always obtain the same timing for an event.

8. Conclusion

We have studied the evolution in the last years and the current state of JavaScript-based timers and timing attacks for Chrome and Firefox, evaluating the resolution and measurement overhead for the two most efficient timers: `performance.now()` and `SharedArrayBuffer`. Timer-based countermeasures like clamping the resolution and adding jitter do not prevent attacks, but increase the time needed to exploit these attacks. Unlike in native environment, exploitation time is an important factor in web-based attacks, where the victim may not stay on the same web page for more than a few seconds or minutes. Unfortunately, the current trend of browser vendors undermining previous timer-based countermeasures by re-enabling `SharedArrayBuffer` and increasing the resolution to improve usability reopens the door to many practical timing attacks, that were thought to be mitigated years ago. For example, the re-introduction of `SharedArrayBuffer` on Chrome brought a 2 000-fold increase in covert channel capacity, compared to `performance.now()` alone. This is even more dramatic on Firefox, where the increase is 800 000-fold. Powerful countermeasures such as site isolation and COOP/COEP only prevent a sub-class transient execution attacks, thus browsers are currently vulnerable to other transient execution attacks, as well as a large range of timing attacks, with a large threat surface.

Availability

The main purpose of this work is to provide an overview of the evolution of JavaScript's timers over the years, allowing the community to start informed

discussions in this area and motivate further research. To ensure the repeatability of our findings, all developed code and data artifacts are available on https://github.com/thomasrokicki/in-search-of-lost-time. This includes all the tests that we ran for this study, the different patches for Chrome and Firefox along with the complete data that form the basis of our results.

Acknowledgments

We thank the reviewers and our shepherd for their helpful feedback. This work benefited from the support of the ANR-19-CE39-0007 MIAOUS project and from the ASCOT project of the Hauts-de-France STaRS framework.

References

[1] Noscript. https://noscript.net/.

[2] time_clamper.cpp. https://source.chromium.org/chromium/chromium/src/+/master:third_party/blink/renderer/core/timing/time_clamper.cc. Accessed: 2020-10-12.

[3] Issue 611420: WebAccessibleResources take too long to make a decision about loading if the extension is installed. https://bugs.chromium.org/p/chromium/issues/detail?id=611420, 2017.

[4] Issue 709464: Detecting the presence of extensions through timing attacks (including Incognito) - Chromium bug tracker. https://bugs.chromium.org/p/chromium/issues/detail?id=709464, 2017.

[5] Always restyle / repaint when a visited query finishes – Mozilla Central. https://hg.mozilla.org/mozilla-central/rev/89fad029456188f03a670ef5f08a5d0856a728b1, 2019.

[6] Bug 884270: Link Visitedness can be detected by redraw timing – Bugzilla. https://bugzilla.mozilla.org/show_bug.cgi?id=884270, 2020.

[7] nsrfpservice.cpp, firefox sourcecode. https://hg.mozilla.org/mozilla-central/file/tip/toolkit/components/resistfingerprinting/nsRFPService.cpp, October 2020.

[8] SeleniumHQ browser automation. https://www.selenium.dev/, oct 2020.

[9] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *S&P*, 2015.

[10] Anne van Kesteren Artur Janc, Charlie Reis. Coop and coep explained. https://docs.google.com/document/d/1zDlfvfTJ_9e8Jdc8ehuV4zMEu9ySMCiTGMS9y0GU92k/edit. Accessed: 2020-10-05.

[11] Bugzilla. Reduce timer resolution to 2ms. https://bugzilla.mozilla.org/show_bug.cgi?id=1435296, feb 2018.

[12] Bugzilla. Set timer resolution to 1ms with jitter. https://bugzilla.mozilla.org/show_bug.cgi?id=1451790, apr 2018.

[13] Bugzilla. Unanticipated security/usability degradation from precision-lowering of performance.now() to 2ms. https://bugzilla.mozilla.org/show_bug.cgi?id=1440863, feb 2018.

[14] Bugzilla. Check crossoriginisolated for all nsrfpservice::reducetimeprecision* callers. https://bugzilla.mozilla.org/show_bug.cgi?id=1586761, apr 2020.

[15] 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.

[16] 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 Symposium*, 2019.

[17] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. Deterministic browser. In *CCS*, 2017.

[18] MDN contributors. Cross-origin-embedder-policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy.

[19] MDN contributors. Cross-origin-opener-policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy.

[20] MDN contributors. Cross-origin resource policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Cross-Origin_Resource_Policy_(CORP).

[21] MDN Contributors. Same-origin-policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed: 2020-10-06.

[22] MDN contributors. Subresource integrity. https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity.

[23] MDN Contributors. Date.now() api. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/now, Sept 2020.

[24] MDN contributors. Performance api. https://developer.mozilla.org/en-US/docs/Web/API/Performance_API, October 2020.

