Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses

Anatoly Shusterman, *Ben-Gurion University of the Negev;* Ayush Agarwal, *University of Michigan;* Sioli O'Connell, *University of Adelaide;* Daniel Genkin, *University of Michigan;* Yossi Oren, *Ben-Gurion University of the Negev;* Yuval Yarom, *University of Adelaide and Data61*

https://www.usenix.org/conference/usenixsecurity21/presentation/shusterman

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses

Anatoly Shusterman
Ben-Gurion Univ. of the Negev
shustera@post.bgu.ac.il

Ayush Agarwal
University of Michigan
ayushagr@umich.edu

Sioli O'Connell
University of Adelaide
sioli.oconnell@adelaide.edu.au

Daniel Genkin
University of Michigan
genkin@umich.edu

Yossi Oren
Ben-Gurion Univ. of the Negev
yos@bgu.ac.il

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

Abstract

The "eternal war in cache" has reached browsers, with multiple cache-based side-channel attacks and countermeasures being suggested. A common approach for countermeasures is to disable or restrict JavaScript features deemed essential for carrying out attacks.

To assess the effectiveness of this approach, in this work we seek to identify those JavaScript features which are *essential* for carrying out a cache-based attack. We develop a sequence of attacks with progressively decreasing dependency on JavaScript features, culminating in the first browser-based side-channel attack which is constructed entirely from Cascading Style Sheets (CSS) and HTML, and works even when script execution is completely blocked. We then show that avoiding JavaScript features makes our techniques architecturally agnostic, resulting in microarchitectural website fingerprinting attacks that work across hardware platforms including Intel Core, AMD Ryzen, Samsung Exynos, and Apple M1 architectures.

As a final contribution, we evaluate our techniques in hardened browser environments including the Tor browser, DeterFox (Cao el al., CCS 2017), and Chrome Zero (Schwartz et al., NDSS 2018). We confirm that none of these approaches completely defend against our attacks. We further argue that the protections of Chrome Zero need to be more comprehensively applied, and that the performance and user experience of Chrome Zero will be severely degraded if this approach is taken.

1 Introduction

The rise in the importance of the web browser in modern society has been accompanied by an increase in the sensitivity of the information the browser processes. Consequently, browsers have become targets of attacks aiming to extract or gain control of users' private information. Beyond attacks that target software vulnerabilities and attacks that attempt to profile the device or the user via sensor APIs, browsers have also been used as a platform for mounting microarchitectural side-channel attacks [22], which recover secrets by measuring the contention on microarchitectural CPU components.

While traditionally such attacks were implemented using native code [7, 29, 49, 58, 60, 79, 80], recent works have demonstrated that JavaScript code in browsers can also be used to launch such attacks [24, 30, 57, 69]. In an attempt to mitigate JavaScript-based side-channel leakage, browser vendors have mainly focused on restricting the ability of an attacker to precisely measure time [15, 16, 84].

Side-channel attackers, in turn, attempt to get around these restrictions by creating makeshift timers with varying accuracies through the exploitation of other browser APIs, such as message passing or multithreading [42, 66, 72]. More recently, Schwarz et al. [67] presented Chrome Zero, a Chrome extension that protects against JavaScript-based side-channels by blocking or restricting parts of the JavaScript API commonly used by side channel attackers, based on a user-selected protection policy. Going even further, DeterFox [14] aims to eliminate side-channel attacks by ensuring completely deterministic JavaScript execution, and NoScript [51] prevents JavaScript-based attacks by completely disabling JavaScript.

A common trend in these approaches is that they are symptomatic and fail to address the root cause of the leakage, namely, the sharing of microarchitectural resources. Instead, most approaches attempt to prevent leakage by modifying browser behavior, striking different balances between security and usability. Thus, we ask the following question.

What are the minimal features required for mounting microarchitectural side-channel attacks in browsers? Can attacks be mounted in highly-restricted browser environments, despite security-orientated API refinements?

Besides being influenced by defenses, microarchitectural attacks are also affected by an increased hardware diversification in consumer devices. While the market for high-end processors used to be dominated by Intel, the past few years have seen an increase in popularity of other alternatives, such as AMD's Zen architecture, Samsung's Exynos, and the recently launched Apple M1 cores.

Most microarchitectural attack techniques, however, are inherently dependent on the specifics of the underlying CPU hardware, and are typically demonstrated on Intel-based machines. While microarchitectural attacks on non-Intel hardware do exist [46, 85], these are also far from universal, and

Countermeasure	Chrome Zero Policy Level	Can Be Bypassed?	Technique	External Requirements
None	None	✓	Cache Contention [24, 57, 69]	None
Reduced timer resolution	Medium	✓	Sweep Counting [69]	None
No timers, no threads	Paranoid	✓	DNS Racing	Non-Cooperating DNS server
No timers, threads, or arrays	—	✓	String and Sock	Cooperating WebSockets server
JavaScript completely blocked	—	✓	CSS Prime+Probe	Cooperating DNS server

Table 1: Summary of results: Prime+Probe Attacks can be Mounted Despite Strict Countermeasures

are also highly tailored to their respective hardware platforms. Thus, given the ever increasing microarchitectural diversification, we ask the following secondary question.

Can microarchitectural side-channel attacks become architecturally-agnostic? In particular, are there universal side channel attacks that can be mounted effectively across diverse architectures, without requiring hardware-dependent modifications?

1.1 Our Contribution

Tackling the first set of questions, in this paper we show that side channel attacks can be mounted in highly restricted browser environments, despite side-channel hardening of large portions of JavaScript's timing and memory APIs. Moreover, we show that even if JavaScript is *completely disabled*, side-channel attacks are still possible, albeit with a lower accuracy. We thus argue that completely preventing side channels in today's browsers is nearly impossible, with leakage prevention requiring more drastic design changes.

Next, tackling the second set of questions, we introduce architecturally-agnostic side channel techniques, that can operate on highly diverse architectures from different vendors. Empirically evaluating this claim, we show side channel leakage from browser environments running on AMD, Apple, ARM and Intel architectures with virtually no hardware-specific modifications. Notably, to the best of our knowledge, this is the first side-channel attack on Apple's M1 CPU.

Reducing Side Channel Requirements. We focus our investigation on website fingerprinting attacks [34]. In these attacks, an adversary attempts to breach the privacy of the victim by finding out the websites that the victim visits. While initially these attacks relied on network traffic analysis, several past works demonstrated that an attacker-controlled website running on the victim machine can determine the identity of other websites the victim visits [6, 39, 53, 57, 74].

To identify the set of JavaScript features required for cache attacks, we build on the work of [69]. We start from their website fingerprinting attacks and design a sequence of new attacks, each requiring progressively less JavaScript features. Our process of progressively reducing JavaScript features culminates in CSS Prime+Probe, which is a microarchitectural attack implemented solely in CSS and HTML, yet is capable of achieving a high accuracy even when JavaScript is completely disabled. To the best of our knowledge, this is the first microarchitectural attack with such minimal requirements.

Architecturally-Agnostic Side Channel Attacks. Next, we tackle the challenge of mounting side channel attacks across a large variety of computing architectures. We show that the reduced requirements of our techniques essentially make them architecturally-agnostic, allowing them to run on highly diverse architectures with little adaptation. Empirically demonstrating this, we evaluate our attacks on AMD's Ryzen, Samsung's Exynos and Apple's M1 architectures. Ironically, we show that our attacks are sometimes more effective on these novel CPUs by Apple and Samsung compared to their well-explored Intel counterparts, presumably due to their simpler cache replacement policies.

Evaluating Existing Side Channel Protections. Having reduced the requirements for mounting side channel attacks in browser contexts, we tackle the question of evaluating the security guarantees offered by existing API hardening techniques. To that aim, we deploy Chrome Zero [67] and measure the attack accuracy in the presence of multiple security policies. We show that while disabling or modifying JavaScript features does attenuate published attacks, it does little to block attacks that do not require the disabled features.

As a secondary contribution, we find that there are several gaps in the protection offered by Chrome Zero, and that fixing those adversely affects Chrome Zero's usability and performance. This raises questions on the applicability of the approach suggested in [67] for protecting browsers.

Attacking Hardened Browsers. Having shown the efficacy of our techniques in both Chrome and Chrome Zero environments, we also evaluate our attacks on several popular security-oriented browsers, such as the Tor Browser [71] and DeterFox [14]. Here, we show that attacks are still possible, albeit at lower accuracy levels.

Summary of Contribution. In summary, in this paper we make the following contributions:

- We design three cache-based side-channel attacks on browsers, under progressively more restrictive assumptions. In particular, we demonstrate the first side-channel attack in a browser that does not rely on JavaScript or any other

mobile code (Section 3).
- We empirically demonstrate architecturally-agnostic side channel attacks, showing the first techniques that can handle diverse architectures with little adaptation (Section 3.5).
- We re-evaluate the JavaScript API-hardening approach taken by Chrome Zero, demonstrating significant limitations that affect security, usability, and performance (Section 5).
- We evaluate our attacks in multiple scenarios, including in the restrictive environments of the Tor Browser and DeterFox (Section 6).

1.2 Responsible Disclosure

Following the practice of responsible disclosure, we have shared a draft of this paper with the product security teams of Intel, AMD, Apple, Chrome and Mozilla prior to publication.

2 Background

2.1 Microarchitectural Attacks

To improve performance, modern processors typically exploit the locality principle, which notes the tendency of software to reuse the same set of resources within a short period of time. Utilizing this, the processor maintains state that describes past program behavior, and uses it for predicting future behavior.

Microarchitectural Side Channels. The shared use of a processor, therefore, creates the opportunity for information leakage between programs or security domains [22]. Leakage could be via shared state [3, 32, 44, 80] or via contention on either the limited state storage space [27, 49, 58, 60] or the bandwidth of microarchitectural components [2, 10, 82]. Exploiting this leakage, multiple side-channel attacks have been presented, extracting cryptographic keys [2, 10, 11, 25, 32, 49, 58, 60, 65, 80, 82], monitoring user behavior [29, 33, 57, 64, 69], and extracting other secret information [7, 36, 79].

