JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks

Michael Schwarz, Moritz Lipp and Daniel Gruss
Graz University of Technology
{michael.schwarz,moritz.lipp,daniel.gruss}@iaik.tugraz.at

***Abstract*—Modern web browsers are ubiquitously used by billions of users, connecting them to the world wide web. From the other side, web browsers do not only provide a unified interface for businesses to reach customers, but they also provide a unified interface for malicious actors to reach users. The highly optimized scripting language JavaScript plays an important role in the modern web, as well as for browser-based attacks. These attacks include microarchitectural attacks, which exploit the design of the underlying hardware. In contrast to software bugs, there is often no easy fix for microarchitectural attacks.**

We propose *JavaScript Zero*, a highly practical and generic fine-grained permission model in JavaScript to reduce the attack surface in modern browsers. *JavaScript Zero* facilitates advanced features of the JavaScript language to dynamically deflect usage of dangerous JavaScript features. To implement *JavaScript Zero* in practice, we overcame a series of challenges to protect potentially dangerous features, guarantee the completeness of our solution, and provide full compatibility with all websites. We demonstrate that our proof-of-concept browser extension *Chrome Zero* protects against 11 unfixed state-of-the-art microarchitectural and side-channel attacks. As a side effect, *Chrome Zero* also protects against 50 % of the published JavaScript 0-day exploits since Chrome 49. *Chrome Zero* has a performance overhead of 1.82% on average. In a user study, we found that for 24 websites in the Alexa Top 25, users could not distinguish browsers with and without *Chrome Zero* correctly, showing that *Chrome Zero* has no perceivable effect on most websites. Hence, *JavaScript Zero* is a practical solution to mitigate JavaScript-based state-of-the-art microarchitectural and side-channel attacks.

I. Introduction

Over the past 20 years, JavaScript has evolved to the predominant language on the web. Of the 10 million most popular websites, 94.7 % use JavaScript [54]. Dynamic content relies heavily on JavaScript, and thus, most pages use JavaScript to improve the user experience, using, e.g., AJAX and dynamic page manipulation. Especially for platform-independent HTML5 applications, JavaScript is a vital component.

With the availability of modern browsers on mobile devices, web applications target smartphones and tablets as well. Furthermore, mobile platforms typically provide a number of sensors and features not present on commodity laptops and desktop computers. To make use of these additional features, the World Wide Web Consortium (W3C) provides drafts and recommendations for additional APIs [53]. Examples include the Geolocation API [49] and the Battery Status API [48]. These APIs are supported by most browsers and allow developers to build cross-platform web applications with similar functionality as native applications.

Undoubtedly, allowing every website to use such APIs has security and privacy implications. Websites can exploit sensors to fingerprint the user [52] by determining the number of sensors, their update frequency, and also their value (e.g., for battery level or geolocation). Furthermore, sensor data can be exploited to mount side-channel attacks on user input [7], [23].

Microarchitectural attacks can also be implemented in JavaScript, exploiting properties inherent to the design of the microarchitecture, such as timing differences in memory accesses. Although JavaScript code runs in a sandbox, Oren et al. [33] demonstrated that it is possible to mount cache attacks in JavaScript. Since their work, a series of microarchitectural attacks have been mounted from websites, such as page deduplication attacks [14], Rowhammer attacks [15], ASLR bypasses [13], and DRAM addressing attacks [40].

As a response to these attacks, some—but not all—of the APIs have been restricted by reducing the resolution (e.g., High Precision Time API) [2], [6], [9] or completely removing them (e.g., DeviceOrientation Event Specification) [51]. However, these countermeasures are incomplete as they do not cover all sensors and are circumventable [13], [40].

A common trait of all attacks is that they rely on the behavior of legitimate JavaScript features, which are rarely required by benign web applications. However, removing these JavaScript features entirely breaks the compatibility with the few websites that use them in a non-malicious way. This is, for example, the case with NoScript, a browser extension that completely blocks JavaScript on a page [10].

We propose *JavaScript Zero*, a fine-grained JavaScript permission system, which combines ideas from existing permission systems in browsers and smartphone operating systems. *JavaScript Zero* facilitates advanced features of the JavaScript language to overcome the following three challenges:

- C1 Restrictions must not be circumventable using self-modifying code, such as higher-order scripts.
- C2 Restricting access to potentially dangerous features must be irreversible for a website.
- C3 Restrictions must not have a significant impact on compatibility and user experience.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23094
www.ndss-symposium.org

To overcome challenge C1, we utilize advanced language features such as virtual machine layering [19] for security. In contrast to previous approaches [56], virtual machine layering allows redefining any function of the JavaScript virtual machine at runtime. Hence, we can cope with obfuscated code and higher-order scripts, *i.e.*, scripts that generate scripts. *JavaScript Zero* replaces all potentially dangerous functions with secure wrappers. When calling such a function, *JavaScript Zero* decides whether to perform a pre-defined action or ask the user for permission to execute the function.

To overcome challenge C2, we utilize closures, another advanced feature of the JavaScript language, for security. Variables in closures cannot be accessed from any outside scope, providing us with language-level protection for our countermeasure. With closures, we make all references to original unprotected functions inaccessible to the website.

We provide a proof-of-concept implementation as a Chrome browser extension, *Chrome Zero*. In contrast to previous protection techniques [24], [18], *Chrome Zero* requires no changes to the browser source code. Hence, *Chrome Zero* requires lower maintenance efforts while at the same time users also benefit from the security of the most up-to-date browser.

To overcome challenge C3, we keep user interactions to a minimum and provide multiple *protection levels* with pre-defined restrictions. We do not only provide a binary permission system to block or allow certain functionality, but we also allow to modify the semantics of functions and objects. For example, the user can allow the usage of the high-precision timing API but decides to reduce the available resolution to $100\,\mathrm{ms}$ instead of $5\,\mu\mathrm{s}$. These settings can be configured by either the user or the community, through a *protection list*.

We evaluate the efficacy of *Chrome Zero* on 23 recent side-channel attacks, microarchitectural attacks, and 0-day exploits. We show that it successfully prevents all microarchitectural and side-channel attacks in JavaScript. Although not a main goal of *Chrome Zero*, it also prevents $50\,\%$ of the published JavaScript 0-day exploits since Chrome 49. This shows that we were able to solve challenges C1 and C2.

To evaluate whether *Chrome Zero* solves challenge C3, we measure the performance overhead and the impact on the user experience for the Alexa Top 25 websites, at the second highest security level. On average, we observe a performance overhead of $1.82\,\%$. In a double-blind user study, we found that for 24 websites out of the Alexa Top 25, users could not distinguish browsers with and without *Chrome Zero* showing that *Chrome Zero* has no significant effect on the user experience.