[25] MDN Contributors. Performance.now() api. https://developer.mozilla.org/fr/docs/Web/API/Performance/now, Sept 2020.

[26] MDN Contributors. Window.requestanimationframe() api. https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame, Sept 2020.

[27] MDN Contributors. Window.settimeout() api. https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout, Sept 2020.

[28] ECMA. Atomics.add - standard. https://www.ecma-international.org/ecma-262/#sec-atomics.add. Accessed: 2020-09-30.

[29] ECMA. Sharedarraybuffer objects. https://tc39.es/ecma262/#sec-sharedarraybuffer-objects. Accessed: 2020-09-30.

[30] ECMA. Standard ecma-262. https://www.ecma-international.org/publications/standards/Ecma-262.htm. Accessed: 2019-09-02.

[31] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, volume 17, page 26, 2017.

[32] Ilya Grigorik. High resolution time level 2. https://www.w3.org/TR/hr-time-2/, Nov 2019.

[33] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed javascript. In *ESORICS*, 2015.

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

[35] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *S&P*, 2011.

[36] Sakamoto K. Reduce resolution of performance.now to prevent timing attacks. https://bugs.chromium.org/p/chromium/issues/detail?id=506723, Jul 2015.

[37] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[38] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *S&P*, 2019.

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

[40] Sami Kyöstilä. Clamp performance.now() to 100us. https://chromium-review.googlesource.com/c/chromium/src/+/849993, Jan 2018.

[41] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed javascript. In *ESORICS*, 2017.

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

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

[44] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. Fingerprinting information in javascript implementations. *Proceedings of W2SP*, 2(11), 2011.

[45] 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.

[46] Yossi Oren. "spy in the sandbox" - security issue related to high resolution time api. https://bugzilla.mozilla.org/show_bug.cgi?id=1167489, may 2015.

[47] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *CT-RSA*, pages 1–20. Springer, 2006.

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

[49] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *USENIX Security Symposium*, 2017.

[50] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. Clock around the clock: Time-based device fingerprinting. In *CCS*, 2018.

[51] Michael Schwarz, Florian Lackner, and Daniel Gruss. Javascript template attacks: Automatically inferring host information for targeted exploits. In *NDSS*, 2019.

[52] Michael Schwarz, Moritz Lipp, and Daniel Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS*, 2018.

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

[54] 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 Symposium*, 2019.

[55] Paul Stone. Pixel perfect timing attacks with HTML5, 2013.

[56] Ben L. Titzer and Jaroslav Sevcik. A year with spectre: a v8 perspective. https://v8.dev/blog/spectre. Accessed: 2021-02-12.

[57] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018.

[58] Harry Twyford. These weeks in firefox: Issue 82. https://blog.nightly.mozilla.org/2020/10/21/these-weeks-in-firefox-issue-82/, oct 2020.

[59] Tom van Goethem and Wouter Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[60] Tom van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *CCS*, 2015.

[61] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *S&P*, 2019.

[62] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in chrome. In *USENIX Security Symposium*, 2017.

[63] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *S&P*, 2019.

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

[65] Mike West. Incrementally better cookies. https://mikewest.github.io/cookie-incrementalism/draft-west-cookie-incrementalism.html. Accessed: 2020-11-05.

[66] Mozilla Wiki. https://wiki.mozilla.org/Project_Fission.

[67] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

Appendix A.
Custom RDTSC implementation

We modified the following files in order to implement `performance.rdtsc()`:

A.1. Firefox 81

mozilla-central/dom/performance/Performance.cpp

```
std::uint64_t Performance::Rdtsc(){
    unsigned int lo,hi;
    __asm__ __volatile__ ("mfence");
    __asm__ __volatile__ ("rdtsc" : "=a
        " (lo), "=d" (hi));
    __asm__ __volatile__ ("mfence");
    return ((std::uint64_t)hi << 32) |
        lo;
}
```

mozilla-central/dom/performance/Performance.h

```
std::uint64_t Rdtsc();
```

mozilla-central/dom/webidl/Performance.webidl

```
typedef double uint64_t;
uint64_t rdtsc();
```

A.2. Chromium 84

chromium/src/third_party/blink/renderer/core/timing/performance.cc

```
std::uint64_t Performance::rdtsc(){
    unsigned int lo,hi;
    __asm__ __volatile__ ("mfence");
    __asm__ __volatile__ ("rdtsc" : "=a
        " (lo), "=d" (hi));
    __asm__ __volatile__ ("mfence");
    return ((std::uint64_t)hi << 32) |
        lo;
}
```

chromium/src/third_party/blink/renderer/core/timing/performance.h

```
std::uint64_t rdtsc();
```

chromium/src/third_party/blink/renderer/core/timing/dom_high_res_time_stamp.idl

```
typedef double uint64_t;
```

chromium/src/third_party/blink/renderer/core/timing/performance.idl

```
uint64_t rdtsc();
```