Side-channel attacks were shown to allow leaking between processes [32, 49, 58, 60, 80], web browser tabs [24, 57, 69], virtual machines [37, 49, 80, 86], and other security boundaries [7, 18, 36, 44]. In this work we are mostly interested in the two attack techniques that target the limited storage in caching elements, mainly data caches.

Prime+Probe. The Prime+Probe attack [49, 58, 60] exploits the set-associative structure in modern caches. The attacker first creates an *eviction set*, which consists of multiple memory locations that map to a single cache set. The attacker then *primes* the cache by accessing the locations in the eviction set, filling the cache set with their contents. Finally, the attacker *probes* the cache by measuring the access time to the eviction set. A long access time indicates that the victim has accessed memory locations that map to the same cache set, evicting part of the attacker's data, and therefore teaches the attacker about the victim's activity.

Cache Occupancy. In the cache occupancy attack [54, 69], the attacker repeatedly accesses a cache-sized buffer while measuring the access time. Because the buffer consumes the entire cache, the access time to the buffer correlates with the victim's memory activity. The cache occupancy attack is simpler than Prime+Probe, and provides the attacker with less detailed spatial and temporal information. It is also less sensitive to the clock resolution [69]. *Sweep counting* is a variant of the cache occupancy attack, in which the adversary counts the number of times that the buffer can be accessed between two clock ticks. The main advantage of this technique is that it can work with even lower-resolution clocks.

2.2 Defenses

The root cause of microarchitectural side-channels is the sharing of microarchitectural components across code executing in different protection domains. Hence, partitioning the state, either spatially or temporally, can be effective in preventing attacks [23]. Partitioning can be done in hardware [19, 77] or by the operating system [40, 45, 50, 68].

Fuzzing or reducing the resolution of the clock are often suggested as a countermeasure [16, 35, 73, 84]. However, these approaches are less effective against the cache occupancy attack, as it does not require high-resolution timers. Furthermore, these approaches only introduce uncorrelated noise to the channel and do not prevent leakage [17].

Randomizing the cache architecture is another commonly suggested countermeasure [61, 77, 78]. These often aim to prevent eviction set creation. However, they are less effective against the cache occupancy attack, both because the attack does not require eviction sets and because these techniques do not change the overall cache pressure.

2.3 The JavaScript Types and Inheritance

JavaScript Typing. JavaScript is an object oriented language where every value is an object, excluding several basic primitive types. For object typing, JavaScript mostly uses "duck typing", where an object is considered to have a required type as soon as it has the expected methods or properties. JavaScript deviates from this model for some built-in types, such as `TypedArrays`, which are arrays of primitive types. While JavaScript code mostly uses these built-in types equivalently to objects, the JavaScript engine itself provides certain APIs that match the arguments against the required built-in types, raising exceptions if they mismatch.

JavaScript Inheritance. JavaScript uses a prototypal inheritance model, where each object can have a single prototype object. When searching for a property of an object, JavaScript first checks the object itself. If the property is not found on in the object, JavaScript proceeds to check its prototype, until it either finds the property or reaches an object that has no prototype. The list of prototypes used in this search is called the

object's prototype chain. Finally, when JavaScript modifies an object property, the prototype chain is not consulted. Instead, JavaScript sets the property on the object itself, creating it if it does not already exist.

2.4 Virtual Machine Layering

Virtual machine layering [43] is a low overhead technique for implementing function call interception. To intercept calls to a particular function, the function is overwritten with a new function, in effect intercepting calls to the original function.

To partially override the behavior of the original function, a reference to the original function is stored, and the desired behavior is delegated to it if needed. To prevent external access to the original intercepted function, a JavaScript closure is used to store this reference. JavaScript closures create new variable scopes, preventing code outside the closure from accessing references stored within the closure.

Virtual machine layering offers a significant advantage over other techniques for guaranteeing that all calls to a given JavaScript function are intercepted. This is because virtual machine layering changes the definition of the function directly, automatically supporting the interception of function calls from code generated at runtime.

3 Overcoming Browser-based Defenses

In this section we present several novel browser-based side-channel techniques that are effective against increasing levels of browser defenses. More specifically, we present a series of attacks that progressively require less JavaScript features, culminating in CSS Prime+Probe– an attack that does not use JavaScript at all and can work when JavaScript is completely disabled. To the best of our knowledge, this is the first side-channel attack implemented solely with HTML and CSS, without the need of JavaScript.

We evaluate the effectiveness of our techniques via *website fingerprinting* attacks in the Chrome browser, which aim to recover pages currently open on the target's machine. Beyond demonstrating accurate fingerprinting levels against the Chrome browser, we show that our attacks are highly portable, and are effective across several different micro-architectures: Intel x86, AMD Ryzen , Samsung Exynos 2100 (ARM), and finally Apple M1.

3.1 Methodology and Experimental Setup

We follow the methodology of Shusterman et al. [69], where we collect *memorygrams*, or traces of cache use over the web site load time. We use these traces to train a deep neural network model, which is then used to identify web sites based on the corresponding memorygrams. Similarly to [69], we measure cache activity using both the cache occupancy and sweep counting methods (described below). Both of these methods measures the overall level of cache contention, obviating the need to construct eviction sets. Finally, we adapt both techniques to progressively more restrictive environments. The specific assumptions on attackers' capabilities appear in the respective sections (Sections 3.2 to 3.4).

The Cache Occupancy Channel. To measure the web page's cache activity, we follow past works [54, 69] and use the cache occupancy channel. Specifically, we allocate an LLC-sized buffer and measure the time to access the entire buffer. The victim's access to memory evicts the contents of our buffer from the cache, introducing delays for our access. Thus, the time to access our buffer is roughly proportional to the number of cache lines that the victim uses.

Compared with the Prime+Probe attack, the cache occupancy channel does not provide any spatial information, meaning that the attacker does not learn any information about the addresses accessed by the victim. Thus, it is less appropriate for detailed cryptanalytic attacks which need to track the victim at the resolution of a single cache set. However,the cache occupancy attack is simpler than Prime+Probe and in particular avoids the need to construct eviction sets. It also requires less accurate temporal information, on the order of milliseconds instead of nanoseconds. Thus, cache occupancy attacks are better suited to restricted environments, such as those considered in this section.

Sweep Counting. Sweep counting [69] is a variant of the basic cache occupancy attack, with reduced temporal resolution. Here, rather then timing the traversal of a cache-sized buffer, the attacker counts the number of sweeps across the buffer than fit within a time unit. While providing even less accuracy than cache occupancy, sweep counting remains effective when used with low-resolution timing sources (e.g., hundreds of milliseconds). Just like the cache occupancy attack, sweep counting does not provide any spatial resolution.

Closed World Evaluation. Using the channels we describe above, we collect memorygrams of visits to the Alexa Top 100 websites. We visit each site 100 times, each time collecting a memorygram that spans 30 seconds. We then evaluate the accuracy of our techniques in the *closed-world model*, where an adversary knows the list of 100 websites and attempts to guess which one is visited. Here, the base accuracy rate of a random guess is 1%, with any higher accuracy indicating the presence of side-channel leakage in the collected traces.

Evaluated Architectures. We demonstrate in the attacks described in this section on several different architectures made by multiple hardware vendors. For Intel, we use several machines featuring an Intel Core i5-3470 CPU that has a 6 MiB last-level cache and 20 GiB memory. The machines are running Windows 10 with Chrome version 78, and are connected via Ethernet to a university network. Next, for AMD, we used six machines equipped with an AMD Ryzen 9 3900X 12-Core Processor, which has a 4x16 MiB last-level cache and 64 GiB memory. These machines were running Ubuntu 20.04 server with Chrome version 88.0, and were connected

via Ethernet to a cloud provider network. For our ARM evaluation we used five Samsung Galaxy S21 5G mobile phones (SM-G991B), featuring an ARM-based Exynos 2100 CPU with an 8 MiB last-level cache and 8 GiB memory. These phones were running Android 11 with Chrome 88 and were connected via Wi-Fi to a University network. Finally, for our evaluation on Apple, we used four Apple Mac Mini machines equipped with an Apple M1 CPU with a 12 MiB last-level cache for performance cores and 4 MiB for efficiency cores. The machines were equipped with 16 GiB memory and were running MacOS Big Sur version 11.1 together with Chrome 88.0 for arm64. These machines were connected via Ethernet to a University network.

Machine Learning Methodology. As a classifier we use a deep neural network model, with 10-fold cross validation. See Appendix A for details. Following previous works [12, 55], we report both the most likely prediction of the classifier and the top 5 predictions, noting that the base accuracy for the top 5 results is 5% for the closed-world scenarios, and 34% for the open world. The collected data volume of all the experiments is 27 GiB consisting of 40 datasets, where each dataset takes about one week to collect, and each classifier takes on average 30 minutes to train on a cluster of Nvidia GTX1080 and GTX2080 GPUs.

3.2 DNS Racing

For our first attack, DNS Racing, we assume a hypothetical JavaScript engine that does not provide any timer, neither through an explicit interface nor via repurposing JavaScript features such as multithreading [42, 66].

DNS-based Time Measurement. Ogen et al. [56] observe that browsers behave very predictably when attempting to load a resource from a non-existent domain, waiting for exactly one network round-trip before returning an error. Thus, it is possible to create an external timer by setting the `onerror` handler on an image whose URL points to a non-existent domain. We evaluate this timer with a local DNS server and with a remote Cloudflare DNS server, using both Ethernet and Wi-Fi connections. The results, depicted in Figure 1, show that all the timers are fairly stable, with little jitter.

For an Ethernet connection to a local DNS server, the timer resolution is about 2 ms, which Shusterman et al. [69] report is high enough for the basic cache occupancy channel. A local server over Wi-Fi gives a resolution of about 9 ms, and the Cloudflare server provides a resolution of roughly 70 ms, for both Ethernet and Wi-Fi. While these resolutions are unlikely to be suitable for the basic cache occupancy attack, Shusterman et al. [69] show that sweep counting works well with the 100 ms timer of the Tor Browser.