Contributions. The contributions of this work are:

1) We propose *JavaScript Zero*, a fine-grained JavaScript permission system to mitigate state-of-the-art microarchitectural and side-channel attacks.
2) We show that combining advanced and novel features of the JavaScript language, e.g., virtual machine layering, closures, proxy objects, and object freezing, can be retrofitted to form a basis for strong security boundaries.
3) We show that *JavaScript Zero* successfully prevents all published microarchitectural and side-channel attacks and as a side effect also mitigates $50\,\%$ of the published JavaScript 0-day exploits since Chrome 49.
4) We evaluate our proof-of-concept implementation *Chrome Zero* in terms of performance and usability. *Chrome Zero* has $1.82\,\%$ performance overhead on average on the Alexa Top 10 websites. In a double-blind user study, we show that users cannot distinguish a browser with and without *Chrome Zero* for 24 of the Alexa Top 25 websites.

The remainder of the paper is organized as follows. Section II provides preliminary information necessary to understand the defenses we propose. Section III defines the threat model. Section IV describes the design of *JavaScript Zero*. Section V details our proof-of-concept implementation *Chrome Zero*. Section VI provides a security analysis of *Chrome Zero* as an instance of *JavaScript Zero*. Section VII provides a usability analysis of *Chrome Zero*. Section VIII discusses related work. We conclude our work in Section IX.

II. Preliminaries

In this section, we provide preliminary information on microarchitectural attacks in native code and JavaScript, and on JavaScript exploits.

A. Microarchitectural Attacks

Modern processors are highly optimized for computational power and efficiency. However, optimizations often introduce side effects that can be exploited in so-called microarchitectural attacks. Microarchitectural attacks comprise side-channel and fault attacks on microarchitectural elements or utilizing microarchitectural elements, e.g., pipelines, caches, buses, DRAM. Attacks on caches have been investigated extensively in the past 20 years, with a focus on cryptographic implementations [17], [4]. The timing difference between a cache hit and a cache miss can be exploited to learn secret information from co-located processes and virtual machines. Modern attacks use either Flush+Reload [55], if read-only shared memory is available, or Prime+Probe [34] otherwise. In both attacks, the attacker manipulates the state of the cache and later on checks whether the state has changed. Besides attacks on cryptographic implementations [34], [55], these attack primitives can also be used to defeat ASLR [13] or to build covert-channels [22].

B. Microarchitectural and Side-Channel Attacks in JavaScript

Microarchitectural attacks were only recently exploited from JavaScript. As JavaScript code is sandboxed and inherently single-threaded, attackers face certain challenges in contrast to attacks in native code. We identified several requirements that are the basis for microarchitectural attacks, *i.e.*, every attack relies on at least one of these primitives. Moreover, sensors found on many mobile devices, as well as modern browsers introduce side-channels which can also be exploited from JavaScript. Table I gives an overview of, to the best of our knowledge, all 11 known microarchitectural and side-channel attacks in JavaScript and their requirements.

a) Memory Addresses: JavaScript is a sandboxed scripting language which does not expose the concept of pointers to the programmer. Even though pointers are used internally, the language never discloses virtual addresses to the programmer. Thus, an attacker cannot use language features to gain knowledge of virtual addresses. The closest to virtual addresses

TABLE I: REQUIREMENTS OF STATE-OF-THE-ART SIDE-CHANNEL ATTACKS IN JAVASCRIPT.

	Memory addresses	Accurate timing	Multithreading	Shared data	Sensor API
Rowhammer.js [15]	●	●	○	○	○
Practical Memory Deduplication Attacks in Sandboxed Javascript [14]	◐	●	○	○	○
Fantastic Timers and Where to Find Them [40]	●	●†	◐	◐	○
ASLR on the Line [13]	●	●†	◐	◐	○
The spy in the sandbox [33]	◐	●	○	○	○
Loophole [47]	○	◐	●	○	○
Pixel perfect timing attacks with HTML5 [44]	○	●†	◐	◐	○
The clock is still ticking [45]	○	●	◐	○	○
Practical Keystroke Timing Attacks in Sandboxed JavaScript [20]	○	◐†	●	◐	○
TouchSignatures [23]	○	○	○	○	●
Stealing sensitive browser data with the W3C Ambient Light Sensor API [31]	○	○	○	○	●

† If accurate timing is not available, it can be approximated using a combination of multithreading and shared data.

are `ArrayBuffers`, contiguous blocks of virtual memory. `ArrayBuffers` are used in the same way as ordinary arrays but are faster and more memory efficient, as the underlying data is actually an array which cannot be resized [26]. If one virtual address within an `ArrayBuffer` is identified, the remaining addresses are also known, as both the addresses of the memory and the array indices are linear [14], [13].

Gras et al. [13] showed that `ArrayBuffers` can be exploited to reconstruct virtual addresses. An attacker with knowledge of virtual addresses has effectively defeated address space layout randomization, thus circumventing an important countermeasure against memory corruption attacks.

Microarchitectural attacks typically do not rely on virtual addresses but physical addresses. Cache-based attacks [33], [15] rely on parts of the physical address to determine cache sets. DRAM-based attacks [15], [14], [40] also rely on parts of the physical address to determine the beginning of a page or a DRAM row. However, for security reasons, an unprivileged user does not have access to the virtual-to-physical mapping. This is not only true for JavaScript, but also for any native application running on a modern operating system.

Consequently, microarchitectural attacks have to resort to side-channel information to recover this information. Gruss et al. [14] and Gras et al. [13] exploit the fact that browser engines allocate `ArrayBuffers` always page aligned. The first byte of the `ArrayBuffer` is therefore at the beginning of a new physical page and has the least significant 12 bits set to '0'.

For DRAM-based attacks, this is not sufficient, as they require more bits of the physical address. These attacks exploit another feature of browser engines and operating systems. If a large chunk of memory is allocated, browser engines typically use `mmap` to allocate this memory, which is optimized to allocate 2 MB transparent huge pages (THP) instead of 4 KB pages [15], [40]. As these physical pages are mapped on demand, *i.e.*, as soon as the first access to the page occurs, iterating over the array indices results in page faults at the beginning of a new page. The time to resolve a page fault is significantly higher than a normal memory access. Thus, an attacker knows the index at which a new 2 MB page starts. At this array index, the underlying physical page has the 21 least significant bits set to '0'.

b) Accurate Timing: Accurate timing is one of the most important primitives, inherent to nearly all microarchitectural and side-channel attacks. As most of the microarchitectural and side-channel attacks exploit some form of timing side channel, they require a way to measure timing differences. The required resolution depends greatly on the underlying side channel. For example, DRAM-row conflicts [15], [40], cache-timing differences [33], [13], and interrupt timings [20] require a timing primitive with a resolution in the range of nanoseconds, whereas for detecting page faults [14], [15], [40], exploiting SVG filters [44], or mounting cross-origin timing attacks [45], a resolution in the range of milliseconds is sufficient.