Exploiting DNS for Cache Attacks. Figure 2a shows how to use the DNS response as a timer. As illustrated in the figure, the attacker first sets the `src` attribute of an image to a non-existent domain, in causing the operating system to access a

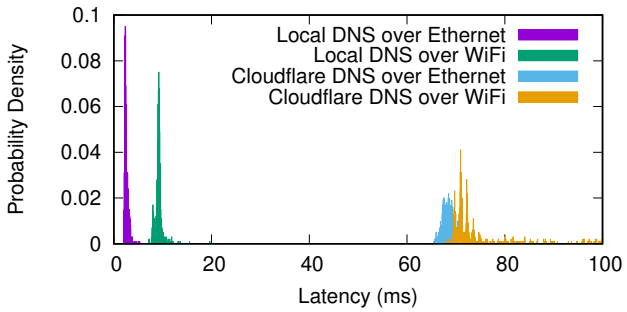


Figure 1: Measured response latencies when loading an image from a non-existent domain (local server).

remote DNS server for address resolution. The attacker then starts the cache probe operation, creating a race between the probe and the asynchronous report of the DNS error. When the asynchronous error handling function is called after name resolution fails, the attacker can determine whether the cache probing operation was faster or slower than the network round-trip time. Alternatively, when the DNS round-trip time is large, the attacker can repeat the probe step, counting the number of probes before the DNS error is reported. We note that the attack generates a large number of DNS requests. Such anomalous traffic may be detected by intrusion detection systems and blocked by the firewall.

3.3 String and Sock

Another commonality feature of most microarchitectural attacks in browsers, including our DNS racing attack, is the use of arrays [24, 28, 47]. Consequently, the use of arrays is often assumed essential for performing cache attacks in browsers and suggested countermeasures aim for hardening arrays against side channels, while maintaining their functionality [67]. To refute this assumption, in this section we investigate a weaker attack model, in which the attacker cannot use JavaScript arrays and similar data structures.

Exploiting Strings. Instead of using JavaScript arrays, our *String and Sock* attack uses operations on long HTML strings. Specifically, we initialize a very long string variable covering the entire cache. Then, to perform a cache contention measurement, we use the standard JavaScript `indexOf()` function to search for a short substring in this long text. We make sure that the substring we search for does not appear within the long string, thus ensuring that the search scans all of the long string. Because the length of the long string is the same as the size of the LLC, the scan effectively probes the cache without using any JavaScript array object. To measure the duration of this probe operation, we take advantage of an external WebSockets [21] server controlled by the attacker.

Socket-Based Time Measurement. Figure 2b shows how the String and Sock method operates. The attacker first sends a short packet to a cooperating WebSockets server. Next, the

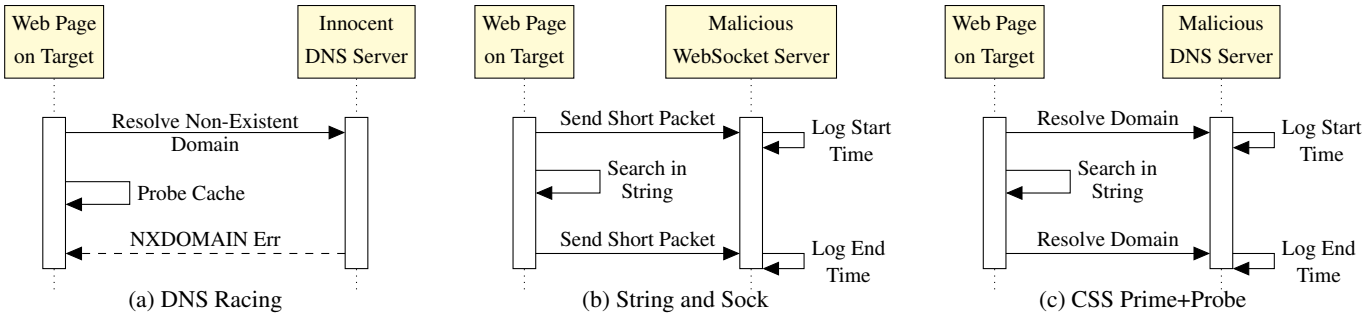


Figure 2: Interaction diagrams for attacks.

attacker performs a string search operation which is known to fail. As this search scans the entire string before failing, it has the side effect of probing the entire LLC cache. Finally, the attacker sends a second short packet to the cooperating WebSockets server. The server calculates the timing difference between the first and second packets, arriving at an estimate of the time taken to probe the cache.

String and Sock in Chrome. We find that Chrome allocates three bytes for each character. As we would like our string to occupy the machines entire last level cache, we allocate different string lengths for each architecture considered in this paper. In particular, we use 2 MiB strings for our Intel machines that feature a 6 MiB LLCs, 3 MiB strings for our AMD machines (4x16 MiB LLCs), 1.5 MiB strings for our Samsung phones (8 MiB LLC), and 2 MiB strings for our Apple machines (12 MiB LLCs on performance cores). We also note that Chrome caches results of recent searches. To bypass this caching, for each search we generate a small fresh sequence of emojis and search for it. With the long string consisting only of ASCII characters, it is guaranteed not to contain any emojis.

3.4 CSS Prime+Probe

Our final attack, *CSS Prime+Probe* targets an even more restricted setting, in which the browser does not support JavaScript or any other scripting language, for example due to the NoScript extension [51]. CSS Prime+Probe only uses plain HTML and Cascading Style Sheets (CSS) to perform a cache occupancy attack, without using JavaScript at all.

CSS Prime+Probe Overview. At a high level, CSS Prime+Probe builds on the String-and-Sock approach, and like it relies on string search for cache contention and an attacker-controlled server for timing, see Figure 2c. Here, the attacker first includes in the CSS an element from an attacker-controlled domain, forcing DNS resolution. The malicious DNS server logs the time of the incoming DNS request. The attacker then designs an HTML page that evokes a string search from CSS, effectively probing the cache. This string search is followed by a request for a CSS element that requires DNS resolution from the malicious server. Finally, the time difference between consecutive DNS requests corresponds to the time it takes to perform the string search, which as described above is a proxy for cache contention.

CSS Prime+Probe Implementation. Figure 3 shows a code snippet implementing CSS Prime+Probe, using CSS *Attribute Selectors* to perform the attack. Specifically, Line 9 defines a `div` with a very long class name (two million characters). This `div` contains a large number of other `div`s, each with its own ID (Lines 10–12). The page also defines a style for each of these internal `div`s (Lines 3–5). Each of these matches the IDs of the internal and external `div`, and uses an attribute selector that searches for a substring in the external `div`. If not found, the style rule sets the background image of the element some URL at an attacker-controlled domain.

```
1  <head>
2    <style>
3      #pp:not([class*='vukghj']) #s0 {
           background-image: url("https://
           kxdfvcgx.attack.com");}
4    [...]
5      #pp:not([class*='vatwjo']) #s9999 {
           background-image: url("https://
           bwpqxunq.attack.com");}
6    </style>
7  </head>
8      <body>
9        <div id="pp" class="AA...A">
10         <div id="s0">X</div>
11 [...]
12         <div id="s9999">X</div>
13       </div>
14     </body>
```

Figure 3: Simplified version of CSS-based Prime+Probe.

When rendering the page, the browser first tries to render the first internal `div`. For that, it performs a long search in the class name, effectively probing the cache occupancy. Having not found the substring, it sets the background image of the `div`, resulting in sending a request to the attacker's DNS server. The browser then proceeds to the next internal `div`. As a result of rendering this page, the browser sends to the attacker a sequence of DNS requests, whose timing depends on the cache contention.

Attack Technique	Top-1 Accuracy (%)				Top-5 Accuracy (%)			
	Intel i5-3470	AMD Ryzen 9 3900X	Apple M1	Samsung Exynos 2100	Intel i5-3470	AMD Ryzen 9 3900X	Apple M1	Samsung Exynos 2100
Cache Occupancy	87.5	69.1	89.7	84.5	97.0	91.4	97.8	95.3
Sweep Counting	45.8	54.9	90.5	69.7	74.3	82.9	98.1	91.5
DNS Racing	50.8	5.4	48.2	5.8	78.5	16.3	83.5	37.1
String and Sock	72.0	53.9	90.6	60.2	90.6	85.5	97.9	85.5
CSS Prime+Probe	50.1	—	15.7	—	78.6	—	32.6	—

Table 2: Closed-world accuracy (percent) across different microarchitectures.

Attack Technique	Intel i5-3470	AMD Ryzen 9 3900X	Apple M1	Samsung Exynos 2100
Cache Occupancy	2.9 ms	6.0 ms	6.3 ms	4.0 ms
Sweep Counting	100.0 ms	100.0 ms	100.0 ms	100.0 ms
DNS Racing	20.3 ms	1.8 ms	7.2 ms	2.9 ms
String and Sock	1.5 ms	2.9 ms	2.6 ms	2.5 ms
CSS Prime+Probe	0.3 ms	6.7 ms	0.3 ms	33.8 ms

Table 3: Temporal accuracy of attack techniques across different microarchitectures.

3.5 Empirical Results

We now present the classification results of the attacks described in this section across different CPU architectures. Table 2 summarizes the accuracy of the most likely prediction of the classifier (Top-1), as well as the likelihood that the correct answer is one of the top 5 results (Top-5). Finally, Table 3 shows the temporal resolution of each measurement method, calculated as the time it takes to capture the entire trace, divided by the number of points in the trace.

Cache Occupancy. This method uses JavaScript code both to iterate over the eviction buffer, and to measure time. The JavaScript code goes iterates over the buffer using the technique of Osvik et al. [58] to avoid triggering the prefetcher, and is written to prevent speculative reordering from triggering the timing measurement before the eviction is completed. As can be seen from the results, this approach provides good accuracy on all of the targets we evaluated, obtaining a top-5 accuracy of over 90% across all platforms.