JavaScript provides two interfaces for measuring time. The `Date` object represents an instance in time, used to get an absolute timestamp. The object provides a method to get a timestamp with a resolution of 1 ms. The second interface is the `Performance` object which is used to provide information about page performance. This interface provides several timing relevant properties and functions, such as the `performance.now()` function, which provides a highly accurate timestamp in the order of microseconds [40]. Another part of the `Performance` object is the User Timing API, a benchmarking feature for developers, which also provides timestamps in the order of microseconds.

However, the resolution of these built-in timers is not high enough to measure microarchitectural side channels, where the timing differences are mostly in the order of nanoseconds. Thus, such attacks require a custom timing primitive. Usually, it is sufficient to measure timing differences, and an absolute timestamp is not necessary. Thus, access to a full-blown clock is not required, and attackers usually settle for some form of a monotonically incremented counter as a clock replacement. Kohlbrenner et al. [18] and Schwarz et al. [40] investigated new methods to get highly accurate timing. Most of their timing primitives rely on either building counting loops using message passing [40] or on interfaces for multimedia content [18]. Using such self-built timers, it is possible to measure timing differences with a nanosecond resolution.

c) Multithreading: JavaScript is inherently single-threaded and based on an event loop. All events, such as function calls or user inputs, are pushed to this queue and then serially, and thus synchronously, handled by the engine. HTML5 introduced multithreading to JavaScript, in the form of worker threads (web workers), allowing real parallelism for JavaScript code. With web workers, every worker has its own (synchronous) event queue. The synchronization is handled via

messages, which are again events. Thus, JavaScript does not require explicit synchronization primitives.

The support for true parallelism allows to mount new side-channel attacks. Vila et al. [47] exploited web workers to spy on different browser windows by measuring the dispatch time of the event queue. This timing side-channel attack allows to detect user inputs and identify pages which are loaded in a different browser window. A similar attack using web workers was shown by Lipp et al. [20]. However, this attack does not exploit timing differences in the browser engine, but on the underlying microarchitecture. An endless loop running within a web worker detects CPU interrupts, which can then be used to deduce keystroke information.

d) Shared Data: To synchronize and exchange data, web workers have to rely on message passing. Message passing has the advantage over unrestricted memory sharing as there is no requirement for synchronization primitives. Sending an object to a different worker transfers the ownership of the object as well. Thus, objects can never be changed by multiple workers in parallel.

As transferring the ownership of objects can be slow, JavaScript introduced `SharedArrayBuffers`. A `SharedArrayBuffer` is a special object which behaves the same as a normal `ArrayBuffer`, but it can be simultaneously accessed by multiple workers. Inherently, this can reintroduce synchronization problems.

Schwarz et al. [40] and Gras et al. [13] showed that this shared data can be exploited to build timing primitives with a nanosecond resolution. Their timing primitive requires only one worker running in an endless loop and incrementing the value in a `SharedArrayBuffer`. The main thread can simply use the value inside this shared buffer as a timestamp. Using this method, it is possible to get a timestamp resolution of $2\,\mathrm{ns}$, which is almost as high as Intel's native timestamp counter, and thus sufficient to mount DRAM- and cache-based side-channel attacks.

e) Sensor API: As JavaScript is also used on mobile devices, HTML5 introduced interfaces to interact with device sensors. Some sensors are already restricted by the existing permission system in the browser, such as the geolocation API. This permission system uses callback functions to deliver results. Hence, it is inherently incompatible with existing synchronous APIs and cannot be instrumented to protect arbitrary JavaScript functions. As these sensors can affect the user's privacy, the user has to explicitly permit usage of these interfaces on a per-page basis. However, several other sensors are not considered invasive in terms of security or privacy.

Mehrnezhad et al. [23] showed that access to the motion and orientation sensor can compromise security. By recording the data from these sensors, they were able to infer PINs and touch gestures (e.g., zoom) of the user. Although not implemented in JavaScript, Spreitzer [42] showed that access to the ambient light sensor (as specified by the W3C [50]) can also be exploited to infer user PINs. Similarly, Olejnik [31] utilized the Ambient Light Sensor API to recover information on the user's browsing history, to violate the same-origin policy, and to steal cross-origin data.

C. JavaScript Exploits

In addition to microarchitectural and side-channel attacks, there are also JavaScript-based attacks exploiting vulnerabilities in the JavaScript engine. An exploit triggers an implementation error in the engine to divert the control flow of native browser code. These implementation errors can—and should—be fixed by browser vendors. Side-channel attacks, however, often arise from the hardware design. In contrast to software, the hardware and hardware design cannot be easily changed.

As exploits are based on implementation errors and not design issues, we cannot identify general requirements for such attacks. Every JavaScript function and each interface can be potentially abused if there is a vulnerability in the engine. Thus, we cannot provide a general protection against exploits, and exploits are therefore not in the scope of this paper. However, we can still reduce the attack surface of the browser, and we provide practical protection against $50\,\%$ of the published JavaScript 0-day exploits since Chrome 49.

Exploits often rely on arrays to craft their payload. Moreover, bugs are often triggered by errors in functions responsible for parsing complex data (e.g., JSON). As some of the functions used in exploits are also requirements for microarchitectural and side-channel attacks, we also evaluate exploits in this paper to confirm that our permission system is also applicable to reduce the general attack surface of the browser, *i.e.*, hardening browsers against 0-day exploits until they are fixed by the browser vendors.

III. Threat Model

In our threat model, we assume that the attacker is capable of performing state-of-the-art microarchitectural and software-based side-channel attacks in JavaScript. This is a reasonable assumption, as we found most published attacks to be accompanied with proof-of-concept source code allowing us to reproduce the attacks.

We assume that the victim actively uses a browser, either natively, or in a virtual machine. The attacker resides either in a different, co-located virtual machine [14], [40] or—for most attacks—somewhere else on the internet. In all state-of-the-art microarchitectural and software-based side-channel attacks, the attacker has some form of remote code execution. In line with these works, we assume that the attacker was able to maliciously place the attack code in a benign website. This can be achieved if the benign website either includes content from a (malicious) third party, such as advertisements or libraries, or if an attacker has compromised the benign site in some way. Another possibility is that the victim navigated to a malicious website controlled by the attacker. Hence, in all cases, the *attacker-controlled JavaScript code is executed in the victim's browser*.