Sweep Counting. This method is designed for situations with lower clock resolution, but still uses JavaScript both for cache eviction and for timing measurement. As the results show, this added limitation translates to a loss in accuracy for most targets, with the Apple M1 target the least affected by the reduced timer resolution.

DNS Racing. This method uses JavaScript for cache eviction, but switches to the network for timing measurements. This added limitation translates to a loss in accuracy for most targets, largely due to the added jitter of the network. The targets most severely affected by the added jitter were the ARM-based mobile phones, which were connected to the network using a wireless link, and the AMD devices, which were located in a third-party data center whose network conditions were beyond our direct control. We hypothesize that these networking circumstances led to jitter related to DNS responses, causing the severe loss of accuracy for these targets.

String and Sock. This is the first method which repurposes the browser's string-handling code for cache eviction. Unlike the adversary-controlled code used for mounting the cache occupancy attack described earlier, this third-party code naturally makes no attempt to trick the processor's cache management heuristics, and, as such, we expected it to have lower performance than the JavaScript-based code.

As we see, this was indeed the case for the Intel, AMD and Samsung targets. The Apple M1 target, on the other hand, did not encounter a loss in accuracy. It seems that, on this target, naïvely accessing a large block of memory is an efficient way to evict the cache, and more advanced approaches for tricking the processor's prefetcher are not necessary.

CSS Prime+Probe. As CSS Prime+Probe requires no JavaScript, we test this attack in the presence of the NoScript [51] extension, applying the countermeasure only to our attacker website. As our attack does not use JavaScript at all, NoScript does nothing to prevent it. The accuracy we obtained using this attack was comparable to the one obtained by the String and Sock attack, showing that there is no need for JavaScript, or any other mobile code, to mount a successful side-channel attack.

When running this attack on the Intel target, the accuracy is similar to DNS racing, which uses JavaScript for cache evictions. On the M1 target, there was still a significant amount of data leaked by the attack, but the accuracy was less than the DNS racing attack. On the ARM and AMD targets, we are unable at the present to extract any meaningful data using this method. As our CSS Prime+Probe also relies on DNS packets, we conjecture that this is due to the network conditions of the devices under test, or due to particular aspects of the micro-architecture of these devices which make cache eviction less reliable.

Architectural Agnosticism. As the results show, we were able to mount our side-channel attack across a large variety of diverse computing architectures. In particular, the Intel,

AMD, ARM and Apple target architectures all incorporate different design decisions concerning different cache sizes, cache coherency protocols and cache replacement policies, as well as related CPU front-end features such as the prefetcher. The reduced requirements of our attack made it immediately applicable to all of these targets, with little to no tuning of the attack's parameters, and without the need of per-device microarchitectural reverse engineering.

Attacking Apple's M1 Architecture. To the best of our knowledge, this is the first side-channel attack on Apple's M1 CPU. The memory and cache subsystem of this new architecture have never been studied in detail, leading one to hope for a "grace period" where attackers will find this target difficult to conquer. As this work shows, the novelty and obscurity of this new target do little to protect it from side-channel attacks. The M1 processor is rumored to toggle between two completely different memory ordering mechanisms, based on the program it is executing. Another noteworthy outcome from the M1 evaluation is that both the native arm64 binary of Chrome, as well as the standard MacOS Intel x64 Chrome binary running under emulation, were vulnerable to the attacks we described here.

Finally, observing Table 2, it can be seen that our attacks are, somewhat ironically, more effective on M1 architecture, than they are on other architectures, including the relatively well studied Intel architecture. Intel x86 CPUs are known to have advanced cache replacement and prefetcher policies, which are have been shown in other works to anticipate and mitigate the effect of large memory workloads on cache performance [8, 62, 76]. We hypothesize that the M1 architecture makes use of less advanced cache heuristics, and that, as a result, the simplistic memory sweeps our attack performs are more capable of flushing the entire cache on these devices than they are on the Intel architecture. This in turn results in a higher signal-to-noise ratio for the attack on these newer targets, and therefore in a higher overall accuracy.

4 Attack Scenarios

We now turn our focus to a deeper investigation of the two new attacks we present, String and Sock and CSS Prime+Probe, on the Intel targets. Table 4 provides a summary of the results discussed in this section.

Attack Scenario	String and Sock	CSS Prime+Probe
Closed World	74.5±1.6	48.8±1.6
Open World	80.2±1.1	60.9±1.4
Artificial Jitter	40.6±1.9	26.6±1.4
Tor Browser	19.5±8.7	—
DeterFox	—	65.7±1.2

Table 4: Attack accuracy (%) with 95% confidence intervals.

4.1 Closed World Evaluation on Newer Intel Architectures

We begin by reproducing the closed world methodology and the results of Section 3 albeit on a newer Intel processor. Specifically, we perform the experiments on an Apple Macbook Pro featuring an Intel Core i5-7267 CPU with a 4 MiB last-level cache, and 16 GiB memory, running macOS 10.15 and Chrome version 81. Despite the microarchitectural changes across 4 CPU generations and the different cache size, the results are very similar to those achieved on the older i5-3470 (72.0±1.3% for String and Sock and 50.1±2.3 for CSS Prime+Probe), with the difference being well inside the statistical confidence levels. We thus argue that our results transfer across a verity of Intel architectures.

4.2 Open-World Evaluation

A common criticism of closed-world evaluations is that the attacker is assumed to know the complete set of websites the victim might visit, allowing the attacker to prepare and train classifiers for these websites [38]. For a more realistic scenario, we follow the methodology proposed by Panchenko et al. [59] and perform an open-world evaluation, collecting 5000 traces of different websites used in [63], in addition to the Alexa Top 100 websites collected in the closed-world setting. We use the same data collection setting as for the closed-world collection. (See Section 4.1.)

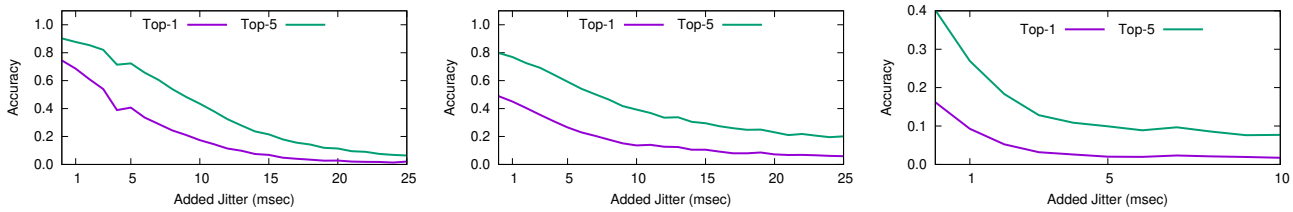
Here, the attacker's goal in this setting is to first detect if the victim visits one of the Alexa Top 100 sites, and secondly to identify the website if it is indeed in the list. We note that in this case, a naive classifier can always claim that the site is not one of the Alexa Top 100, achieving a base rate of 30%, resulting in slightly higher accuracy scores for any classifier.

In this open-world setting, the String and Sock and CSS Prime+Probe attacks obtain accuracy results of 80% and 61%, respectively. The data in this setting is unbalanced – there are more traces from "other" web sites than from each of the Alexa Top 100 sites. For such data, the F1 score may be more representative than accuracy. The F1 scores are 67% and 45%, for String and Sock and CSS Prime+Probe, respectively. These are similar to those of the closed-world settings (70% and 48%). We can therefore conclude that our attacks are as effective in the open-world as in the closed-world setting.

4.3 Robustness to Jitter

As DNS racing, String and Sock, and CSS Prime+Probe use an external server for time measurement, these techniques are inherently sensitive to jitter naturally present on the network between the victim and the web server.

Measuring Network Jitter. We measure the network jitter in two scenarios. First, we perform a local measurement, where the target and an attacker-controlled WebSockets server



(a) String and Sock (b) CSS Prime+Probe (c) DNS Racing (note different scale)

Figure 4: Attack classifiers performance with additional jitter.

are located on the same institutional network at Ben Gurion University, Israel. Next, we also perform an inter-continental measurement, where the attacker is located in Israel, while the server is located in the United States (University of Michigan). Figure 5 shows the distribution of the jitter observed while sending 100 packets per second for 30 seconds to the WebSockets servers. We find that the jitter in the local network has a standard deviation of 0.17 ms, whereas the jitter to the cross-continent server has standard deviation of 0.78 ms.

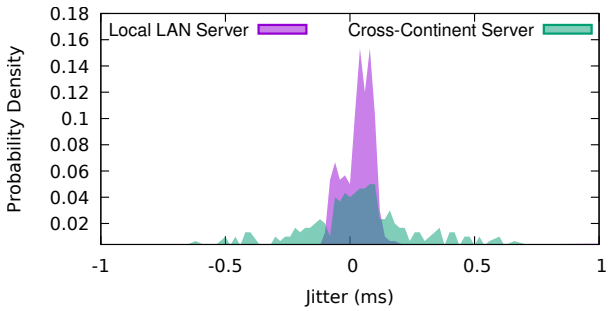


Figure 5: Measured Jitter of the WebSockets server response.

Evaluating Robustness to Jitter. Having established the typical jitter between the target and the external server, we now evaluate the robustness of our techniques to various levels of jitter. To that aim, we artificially inject different amounts of jitter to the closed-world dataset of Section 4.1. The jitter is injected by adding random noise to the timing of the monitored events. This noise is selected at random from a normal distribution with a mean zero and a standard deviation that varies from 1 to 25 milliseconds, with higher standard deviation corresponding to larger jitter.

As Figure 4 shows, both the String and Sock and the CSS Prime+Probe attacks still retain most of their accuracy even if the jitter is an order of magnitude larger than the ones we measured on a real network. We finally note that the DNS Racing attack is more sensitive to added jitter, as it relies on a binary race condition to determine timing.