The browser contains a JavaScript engine that executes code embedded in a website inside the browser sandbox. The sandbox ensures that JavaScript code cannot access any system resources not intended to be accessed. Furthermore, every page has its own execution context protected by the sandbox, *i.e.*, code on different pages cannot influence each other. We assume that an attacker is not aware of exploitable bugs in the JavaScript engine, and hence, can *only use legitimate JavaScript features*. Exploiting bugs in the interpreter,

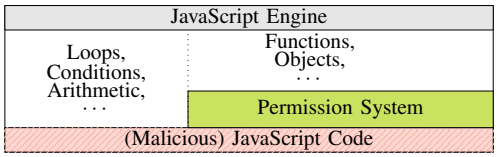


Fig. 1: The permission system acts as an abstraction layer between the JavaScript engine and the interfaces provided to a JavaScript developer.

```
function : {
  "window.performance.now":
    { action: "modify",
      return: "Math.floor(window.performance.now()
        / 1000.0) * 1000.0" },
  "history.back":
    { action: "block" },
  "navigator.getBattery":
    { action: "ask", default: "null" }
```

Listing 1: Excerpt of a protection policy. The function `performance.now` is modified to return timestamps with a lower resolution, the function `history.back` is blocked.

sandbox, or other execution environments, is out of scope for this paper.

IV. Design of *JavaScript Zero*

In this section, we present the design of our JavaScript permission system, *JavaScript Zero*. We propose a fine-grained policy-based system which allows to change the behavior of standard JavaScript interfaces and functions. Policies enforce certain restrictions to a website to protect users from malicious JavaScript. They allow to quickly adapt the permission system to protect against newly discovered attacks. Furthermore, different policies can be combined by the user, depending on the desired level of protection.

The idea of *JavaScript Zero* is to introduce an abstraction layer between the JavaScript engine and the interface provided to a (malicious) JavaScript developer. The basic idea of this layer is to protect functions, interfaces, and object properties, as shown in Figure 1. The abstraction layer can block, modify, or simply forward every interaction of the code with the JavaScript engine. The layer is completely transparent to the web application and, thus, no modification of any existing source code is required to deploy *JavaScript Zero*. *JavaScript Zero* can intercept all calls to functions provided by the language, which also includes constructors of objects and getters of object properties. However, it does not interfere with the constructs of the language itself, *i.e.*, loops and primitive data types bypass the abstraction layer. In the remainder of this paper, we use the term "functions" to refer to general functions, object constructors, and getters of object properties for the sake of brevity.

The exact behavior of *JavaScript Zero* is defined by a *protection policy*. A protection policy is a machine-readable description which contains a policy for every function, property, or object that should be handled by the permission system. Listing 1 shows an excerpt of such a policy. In this sample policy, the function to go back to the last website is completely blocked, *i.e.*, if a script calls this function, it does nothing. Furthermore, the resolution of the high-resolution timer is reduced from several microseconds to one second. Finally, the battery API requires permission from the user, and if the user denies access to the function, it simply returns no information.

The policies can be designed by any user and shared with other users. Thus, as soon as a new exploit, side-channel attack, or microarchitectural attack emerges, a new policy preventing it can be created and shared with all users. We propose a community-maintained policy repository where users can subscribe to certain kinds of policies, e.g., more or less strict policies for their specific hardware and software. The functionality of *JavaScript Zero* does not fundamentally rely on the community, and every user can also write their own policies, or thoroughly inspect third-party policies before applying them. Hence, a careful user can avoid the inherent limitations of a community-maintained policy, e.g., adversarial modifications, which can happen in any open-source project.

For every policy, there are four different possibilities how it affects a function used on a website:

1) **Allow.** The function is explicitly allowed or not specified in the policy. In this case, no action is performed and the function can be used normally.
2) **Block.** The function is blocked. In this case, *JavaScript Zero* replaces the function by a stub that only returns a given default value.
3) **Modify.** The function is modified. In this case, *JavaScript Zero* replaces the original function with a policy-defined function. This function can still call the original function if necessary.
4) **User permission.** The function requires the permission of the user. In this case, *JavaScript Zero* has to pause execution of the current function, display a notification to the user, and wait for the response of the user.

In the fourth case, the user has to explicitly grant permission. The user can opt to save the decision, to not be bothered again, and, thus, user interruptions are kept to a minimum.

We opted for a browser extension, as it can be easily installed in a user's browser and neither relies on modification of the source code of the website or the browser, nor any external service, e.g., a web proxy. Thus, there is no constant maintenance of a forked browser source base necessary. Moreover, by designing *JavaScript Zero* as a browser extension, it can easily be implemented for any browser supporting extensions (e.g., Chrome, Firefox, Edge) as the design of *JavaScript Zero* is independent of the browser.

Figure 2 shows the general design of this approach. Functions are replaced by wrapper functions which can either immediately return a result or divert the control flow to the browser extension. The browser extension can then ask the user whether to allow the function call or block it.

To allow regular users to use such a browser extension on a day-to-day basis, we propose a simple interface for handling protection policies. This interface defines so-called *protection levels*, each grouping one or more protection policies. Thus, a user only chooses a protection level out of a predefined set of levels, e.g., one of *none*, *low*, *medium*, *high*, *paranoid*. Although this simplification reduces the flexibility of the

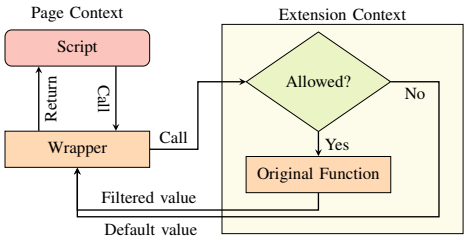


Fig. 2: A policy replaces a function by a wrapper. The extension implements the logic to ask the user whether the function shall be blocked or the original function is executed.

extension, we chose this approach as for a regular user it is clearly not feasible to choose from protection policies or even define custom protection policies.

V. Implementation of *Chrome Zero*

In this section, we describe *Chrome Zero*, our open-source[1] proof-of-concept implementation of *JavaScript Zero* for Google Chrome 49 and newer. Implementing *Chrome Zero* faces certain challenges:

- C1 Restrictions must not be circumventable using self-modifying code, such as higher-order scripts.
- C2 Restricting access to potentially dangerous features must be irreversible for a website.
- C3 Restrictions must not have a significant impact on compatibility and user experience.

In addition to the aforementioned challenges, implementing *JavaScript Zero* as a browser extension results in a trade-off between compatibility with up-to-date browsers and functionality we can use, *i.e.*, we cannot change the browser and thus have to rely on functions provided by the extension API.

First, we describe in Section V-A how to retrofit virtual machine layering for security and extend it for objects using *proxy objects* [28]. Virtual machine layering was originally developed for low-overhead run-time monitoring of functions [19]. We use it to guarantee that a policy is always applied to a function (Challenge C1). In Section V-B, we show that JavaScript closures in combination with object freezing can be utilized to secure the virtual machine layering approach to guarantee irreversibility of policies (Challenge C2). This combination of virtual machine layering and closures provides strong security boundaries for *Chrome Zero*. Finally, we discuss in Section V-C how to maintain practical usability of *Chrome Zero* (Challenge C3), despite the restrictions it introduces.

A. Virtual machine layering

To ensure that our own function is called instead of the original function, without modifying the browser, we facilitate a technique known as virtual machine layering [19]. Although this technique was originally developed for low-overhead run-time monitoring, we show that in combination with JavaScript closures, it can also be applied as a security mechanism. For security, virtual machine layering has a huge advantage

[1] *Chrome Zero*: `https://github.com/IAIK/ChromeZero`

```
1 var original_reference = window.performance.now;
2 window.performance.now = function() { return 0; };
3 // call the new function (via function name)
4 alert(window.performance.now()); // == alert(0)
5 // call the original function (only via reference)
6 alert(original_reference.call(window.performance));
```

Listing 2: Virtual machine layering applied to the function `performance.now`. The function name points to the new function, the original function can only be called using the reference.

over state-of-the-art source rewriting techniques [38], [56], [36], where functions are replaced directly in the source code. Ensuring that source rewriting cannot be circumvented is a hard problem, as function calls can be generated dynamically and are thus not always visible in the source code [1]. Support for such higher-order scripts is strictly necessary for full protection, as failing to apply a policy to only one function breaks the security guarantees of the security policies. However, higher-order scripts are often out-of-scope or not fully supported [24]. In contrast, virtual machine layering ensures that functions are replaced at the lowest level, right before they are executed by the JavaScript engine.

Listing 2 shows an example of virtual machine layering. As JavaScript allows to dynamically extend and modify prototypes, existing functions can be changed, and new functions can be added to every object. Virtual machine layering takes advantage of this language property. *Chrome Zero* saves a reference to the original function of an object and replaces the original function with a wrapper function. The original function can only be called using the saved reference. Calling the function by using the function name will automatically call the wrapper function. As *Chrome Zero* has full access to the website, it can use virtual machine layering to replace any original function. We can ensure that the code of *Chrome Zero* is executed before the page is rendered, and thus, that no other code can save a reference to the original function.

Additionally, virtual machine layering covers higher-order scripts without any additional costs. Higher-order scripts are scripts which are dynamically created (or loaded) and executed by existing scripts. There are multiple ways of creating higher-order scripts, including `eval`, script injection, function constructors, event handlers, and `setTimeout`. As any higher-order script automatically uses the re-defined function without further changes, policies are automatically applied to higher-order scripts as well, and there is no possibility for obfuscated code to circumvent the function replacement.

As *JavaScript Zero* supports policies not only for functions but also for properties and objects, we have to extend virtual machine layering, which was originally only intended for functions.

1) Properties: Typically, a property of a prototype is not a function, but a simple data value. JavaScript allows to replace all properties by special functions called *accessor properties*. Accessor properties are a special kind of property, where every access to the property triggers a user-defined function. In case the property was already an accessor property, *Chrome Zero* simply replaces the original function. Thus, regardless of the

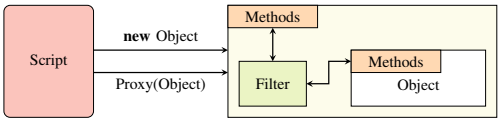


Fig. 3: Dangerous objects are wrapped in proxy objects. The proxy object decides whether methods of the original object are called or substituted by a different functions.

```
1 (function() {
2 // original is only accessible in this scope
3 var original = window.performance.now;
4 window.performance.now = function() {
5   return Math.floor((original.call(window.performance))
6     / 1000.0) * 1000.0;
7 }; })();
```

Listing 3: Virtual machine layering applied to the function `performance.now` within a closure. The function name points to the new function, the original function can only be called using the reference. However, the reference is not visible outside the scope, *i.e.*, only the wrapper function can access the reference to the original function.

type of property, we can convert every property to an accessor property using `Object.defineProperty`.

2) Objects: To be able to apply policies to objects, we wrap the original object within a *proxy object* as shown in Figure 3. The proxy object contains the original object and forwards all functions (which are not overwritten) to the original object. Thus, all states are still handled by the original object, and only functions for which a policy is defined have to be re-implemented in the proxy object.

Although JavaScript prototypes have a constructor, simply applying virtual machine layering to the constructor function is not sufficient. The constructor function is only a pointer to the real constructor and not used to actually instantiate the object. The alternative to the proxy object for replacing the constructor is to re-implement the entire object with all methods and properties. However, as this requires considerable engineering effort and cannot easily be automated, it is not feasible, and we thus rely on the proxy object.

B. Self-protection

An important part of the implementation is that it is not possible for an attacker to circumvent the applied policies (Challenge C2). Thus, an implementation has to ensure that an attacker cannot get hold of any reference to the original functions. We utilize JavaScript closures for security, by creating anonymous scopes not connected to any object and thus inaccessible to code outside of the closure. Listing 3 shows virtual machine layering wrapped in a closure.

With the original version of virtual machine layering as shown in Listing 2, an attacker could simply guess the name of the variable holding the original reference. Furthermore, all global variables are members of the JavaScript root object `window`, and an attacker could use reflection to iterate over all variables until the function reference is discovered. Closures provide a way to store data in a scope not connected to the `window` object. Thus, by applying the virtual machine layering process within a closure, the reference to the original function is still available to the wrapper function but inaccessible to any code outside of the closure. This guarantees that the virtual machine layering is irreversible.

At the time of writing, there is no mechanism to modify a function without redefining it. Thus, an attacker cannot inject new code into the wrapper function (or modify the existing code) without destroying the closure and therefore losing the reference to the original function.

Additionally, objects have to be protected using `Object.freeze` after the virtual machine layering process. This ensures that deleting the function does not revert to the original function pointer, as it is otherwise the case in Google Chrome.