5 Analysis of an API-based Defense

Having established the efficacy of our techniques on various microarchitectures, in this section we evaluate our attacks in the presence of increasing levels of browser hardening.

To that aim, we make use of Chrome Zero [67], a Chrome extension that supports per-website restrictions on JavaScript browser API features. We begin by presenting an overview of Chrome Zero's JavaScript implementation and security objectives, focusing on a subset of Chrome Zero's features which are relevant to this work. We next describe how we modified Chrome Zero to offer more comprehensive protection, at the cost of usability and performance. Finally, we show that even with these modifications, Chrome Zero is unable to offer side channel protections against the techniques presented in this paper. Unless stated otherwise, we use the current version at Chrome Zero's Git repository.*

5.1 Chrome Zero Overview

Chrome Zero implements a list-based access control policy, which dictates actions to be taken when a website invokes a JavaScript function or accesses an object property. When an access is detected, Chrome Zero either allows the access, modifies it, or completely blocks the access based on the policy chosen for the particular website.† Chrome Zero also supports the option of asking the user about the action to take.

Default Policies. Chrome Zero offers five preset protection policies for the user to choose from: None, Low, Medium, High, and Paranoid. ‡ As it progresses through protection policy levels, Chrome Zero makes increasingly severe restrictions on JavaScript capabilities and resources, including blocking them altogether. Table 5 summarizes which capabilities and resources are available at each protection level.

Performance. Schwarz et al. [67] claim that Chrome Zero blocks all of the building blocks required for successful side-channel attacks, including high resolution timers, arrays and access to hardware sensors. Moreover, they claim that Chrome Zero prevents many known CVEs and 50 percent of zero-day exploits published since chrome 49. Finally, Schwarz et al. [67] benchmark Chrome Zero's performance and perform a

*https://github.com/IAIK/ChromeZero commit fee8adc6c8fce9dd1ab62d7ff8f0697b44a188c1

†Chrome Zero currently only supports a global protection policy that can be changed but applies to all websites.

‡The Chrome Zero extension uses the name "Tin Foil Hat" for Paranoid. We stick to the naming in Schwarz et al. [67].

Policy Level	Low	Medium	High	Paranoid
Memory Addresses	Buffer ASLR	Array preloading	Non-deterministic array	Array index randomization
Timer manipulation	Ask User	Low-resolution timestamp	Fuzzy time	Disabled
Multithreading	—	Message delay	WebWorker polyfill	Disabled
Shared Array Buffer	—	Slow SharedArrayBuffer	Disabled	Disabled
Sensor API	—	Ask User	Fixed Value	Disabled

Table 5: Defense techniques used in each Chrome Zero Policy Level.

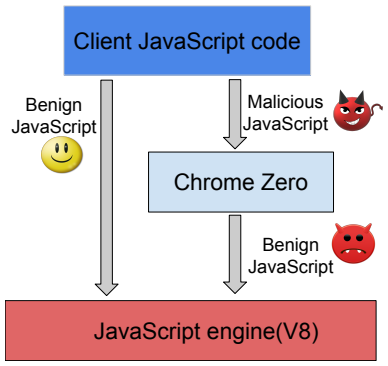


Figure 6: High-level concept of Chrome Zero

usability study. They claim that Chrome Zero has an average overhead of 1.82% at the second-highest protection level (High) and that its presence is indistinguishable to users in 24 of Alexa's Top 25 websites.

Chrome Zero's Access Control Implementation. To enforce security policies, Chrome Zero intercepts JavaScript API calls using Virtual Machine Layering. Specifically, Chrome Zero is implemented as JavaScript code that is injected into a web page when upon initialization. This injected code wraps sensitive API functions, having the wrappers implement actions specified by Chrome Zero's policy. Chrome Zero uses closures to ensure that the wrapper contains the only reference to the original API functions, thus ensuring that websites do not trivially bypass its protection (Figure 6).

Protecting Timers. Traditionally, microarchitectural side-channel attacks rely on having access to a high-resolution timer, e.g. to distinguish cache hits from cache misses. This includes attacks implemented in native code [3, 27, 29, 31, 49, 58, 60, 80, 82] as well as attacks in JavaScript running inside the browser [24, 26, 57, 66]. As a countermeasure for such attacks, Chrome's current implementation of `performance.now()` already reduces timer resolution from nanoseconds to microseconds and introduces a small amount of jitter. Although these mitigations protect against some high-resolution attacks [26, 57, 66], microsecond-accurate timers still provide sufficient resolution for other side-channel attacks from within JavaScript [28, 30, 66, 70, 72].

To block attacks that exploit microsecond-accurate timers, Chrome Zero employs two main strategies. At its Medium protection policy, Chrome Zero applies a "rounded floor" function, matching the 100 ms resolution of the Tor Browser. While this already prevents many attacks [66], higher resolution timers may still be constructed [42, 66, 72]. Thus, at higher protection levels, instead of using a simple "rounded floor" 100 ms timers, Chrome Zero follows the approach of Vattikonda et al. [73] and fuzzes the timer measurements by adding random microsecond-level noise. Finally, at its highest protection level, Chrome Zero disables timers altogether.

Arrays. Schwarz et al. [67] identify that many side-channel attacks in browsers [24, 26, 28, 30, 57, 66] require some information about memory addresses. Typically, recovering the page offset (least significant 12 of 21 bits of the address) facilitates the attacks. Using this information the attacker then analyzes the victim's behavior, deducing information about its control flow and internal data. Chrome Zero therefore applies several mitigations to JavaScript array APIs.

More specifically, Chrome Zero's second-highest protection level introduces array non-determinism, adding an access to a random element for each array access. The idea is that the random accesses themselves force page faults, impeding the use of page faults as signals for page boundaries. Schwarz et al. [67] argue that this method prevents eviction set construction [24, 30, 57, 66, 81], as it interferes with the specific sequences required to construct an eviction set, while adding noise to the timing information.

Next, Chrome Zero further deploys the buffer ASLR policy, which shifts the entire buffer by a random offset. This is achieved by intercepting the array constructors and access methods. To prevent page alignment, Chrome Zero increases the requested array size by 4 KiB, and associates a random page offset with the array. On array access, Chrome Zero adds the random offset to the requested array index, thereby shifting the access by the random offset.

Finally, to protect the offset from being discovered, Chrome Zero attempts to use the additional accesses to random elements to pre-load all the array's memory pages into the cache, thus preventing attackers from detecting page boundaries by looking for array elements which have an increased access time due to page faults.

Protecting Against Browser Exploits. While not being a primary goal of Chrome Zero, Schwarz et al. [67] argue that Chrome Zero is also capable of protecting users against some

browser exploits. To validate their claim, they reproduced 12 CVEs in the then-current Chrome JavaScript engine, and found that Chrome Zero prevents exploiting half of the CVEs. Schwarz et al. [67] attribute this protection to the modification of JavaScript objects in Chrome Zero, which breaks the CVE exploit code.

5.2 API Coverage

As stated above, Chrome Zero is essentially an interception layer, which intercepts the critical JavaScript API calls and subsequently directs them to the appropriate logic based on the current website and protection policy. Thus, to guarantee security, it is critical to ensure that malicious JavaScript code cannot access the original API or otherwise bypass the Chrome Zero protections.

Our investigation of Chrome Zero demonstrated that API coverage in Chrome Zero leaves a lot to be desired. Specifically, we have identified multiple instances of APIs that are not protected by Chrome Zero. These include:

- **Delayed Extension Initialization.** The Chrome Zero extension initializes after the browser finishes constructing the Document Object Model (DOM) for the page. Consequently, Chrome Zero does not protect JavaScript objects created before the DOM is constructed.
- **Missed Contexts.** Chrome Zero only applies its security policies in the context of the topmost page in each browser tab. It does not, however, protect code in sub-contexts of the page, including worker threads and iframes.
- **Unprotected Prototype Chains.** As we discuss in Section 2.3, properties of global objects may be inherited from their prototypes. Yet, while Chrome Zero does protect global objects, it fails to protect their prototype chains, allowing attackers to access the original JavaScript API.

Exploitation. We have exploited each of those omissions and demonstrated complete bypass of Chrome Zero protections. In most cases, such bypasses are fairly trivial. As an example we show how we exploit unprotected prototype chains.

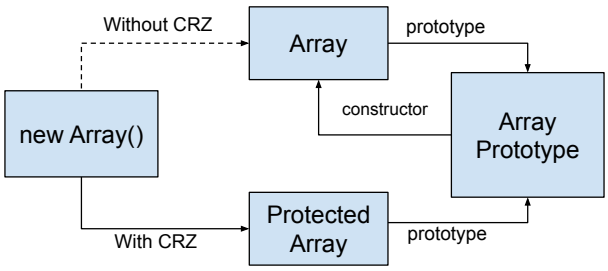


Figure 7: Object hierarchy with Chrome Zero.

Figure 7 shows the object hierarchy for `Array` with Chrome Zero (solid line) and without it (dotted line). The original unprotected `Array` class can be accessed using the `Array` constructor method of the prototype object. Figure 8 shows a bypass of Chrome Zero object protections, allowing the attacker to create original non-proxied JavaScript objects. Lines 1 and 2 show the standard ways of creating an array or getting the timer, both protected by Chrome Zero. In contrast, Lines 4 and 5 show how to use prototypes to achieve the same functionality, bypassing Chrome Zero.

```
1  let secureArray = new Array(10);
2  let secureTimer = performance.now();
3
4  let insecureArray = new
       secureArray.__proto__.constructor(10);
5  let insecureTimer =
       performance.__proto__.now.call(
       performance);
```

Figure 8: Bypassing Chrome Zero defenses using prototypes.

Evaluating Chrome Zero's CVE Protection. We also evaluate Chrome Zero's claimed protection against browser exploits. We first reproduce the results of Schwarz et al. [67] finding that Chrome Zero prevents six of the 12 exploits they experiment with. We then extend the evaluation to CVEs reported after the Chrome Zero publication and find that Chrome Zero blocks four of the 17 exploits we managed to reproduce in Chrome. We then modify the exploits that Chrome Zero blocks to use APIs that Chrome Zero fails to protect, allowing the attacks to run unhindered.