If a policy requires user interaction, *i.e.*, the user has to decide whether a function shall be executed or not, this logic must also be protected against an attacker. By relying on a browser extension, we already have the advantage of a different execution context and thus a different security context. A website cannot access data inside an extension or influence code running inside an extension. This also protects the policies, which are stored within the extension. Therefore, there is no possibility for a malicious website to modify or inject new policies.

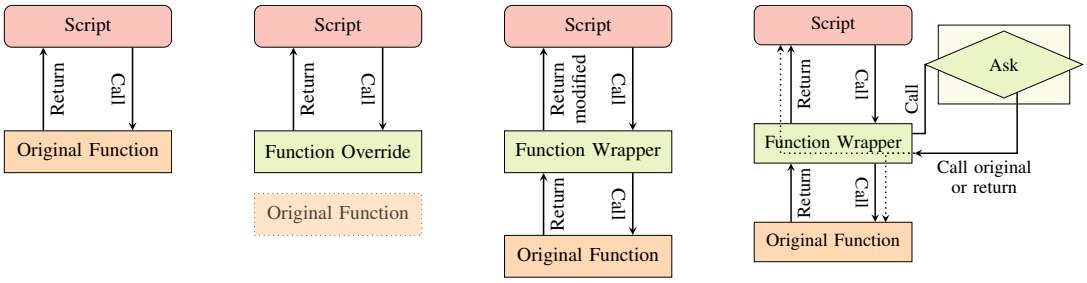
C. User Interface

Challenge C3 is to have no significant impact on compatibility and user experience. This implies that *Chrome Zero* must not have a perceivable performance impact (cf. Section VI).

As diverting the control flow into the extension (cf. Figure 2) is relatively costly, we only do that if absolutely necessary. Figure 4 shows how *Chrome Zero* only diverts the control flow to the extension if a policy requires that the user is asked for permission. In all other cases, we can directly replace the function with a stub or wrapper function (Figure 4b and Figure 4c) before loading a page.

As JavaScript does not provide a mechanism to block scripts, except for the built-in pop-up boxes (e.g., `alert`), pausing a function to ask the user for permission requires interaction with the browser extension. *Chrome Zero* relies on the Google Chrome Debugger API [12] which extends the functionality of JavaScript to influence and inspect the internal state of the JavaScript engine. Using Chrome's remote debugging protocol [11], *Chrome Zero* registers a function which is called whenever a script uses the `debugger` keyword. The effect of the `debugger` keyword is that the JavaScript engine pauses all currently executing scripts before calling the registered function [27].

While the script—and thus the entire page—is paused, *Chrome Zero* asks the user for permission to execute the current function. The result is then returned to the calling function by writing it to a local variable within the closure, and function execution is resumed using the Debugger API. Note that only *Chrome Zero* can access the local variable that stores the result, as all variables within the closure are inaccessible to the remaining page (cf. Section V-B). The function then either resumes execution of the function, or returns a default value in



(a) Normal function (b) Blocked function (c) Modified function (d) Function with user permission

Fig. 4: (a) A normal, unmodified function call as reference. (b) If a function is blocked, it can be immediately replaced with a function returning the default value. (c) If the return value has to be modified, the function can be replaced by an anonymous JavaScript closure which applies the modification directly on the page. (d) Only if the user has to be asked for permission, a switch into the extension context is necessary.

case the user does not give permission to execute the function. Spurious usage of the `debugger` keyword on a (malicious) website has no effect, as *Chrome Zero* just continues if no policy is found for the current function.

Chrome Zero does not instrument the existing browser permission system, as it cannot be retrofitted to protect arbitrary functions and objects. The existing browser permission system only works for APIs designed to be used with the permission system, *i.e.*, the API has to be asynchronous by relying on callback functions or promises. The browser asks the user for permission, and if the user accepts, the browser calls a callback function with the result, e.g., the current geolocation. Hence, synchronous APIs, e.g., the result of the function call is provided as a return value, cannot be protected with the browser's asynchronous permission system. For the protection to be complete, we have to handle both synchronous as well as asynchronous function calls, and can therefore not rely on the browser's internal permission system.

VI. Security Evaluation

In this section, we evaluate *JavaScript Zero* by means of our proof-of-concept Chrome extension, *Chrome Zero*. In the first part of the evaluation, we show how *Chrome Zero* prevents all microarchitectural and side-channel attacks that can be mounted from JavaScript (cf. Table I). Furthermore, we show how policies can prevent exploits. We evaluate how many exploits are automatically prevented by protecting users against microarchitectural and side-channel attacks.

A. Microarchitectural and Side-Channel Attacks

To successfully prevent microarchitectural and side-channel attacks, we have to eliminate the requirements identified in Section II-B. Depending on the requirements we eliminate, microarchitectural and side-channel attacks are not possible anymore (cf. Table I). Consequently, we discuss policies to eliminate each requirement. Table II shows a summary of all policies and how they affect state-of-the-art attacks. Table III shows which policy is active on which protection level.

1) Memory Addresses: In all known attacks, array buffers are used to retrieve information on the underlying memory address. An attacker can exploit that array buffers are page-aligned to learn the least significant 12 bits of both the virtual and physical address [33]. Thus, we have to ensure that array buffers are not page-aligned and that an attacker does not know the offset of the array buffer within the page.

Array buffers are raw binary data buffers, storing values of arrays. However, their content cannot be accessed directly, but only using typed arrays or `DataView`s. Thus, we have to proxy both the `DataView` object as well as all typed arrays (e.g., `Uint8Array`, `Uint16Array`, etc.).

a) Buffer ASLR: To prevent the arrays from being page-aligned, we introduce buffer ASLR, which randomizes the start of the array buffer. We overwrite the length argument of the constructor to allocate additional $4\,\mathrm{KB}$. This allows us to move the start of the array anywhere on the page by generating a random offset in the range $[0; 4096)$. This offset is then added to the array index for every access. Hence, all data is shifted, *i.e.*, the value at index 0 is not page-aligned but starts at a random position within the page. Thus, an attacker cannot rely on the property anymore that the 12 least significant address bits of the first array buffer index are '0'.

b) Preloading: However, the protection given by buffer ASLR is not complete, as an attacker can still iterate over a large array to detect page borders using page faults [15], [40]. With 21 bits of the virtual and physical address, a THP page border contains even more information for an attacker. One simple prevention for this attack is to iterate through the array after constructing it. Accessing all underlying pages triggers a page fault for every page, and an attacker subsequently cannot learn anything from iterating over the array, as the memory is already mapped. Thus, Rowhammer.js [15] and the DRAM covert channel [40] are prevented by *Chrome Zero*.

c) Non-determinism: Instead of preloading, (*i.e.*, iterating over the array after construction), we can modify the setter of the array to add a memory access to a random array index for every access to the array. This has two advantages in terms of security. First, with only preloading, an attacker could wait for the pages to be swapped out, or deduplicated, enabling

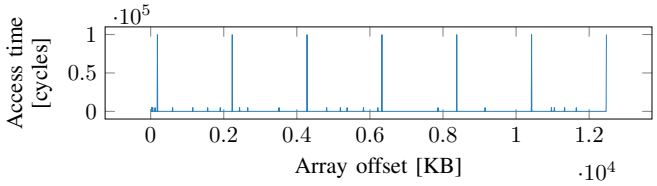
TABLE II: ALL DISCUSSED POLICIES (EXCEPT FOR SENSORS) AND THEIR EFFECT ON ATTACKS.