We further note that Chrome Zero only protects incidental properties of the exploits rather than addressing the underlying vulnerabilities. Specifically, we can easily modify many of the blocked exploits to avoid using features that Chrome Zero protects. For the four exploits we cannot modify to bypass Chrome Zero, we find that the cause is that the use of protected typed arrays prevents Chrome from compiling Web Assembly [75, "read the imports"]. Since the Web Assembly compiler is not invoked, the browser remains protected.

5.3 Fixing and Re-evaluating Chrome Zero

Chrome Zero's failure to protect all of the JavaScript API has implications beyond security. Unprotected objects do not affect the usability or the performance of the browser. To evaluate the impact of the approach on usability and performance, we fix Chrome Zero to improve its API coverage. Specifically, we set Chrome Zero to initialize before any other script executes and to also apply to frames. We further modify Chrome Zero to apply its interception to protected objects and all the objects in their prototype chain. We do not protect Web Workers, hence our analysis below may still understate the impact on usability and performance. We further remove bypasses of array protections that apply to some hard-coded websites. Specifically, Chrome Zero does not apply some array protections to YouTube and to Google Maps.[§]

[§]We note that without the bypass, YouTube does not play videos. We could not find any indication of this bypass in Schwarz et al. [67], which we

Finally, Schwarz et al. [67] argue that Chrome Zero offers no noticeable impact on user experience while only having a negligible performance cost. We test this claim with and without our security fixes.

Experimental Setup. We use a ThinkPad P50 featuring an Intel Core i7-6820HQ CPU, with 16 GiB of memory, running Ubuntu version 18.04, with a Chrome 80 browser without any extensions. We evaluate usability on Alexa's Top 25 USA websites, checking for discernible differences in behavior.

Usability Results. We first replicate the results of Schwarz et al. [67], finding that an unmodified Chrome Zero has no discernible impact on the usability of websites. However, after fixing the issues identified in Section 5, we observe a significant impact on the usability of websites. Even when setting Chrome Zero to the Low policy, less than half of the websites function without noticeable problems. At the a higher protection level, High, only the websites for Wikipedia and eBay function properly.

Strict Type Checking. Investigating the difference in website usability between the original and modified Chrome Zero, we find that forcing Chrome Zero to apply its policies before document loading results in type mismatch exceptions while loading many JavaScript-enabled web sites.

The cause of the issue is that as part of applying its policies, Chrome Zero replaces any JavaScript object it protects with a proxy that masquerades as the original object. Typically this does not cause any problems due to JavaScript's use of "duck typing", since replacing objects with the corresponding proxy objects is transparent to most JavaScript code, as long as the original object's properties are all supported. However, the W3C standard [20] dictates strict type checking for many internal JavaScript functions, especially for typed array objects. In this case, passing a proxy object instead of the original object results in a type mismatch exception from the browser's JavaScript engine, causing the website's loading to fail.

Unfortunately, fixing this issue turns out to be a non-trivial problem, as a significant portion of the JavaScript environment is forced to strictly type check its inputs. This goes well beyond the member functions of `TypedArrays` and includes diverse JavaScript libraries, such as, for example, the Web Crypto and Web Socket APIs.

Estimating Performance Impact. While we do not claim to know an efficient method of automatically solving this problem for the entire JavaScript API, we can efficiently solve the issue for specific functions through manual intervention, allowing us to benchmark the result. While we acknowledge that this does not produce a secure or even correct implementation, we argue that it nonetheless allows us to measure a lower-bound of the performance impact that any JavaScript zero implementation must have. To that aim, we enumerate all of the functions used by the JetStream 1.1 benchmark, and manually implement fixes for functions that perform strict type checking. We note that only the `set` and `subarray` methods for typed arrays need to be fixed, while all other parts of the JavaScript environment can remain unaltered.

Benchmarking Performance For performance benchmarks we first try to reproduce the results of Schwarz et al. [67]. We use the JetStream 1.1 benchmark to facilitate comparison with Schwarz et al. [67]. We find a slight performance impact of 1.54% when using an unmodified Chrome Zero. However, when ensuring that Chrome Zero applies its protections correctly and applying the minimum level of fixes for strict type checking we observe a performance impact of 26% in the latency benchmarks and 98% in the throughput benchmarks.

5.4 Bypassing Non-Deterministic Arrays

With the exception of speculative execution attacks [9, 13, 41, 48], most microarchitectural side-channel attacks retrieve information about memory access patterns performed by the victim. For a language such as JavaScript with no notion of pointers or addresses, most attacks exploit the contiguous nature and predictable memory layout of arrays to reveal information about the least significant 12 or 21 bits of the addresses accesses by the victim [26, 30, 57, 66].

To prevent this leakage, Chrome Zero's second-highest protection level introduces array non-determinism, performing a spurious access to a random array index whenever the script accesses an array element. Chrome Zero further deploys the buffer ASLR policy, which shifts the entire buffer by a random offset, thereby preventing the attacker from obtaining page-aligned buffers. The main idea is to use the random offset to deny the attacker from finding the array elements located on page boundaries. To protect the offset from being discovered, Chrome Zero attempts to use the additional accesses to random elements in order to pre-load all the array's memory pages into the cache, thus preventing the attacker from discovering the array elements which have an increased accesses time due to page faults.

We now show how we can reliably recover the array elements corresponding to page boundaries, despite Chrome Zero's use of buffer ASLR, non-deterministic arrays, and fuzzy timers.

Array Implementation in Chrome. Unlike their C counterparts, JavaScript arrays are quite flexible and can be extended [5], shrunk [4] and even have their type changed [52] at run-time. While the W3C standards require browsers to support the extension and shrink APIs, the implementation of these capabilities is left entirely to the browser vendors.

In Chrome's V8 JavaScript engine, whenever an array is initialized, V8 allocates the memory required for the array, along with an additional memory to support insertion of more elements in $O(1)$ amortized time. However, after the addition of enough elements, memory reallocation is eventually

find odd given the use of YouTube in the usability evaluation. The Chrome Zero source code claims that the bypass is due to a bug in Chrome, however our root cause analysis shows that YouTube fails to play videos due to the type mismatch we discuss in this section.

needed. Hence V8 allocates a new chunk of memory which is about 1.5× larger than the old one, and frees the old one after copying the array's content to the new location. The formula used by V8 to determine the size of the new memory buffer is

$$\text{new\_size} = \text{size} + \text{size} \gg 1 + 16, \quad (1)$$

where $\gg$ is a bit-wise shift-right operation.

```
1 let array = new Array();
2 let times = new Array();
3
4 for(let i=0; i<10000000; i++){
5   let start = performance.now();
6   array.push(0);
7   let delta = performance.now() - start;
8   times.push(delta);
9 }
```

Figure 9: Measuring Array.push timings

Attack Methodology. We begin by measuring the timings of `Array.push` using the code presented in Figure 9. We start with an empty array `array` (Line 1). We then append data to the end of the array using the JavaScript `Array.push` method (Line 6). On every such element addition we measure the time taken to add an element (Lines 5 and 7). While most of these additions are fast, at the point where the memory allocated for the current size of `array` is exhausted, V8 performs additional work by allocating new memory using Equation 1 and copying the old content to the newly-allocated space.

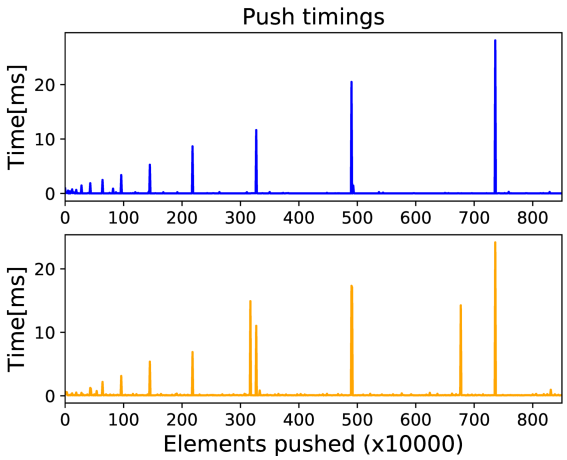


Figure 10: Push timings with native Chrome (top), and with Chrome Zero at High level (bottom).

Figure 10 shows the insertion times for elements, using both a high resolution timer (top) and Chrome Zero's fuzzy timer (bottom). As can be seen, some array insertions are slower than others. We verify that these additional time costs happened at a point where the buffer allocated by V8 to support the array `array` was exhausted, forcing V8 to allocate a new memory space using using Equation 1.

Observing Figure 10, the time required to handle the element addition at the point of buffer exhaustion increases as the size of the array grows. This is expected as more elements need to be copied by V8 as the buffer grows. However, as the number of elements added to the array is attacker-controlled, we can make `Array.push` take an arbitrary amount of time.

We exploit this property to mount an attack against Chrome Zero's Buffer ASLR policy despite Chrome Zero's attempts at reducing the resolution of JavaScript timers. More specifically, after a sufficient number of iterations of the loop in Line 4, the time taken to handle the re-allocation of `array` during the insertion of an additional element in Line 6 becomes visible despite Chrome Zero's low resolution timer. To deduce the buffer's offset generated by Chrome Zero, we apply Chrome Zero's buffer ASLR policy to Equation 1 to obtain the following equation.

$$\text{new\_size} + \mathit{offset} = (\text{size} + \mathit{offset}) + (\text{size} + \mathit{offset}) \gg 1 + 16. \quad (2)$$

Observing the spikes in Figure 10, an attacker can detect when the memory of `array` is exhausted. From that, to recover the value of *offset*, we rearrange Equation 2 as

$$\mathit{offset} = 2 \times \text{new\_size} - 3 \times \text{size} - 2 \times 16, \quad (3)$$

where size and new_size are the size's of `array` before and after resizing. Finally, to detect resizing events, an attacker can observe spikes in Figure 10. Thus, Chrome Zero's buffer ASLR policy can be defeated using two sequential resizing events and applying Equation 3 to solve for *offset*.

5.5 Attacking Chrome Zero

We now present the classification results of the attacks described in Section 3 across different Chrome Zero policies, starting with the closed-world scenario. Table 6 summarizes the accuracy of our technique, using the Intel i5-3470 setup outlines in Section 3.1.

Cache Occupancy and Sweep Counting. As we can see, for the basic cache occupancy attack, Chrome Zero policies have varying impact on the attack accuracy. Low has some impact, but the accuracy is still high. Medium almost completely blocks the attack, with the accuracy being slightly more than the base rate. Surprisingly, High is less effective than the two lower policy levels, possibly because of its simpler code design, resulting only in a slight decrease in the accuracy compared to no protection at all. For the sweep counting attack, we see that the accuracy is lower than that of the basic cache occupancy channel. However, the Medium policy no longer breaks the attack. Furthermore, while lower than that of the cache occupancy attack, the accuracy is still significantly higher than the base rate. Finally, because these attacks require Worker threads, which are blocked in Paranoid, they both fail in this policy.

Attack Technique	Temporal Resolution	Top-1 Accuracy (%) None	Low	Medium	High	Paranoid	Top-5 Accuracy (%) None	Low	Medium	High	Paranoid
Cache Occupancy	2.9 ms	87.5	71.1	2.2	81.8	N/A	97.0	87.4	6.1	96.5	N/A
Sweep Counting	100.0 ms	45.8	24.1	32.2	60.1	N/A	74.3	50.1	59.0	88.3	N/A
DNS Racing	20.3 ms	50.8	20.9	61.1	37.2	16.2	78.5	48.9	86.0	67.7	40.1
String and Sock	1.5 ms	72.0	51.3	46.2	58.4	59.9	90.6	80.0	75.9	85.3	82.8
CSS Prime+Probe	2.8 ms	(with the NoScript extension) 50.1					(with the NoScript extension) 78.6				

Table 6: Closed-world accuracy (percent) with different API restriction levels (Intel i5-3470).

DNS Racing. The DNS Racing technique achieves a moderate accuracy in the range 20% to 61%. As expected for a technique that requires neither timers nor threads, the attack also works with Paranoid policy.

String and Sock. The results with the String and Sock tend to be better than DNS Racing. In fact, the results tend to only be slightly inferior to those of the cache occupancy attack, despite not requiring timers, arrays, or threads. We further observe that because the attack uses no protected API, the various Chrome Zero policies have only a marginal effect on attack success.

CSS Prime+Probe. As mentioned in Section 3.4, our CSS Prime+Probe technique does not require JavaScript and is effective even if the attacker's website is banned from executing any JavaScript code (e.g., due to the NoScript extension [51]). In particular, Chrome Zero's focus on JavaScript does not effect our CSS Prime+Probe technique, leaving CSS Prime+Probe completely unmitigated.

Discussion. Examining the results in Table 6, we see that restricting browser APIs such as threads, timers, and array access can thwart the standard Cache Occupancy and Sweep Counting attacks, and can significantly degrade the effectiveness of the DNS Racing attack. Nevertheless, the two remaining attacks, String and Sock and CSS Prime+Probe, are not affected by this browser-based countermeasure, since they do not use any API which is receiving protection. While there is some variation in accuracy between the different protection modes for String and Sock, this is likely due to the usability and site loading side-effects related to our fortified version of Chrome Zero, and not due to any intrinsic protection offered the API limiting approach. We thus argue that preventing side channels in today's browsers using API modifications is practically impossible. Properly preventing leakage would require a more systematic approach which considers the sources of leakage, and not merely the means for measuring it.

6 Attacking Hardened Browsers

Having established the feasibility of mounting cache side channel attacks while only having limited (or no) access to the JavaScript API, in this section we proceed to demonstrate the effectiveness of our techniques on two privacy enhanced browsers: Tor [71] and DeterFox [14].

6.1 Attacking the Tor Browser

The Tor Browser [71] is a highly-modified version of Firefox, designed to offer a high level of privacy even at the cost of usability and performance. At a high level, the Tor Browser combines two elements to achieve a higher level of protection compared to other browsers. First, it hides the user's browsing habits from network adversaries by using the Tor network as an underlying transport layer. Second, it provides a highly restrictive browser configuration, designed to limit or disable convenience features that may have a security impact. In the context of side channel attacks, the Tor Browser limits the resolution of the timer API to only 100 milliseconds.

In this section we evaluate our attack techniques from within the Tor Browser and demonstrate that they are possible even within this restricted environment. We note that Shusterman et al. [69] have already demonstrated the Sweep Counting attack in the Tor Browser. We extends that result, demonstrating that making the environment more restrictive by disabling JavaScript feature does not guarantee protection.

Negative Result: DNS Racing and CSS Prime+Probe. We begin with a negative result, that the CSS Prime+Probe attack we designed is not effective in the Tor Browser. The cause is that for security reasons, the Tor Browser does not directly resolve DNS requests. Instead, it asks a Tor exit relay to resolve the name on its behalf. This extra redirection step adds a very large delay to DNS requests, on the order of hundreds of milliseconds, as well as a high degree of jitter, well beyond what the attack can handle. This issue also affects the DNS Racing attack, making it inapplicable.

Adapting String and Sock to Tor. The String and Sock technique described in Section 3.3 uses a high bandwidth WebSockets connection to offload timing measurements to a remote server. Unfortunately, due to the high round-trip delay of a Tor connection, the bandwidth available to a WebSockets connection over the Tor transport is significantly lower than a connection made over a regular TCP transport. Effectively the connection operates in a *stop-and-wait* mode, buffering outgoing packets as long as not all previously transmitted

packets are acknowledged. This buffering removes the timing information that the attack needs.

To avoid buffering, we reduce the communication of our String and Sock attack by sending a probe packet only once every n sweeps over the cache, instead of after every sweep. We experimentally find that $n = 72$ provides the best accuracy.

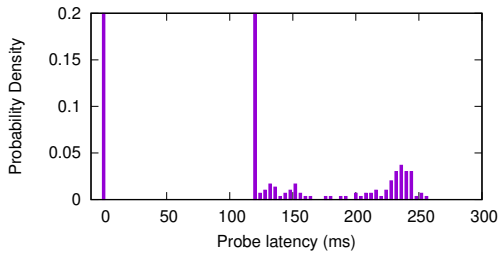


Figure 11: String and Sock Probe latency distribution on Tor Browser using an Intel i5-3470 target (6MB LLC).

Observing the Distribution of Probe Times. Figure 11 shows the probe time distribution using the Intel i5-3470 target. As the figure shows, there are three main elements to this distribution. First, we note a large subset of the probes have a fixed latency of around 120 ms. These are buffered by Tor's network layer, as described above, and sent immediately after all previously sent packets are acknowledged. Thus, these packets do not measure contention of the cache, but instead measure the round-trip delay of the Tor connection. Next, a large number of probes have a near-zero latency. These are packets which are sent together with other packets, and similarly do not encode any cache information. The final subset of the probes has a more diverse set of values, with an estimated mean of between 150 and 250 milliseconds. These probes encode cache contention information.

Website Fingerprinting. To demonstrate that these probes indeed contain cache information, we collect a dataset of 10,000 traces of Alexa Top 100 websites on the i5-3470 target running Tor Browser, using our adapted String and Sock method described above. Using this data, we can correctly fingerprint websites, obtaining a Top-1 accuracy of 20% and a Top-5 accuracy of 49%. Well above base rates of 1% and 5%, respectively. This demonstrates that completely eliminating access to timer and array APIs in the Tor Browser does prevent cache attacks.

6.2 Attacking DeterFox

DeterFox is a Firefox fork aiming to provably prevent timing attacks from within browser executed code [14]. Its authors argue that when using DeterFox, "an observer in a JavaScript reference frame will always obtain the same fixed timing information, so that timing attacks are prevented". To achieve this, DeterFox splits its execution context into multiple deterministic reference frames, and uses a priority-based event queue for communication between these reference.

However, we note that our CSS Prime+Probe technique does not require any JavaScript, with the colluding DNS server providing time measurement remotely. Thus, our techniques effectively sidestep all of the side channel protections offered by DeterFox. To demonstrate the effectiveness of our attacks on DeterFox, we collect one more dataset of 10,000 traces of Alexa Top 100 websites, using the CSS Prime+Probe method while using DeterFox. As expected, DeterFox's provably secure deterministic timing countermeasure did not prevent our attack, giving us a Top-1 accuracy of 66% and a Top-5 accuracy of 88%.

7 Conclusion

This paper shows that defending against JavaScript-based side-channel attacks is more difficult than previously considered. We show that advanced variants of the cache contention attack allow Prime+Probe attacks to be mounted through the browser in extremely constrained situations. Cache attacks cannot be prevented by reduced timer resolution, by the abolition of timers, threads, or arrays, or even by completely disabling scripting support. This implies that any secret-bearing process which shares cache resources with a browser connecting to untrusted websites is potentially at risk of exposure.

We also show that the reduced requirements of our attack make it agnostic across a variety of microarchitectures with no modifications. This allows us to present the first end-to-end side-channel attack which targets Apple's new M1 processors.

So, how can security-conscious users access the web? One complicating factor to this concept is the fact that the web browser makes use of additional shared resources beyond the cache, such as the operating system's DNS resolver, the GPU and the network interface. Cache partitioning seems a promising approach, either using spatial isolation based on cache coloring [40], or by OS-based temporal isolation [23].

Acknowledgements

This work was supported the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; an ARC Discovery Early Career Researcher Award (project number DE200101577); an ARC Discovery Project (project number DP210102670); the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL) under contracts FA8750-19-C-0531 and HR001120C0087; Israel Science Foundation grants 702/16 and 703/16; the National Science Foundation under grant CNS-1954712; the Research Center for Cyber Security at Tel-Aviv University established by the State of Israel, the Prime Minister's Office and Tel-Aviv University; and gifts from Intel and AMD.