Policy \ Prevents	Rowham-mer.js [15]	Page Dedu-plication [14]	DRAM Covert Channel [40]	Anti-ASLR [13]	Cache Eviction [33], [15], [40], [13]	Keystroke Timing [47], [20]	Browser [44], [45], [47]	Exploits (cf. Section VI-B)
Buffer ASLR	○	◐	○	●	●	○	○	◐
Array preloading	●	○	●	○	○	○	○	○
Non-deterministic array	●	◐	◐	●	●	○	○	○
Array index randomization	○	●	○	●	○	○	○	◐
Low-resolution timestamp	○	◐	○	○	○	◐	◐	○
Fuzzy time	○	◐*	○	○*	○	●*	●*	○
`WebWorker` polyfill	○	○	●	●	●	●	○	○
Message delay	○	○	○	○	○	◐	◐	○
Slow `SharedArrayBuffer`	○	○	●	◐	●	○	○	○
No `SharedArrayBuffer`	○	○*	●	●*	●	○*	○*	◐
Summary	●	●	●	●	●	●	●	◐

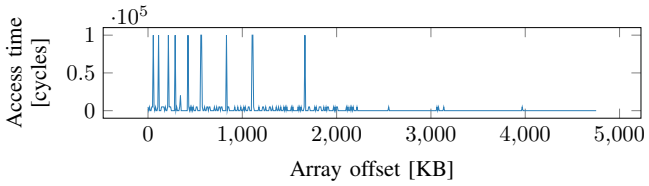
Symbols indicate whether a policy fully prevents an attack, (●), partly prevents and attack by making it more difficult (◐), or does not prevent an attack (○).
A star (*) indicates that all policies marked with a star must be combined to prevent an attack.

TABLE III: A TABLE OF HOW POLICIES CORRESPOND TO THE PROTECTION LEVELS OF *Chrome Zero*.

Requirement \ Protection Level	Off	Low	Medium	High	Paranoid
Memory addresses	-	Buffer ASLR	Array preloading	Non-deterministic array	Array index randomization
Accurate Timing	-	Ask	Low-resolution timestamp	Fuzzy time	Disable
Multithreading	-	-	Message delay	`WebWorker` polyfill	Disable
Shared data	-	-	Slow `SharedArrayBuffer`	Disable	Disable
Sensor API	-	-	Ask	Fixed value	Disable

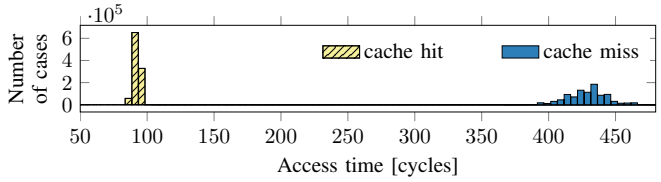


(a) Page border detection without random accesses.

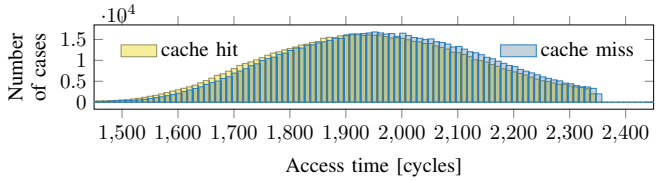


(b) Page border detection with random accesses.

Fig. 5: Page border detection without and with *Chrome Zero*. When iterating over an array, page faults cause a higher timing than normal accesses, visible as timing peaks.



(a) Prime+Probe results without random accesses.



(b) Prime+Probe results with random accesses.

Fig. 6: When adding random accesses, the timings for cache hits and misses blend together, making it impractical to decide whether an access was a cache hit or a miss. In contrast to a benign use case, the access time is significantly increased as the adversary is priming (*i.e.*, thrashing) the cache and any memory access is likely a cache miss.

page border detection again. The random accesses prevent the page border detection, as an attacker cannot know whether the page fault was due to the regular access or due to a random access. As shown in Figure 5, with the random accesses, the probability to trigger a page fault for the first accesses is relatively high, as pages are not mapped in the beginning. This probability decreases until all pages are mapped. Thus, an attacker cannot reliably detect the actual border of a page but only the number of pages. Second, this prevents the generation of eviction sets [33], [15], [40], [13]. A successful eviction of a cache set requires an attacker to measure the access time of a special memory access pattern [15]. Adding random accesses in between prevents an attacker from obtaining accurate measurements, as the random accesses influence the overall timing (cf. Figure 6). Note that the access time is significantly increased as the adversary is priming (*i.e.*, thrashing) the cache and thus, any additional memory access is likely a cache miss.

Hence, this does not relate to any benign use case performance or access time.

d) Array Index Randomization: One attack that cannot be thwarted with the discussed policies is the page-deduplication attack [14]. In this attack, an attacker only has to be in control over the content of one page to deduce if a page with the same content already exists in memory. Allocating a large array still gives an attacker control over at least one page. To prevent a page deduplication attack from being successful, we have to ensure that an attacker cannot deterministically choose the content of an entire page. One possible implementation is to make the mapping between the array index and the underlying memory unpredictable by using a random linear function.

We overwrite the array to access memory location $f(x)$ when accessing index x with $f(x) = ax + b \mod n$ where a and b are randomly chosen and n is the size of the `ArrayBuffer`. Furthermore, a and n must be co-prime to generate a unique mapping from indices to memory locations. To find a suitable a, we can simply choose a random a in the array constructor and test whether $\gcd(a, n) = 1$. As the probability of two randomly chosen numbers to be co-prime is $\frac{1}{\zeta(2)} = \frac{6}{\pi^2} \approx 61\%$ (for $n \to \infty$) [30], this approach is computationally efficient. That is, the expected value is 1.64 random choices to find two co-prime numbers (for $n \to \infty$).