The authors thank Jamil Shusterman for his assistance in bringing up the measurement setup.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[2] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *FDTC*. IEEE Computer Society, 2007.

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, pages 225–242, 2007.

[4] Array.prototype.pop. Array.prototype.pop(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/pop, 2020.

[5] Array.prototype.push. Array.prototype.push(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push, 2020.

[6] Jo M. Booth. Not so incognito: Exploiting resource-based side channels in JavaScript engines. Bachelor thesis, Harvard, April 2015.

[7] 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.

[8] Samira Briongos, Pedro Malagón, José Manuel Moya, and Thomas Eisenbarth. Reload+Refresh: abusing cache replacement policies to perform stealthy cache attacks. In *USENIX Security*, pages 1967–1984, 2020.

[9] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[10] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE SP*, pages 870–887, 2019.

[11] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4): 213–242, 2019.

[12] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *USENIX Sec*, pages 255–270, 2015.

[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*, pages 249–266, 2019.

[14] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. Deterministic browser. In *CCS*, pages 163–178, 2017.

[15] Alex Christensen. Reduce resolution of performance.now. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now, 2015.

[16] Chromium Project. window.performance.now does not support sub-millisecond precision on Windows. https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110, 2016.

[17] David Cock, Qian Ge, Toby C. Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. In *CCS*, pages 570–581, 2014.

[18] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):171–191, 2018.

[19] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *TACO*, 8(4):35:1–35:21, 2012.

[20] ECMA International. ECMAScript 2016 language specification. https://www.ecma-international.org/ecma-262/7.0/index.html, 2016.

[21] I. Fette and A. Melnikov. The WebSocket protocol. RFC 6455, IETF, December 2011.

[22] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8(1):1–27, 2018.

[23] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing OS abstraction. In *EuroSys*, pages 1:1–1:17, 2019.

[24] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ACNS*, pages 83–102, 2018.

[25] Daniel Genkin, Romain Poussier, Rui Qi Sim, Yuval Yarom, and Yuanjing Zhao. Cache vs. key-dependency: Side channeling an implementation of Pilsung. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1): 231–255, 2020.

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

[27] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, pages 955–972, 2018.

[28] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed JavaScript. In *ESORICS*, pages 108–122, 2015.

[29] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, pages 897–912, 2015.

[30] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *DIMVA*, pages 300–321, 2016.

[31] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *DIMVA*, pages 279–299, 2016.

[32] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *IEEE SP*, pages 490–505, 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*, pages 214–227, 2019.

[34] Andrew Hintz. Fingerprinting websites using traffic analysis. In *Privacy Enhancing Technologies*, 2002.

[35] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *IEEE SP*, pages 8–20, 1991.

[36] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE SP*, pages 191–205, 2013.

[37] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *CHES*, pages 368–388, 2016.

[38] Marc Juárez, Sadia Afroz, Gunes Acar, Claudia Díaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *CCS*, pages 263–274, 2014.

[39] Hyungsub Kim, Sangho Lee, and Jong Kim. Inferring browser activity and status through remote monitoring of storage usage. In *ACSAC*, 2016.

[40] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security Symposium*, pages 189–204. USENIX Association, 2012.

[41] 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*, pages 1–19, 2019.

[42] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Sec*, pages 463–480, 2016.

[43] Erick Lavoie, Bruno Dufour, and Marc Feeley. Portable and efficient run-time monitoring of JavaScript applications using virtual machine layering. In *ECOOP 2014*, pages 541–566, 2014.

[44] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, pages 557–574, 2017.

[45] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *RTAS*, pages 213–224, 1997.

[46] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *USENIX Security*, pages 549–564, 2016.

[47] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed JavaScript. In *ESORICS (2)*, pages 191–209, 2017.

[48] 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*, pages 973–990, 2018.

[49] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015.

[50] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, pages 406–418, 2016.

[51] Giorgio Maone. Noscript. https://noscript.net.

[52] Bynens Mathias. Elements kinds in V8. https://v8.dev/blog/elements-kinds, 2017.

[53] Nikolay Matyunin, Yujue Wang, Tolga Arul, Kristian Kullmann, Jakub Szefer, and Stefan Katzenbeisser. Magneticspy: Exploiting magnetometer in mobile devices for website and application fingerprinting. In *WPES*, pages 135–149, 2019.

[54] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *DIMVA*, pages 46–64, 2015.

[55] Arvind Narayanan, Hristo Paskov, Neil Zhenqiang Gong, John Bethencourt, Emil Stefanov, Eui Chul Richard Shin, and Dawn Song. On the feasibility of internet-scale author identification. In *IEEE SP*, pages 300–314, 2012.

[56] Rom Ogen, Kfir Zvi, Omer Shwartz, and Yossi Oren. Sensorless, permissionless information exfiltration with Wi-Fi micro-jamming. In *WOOT*, 2018.

[57] 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*, pages 1406–1418, 2015.

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

[59] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In Yan Chen and Jaideep Vaidya, editors, *WPES*, pages 103–114, 2011.

[60] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, 2005. URL http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf.

[61] Moinuddin K. Qureshi. CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, pages 775–787, 2018.

[62] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel S. Emer. Set-dueling-controlled adaptive insertion for high-performance caching. *IEEE Micro*, 28(1):91–98, 2008.

[63] Vera Rimmer, Davy Preuveneers, Marc Juárez, Tom van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. In *NDSS*, 2018.

[64] 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*, pages 199–212, 2009.

[65] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of Bleichenbacher's CAT: new cache attacks on TLS implementations. In *IEEE SP*, pages 435–452, 2019.

[66] 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*, pages 247–267, 2017.

[67] Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript Zero: Real JavaScript and zero side-channel attacks. In *NDSS*, 2018.

[68] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DSN Workshops*, pages 194–199. IEEE Computer Society, 2011.

[69] 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*, pages 639–656, 2019.

[70] Paul Stone. Pixel perfect timing attacks with HTML5. https://www.contextis.com/media/downloads/Pixel_Perfect_Timing_Attacks_with_HTML5_Whitepaper.pdf, 2013.

[71] The Tor Project, Inc. The Tor Browser. https://www.torproject.org/projects/torbrowser.html.en.

[72] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *ACSAC*, pages 1382–1393, 2015.

[73] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In *CCSW*, pages 41–46, 2011.

[74] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in Chrome. In *USENIX Sec*, pages 849–864, 2017.

[75] W3C. Webassembly JavaScript interface. https://webassembly.github.io/spec/js-api/index.html, 2020.

[76] Daimeng Wang, Zhiyun Qian, Nael B. Abu-Ghazaleh, and Srikanth V. Krishnamurthy. PAPP: prefetcher-aware prime and probe side-channel attack. In *DAC*, page 62, 2019.

[77] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, pages 494–505, 2007.

[78] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *USENIX Security*, pages 675–692, 2019.

[79] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security*, 2020.

[80] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014.

[81] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. IACR Cryptology ePrint Archive 2015/905, 2015.

[82] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *CHES*, pages 346–367, 2016.

[83] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer Berlin Heidelberg, 2003.

[84] Boris Zbarsky. Clamp the resolution of performance.now() calls to 5us. https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab, 2015.

[85] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented Flush-Reload side channels on ARM and their implications for android devices. In *CCS*, pages 858–870, 2016.

[86] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, pages 305–316, 2012.

A Machine Learning Model

Our machine learning classifier receives as input a side-channel trace, and outputs a probability distribution over the 100 potential websites. Before the trace is fed to the model, the input vector was normalized between 0 and 1. We then used a deep learning network to perform our analysis, meaning that feature extraction was done inside the neural network and did not require additional preprocessing steps. We used the deep learning model whose hyperparameters are presented in Table 7. The model begins with a convolution layer which learns the unique patterns of each label, followed by a Max-Pooling layer which reduces the dimensionality of the output of the previous layer. The output of the Max-Pooling layer is then reshaped to a one dimension vector and fed to a Long-Short Term Layer, which extracts temporal features over its input. Finally, the output layer of the network is a fully-connected layer with a softmax activation function.

The model was evaluated on a test set whose traces are not part of the training set. The metric we use is accuracy – the probability of a trace to be classified correctly. To avoid overfitting in model estimation, we employ 10 fold cross validation, a method which divides the dataset into 10 parts, with each part becoming the test set while the others are used as the train set. Each training set is fed to a different model, and the evaluation is made on the related test set. After each experiment, we noted the average cross-fold accuracy, as well as the standard deviation between folds.

Table 7: Hyperparameters for the deep learning classifier

Hyperparameter	**Value**
Optimizer	Adam
Learning rate	0.001
Batch size	128
Training epoch	Early stop by validation accuracy
Input units	vector size of the 30 seconds input
Convolution layers	2
Convolution activation	relu
Convolution Kernels	256
Convolution Kernel size	16,8
Pool size	4
LSTM activation	tanh
LSTM units	32
Dropout	0.7

The output of our classifier is not only the label of the most probable class, but rather a complete probability distribution over all possible labels. This flexibility allows us to capture the case where the attacker has some prior knowledge of the victim and some expectation of the websites they may be browsing. To do so, we look not only at the top-rated label, but also at a few of the next most probable predictions. This methodology was previously used in similar works where low-accuracy classifiers were evaluated [12, 55]. We thus calculated not only the raw accuracy, but also the probability that the right prediction is among the top 5 websites output as the most probable by the classifier. The base accuracy rate of this prediction method, as obtained by a random classifier with no knowledge of the traces, is 5%.

The machine learning model was implemented in python version 3.6, using TensorFlow [1] library version 1.4. The model training algorithms were run on a cluster made out of Nvidia GTX1080 and GTX2080 graphics processing units (GPUs), managed by Slurm workload manager [83] version 19.05.4.