As a and b are inaccessible to an attacker, the mapping ensures that an attacker cannot predict how data is stored in physical memory. An attacker can also not reverse a and b because there is no way to tell whether a guess was correct, as buffer ASLR and preloading prevent any reliable conclusions from page faults. Thus, an attacker cannot control the content of a page, thwarting page deduplication attacks [14].

2) Accurate Timing: Many attacks within the browser require highly accurate time measurements. Especially microarchitectural attacks require time measurements with a resolution in the range of nanoseconds [33], [15], [40], [13]. Such high-resolution timestamps were readily available in browsers [33] but have been replaced by lower resolution timestamps in all modern browsers in 2015 [2], [6], [9], [25]. However, Schwarz et al. [40] showed that it is still possible to get a nanosecond resolution from these timestamps by exploiting the underlying high-resolution clock.

a) Low-resolution timestamps: As a policy, we can simply round the result of the high-resolution timestamp (`window.performance.now`) to a multiple of $100\,\mathrm{ms}$. This is exactly the same behavior as implemented in the Tor browser. Thus, we achieve the same protection as the Tor browser, where the recoverable resolution is only $15\,\mu\mathrm{s}$, rendering it useless for many attacks [40].

b) Fuzzy time: A different approach is to apply fuzzy time to timers [46]. In addition to rounding the timestamp, fuzzy time adds random noise to the timestamp to prevent exact timing measurements but still guarantees that the timestamps are monotonically increasing. Kohlbrenner et al. [18] implemented this concept in Fuzzyfox, a Firefox fork. We can achieve the same results using a simple policy that implements a variant of the algorithm proposed by Vattikonda et al. [46] shown in Listing 4, without requiring constant maintenance of a browser fork.

```
(function() {
var wpn = window.performance.now, last = 0;
window.performance.now = function() {
var fuzz = Math.floor(Math.random() * 1000), //1ms
    now = Math.floor(wpn.call(window.performance)*1000);
var t = now - now % fuzz;
if(t > last) last = t;
return last / 1000.0;
};})();
```

Listing 4: Fuzzy time [46] applied to the high-resolution timing API with a $1\,\mathrm{ms}$ randomization.

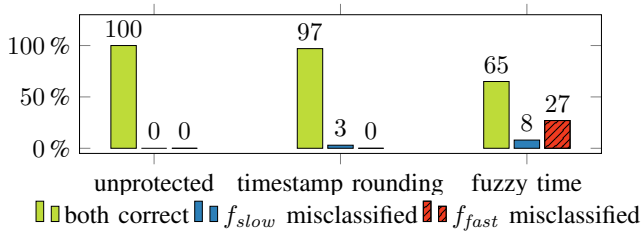


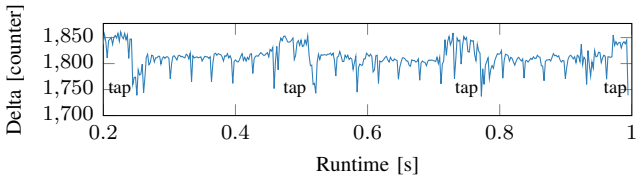
Fig. 7: Edge thresholding to distinguish whether the function f_{slow} takes longer than f_{fast}. The difference between the execution times is less then the provided resolution.

We evaluated our policies for low-resolution timestamps and fuzzy time by creating two functions with a runtime below the resolution of the protected high-resolution timer. Using edge thresholding [40], we tried to distinguish the two functions based on their runtime. For the evaluation, we rounded timestamps to a multiple of $1\,\mathrm{ms}$ and used a $1\,\mathrm{ms}$ randomization interval for the fuzzy time. The two functions f_{slow} and f_{fast}, which we distinguish, have an execution time difference of $300\,\mu\mathrm{s}$. Figure 7 shows the results of this evaluation. If no policy is applied to the high-resolution timer, the functions can always be distinguished based on their runtime. With the low-resolution timestamp and edge thresholding, the functions are correctly distinguished in $97\,\%$ of the cases, as the underlying clock still has a resolution in the range of nanoseconds. When fuzzy time is enabled, the functions are correctly distinguished in only $65\,\%$ of the cases, and worse, in $27\,\%$ of the cases the functions are wrongly classified, *i.e.*, the faster-executing function is classified as the slower function.

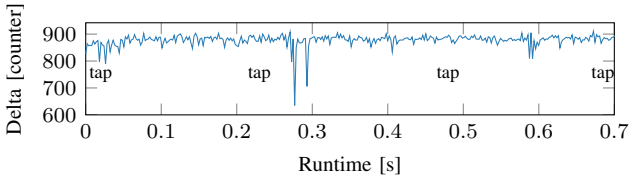
Figure 8 shows the result of fuzzy time on the JavaScript keystroke detection attack by Lipp et al. [20]. Without fuzzy time, it can be clearly seen whenever the user taps on the touch screen of a mobile device (Figure 8a). By enabling the fuzzy time policy, the attack is fully prevented, and no taps can be seen in the trace anymore (Figure 8b).

3) Multithreading: As the resolution of the built-in high-resolution timer has been reduced by all major browsers [2], [6], [9], [25], alternative timing primitives have been found [18], [40], [13]. Although several new timing primitives work without multithreading [18], [40], only the timers using multithreading achieve a resolution that is high enough to mount microarchitectural attacks [13], [40], [47], [20].

a) `WebWorker` polyfill: A drastic—but effective—policy is to prevent real parallelism. To achieve this, we can

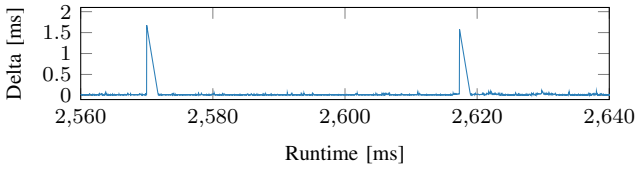


(a) Without *Chrome Zero*.

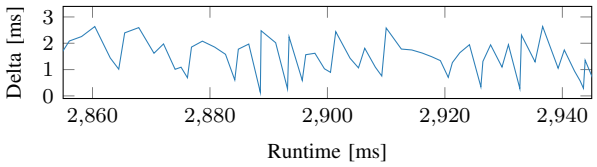


(b) With *Chrome Zero*.

Fig. 8: Without *Chrome Zero*, taps can be clearly seen in the attack by Lipp et al. [20] (Figure 8a). With *Chrome Zero*, the attack is prevented and no taps are visible (Figure 8b).



(a) Without *Chrome Zero*.

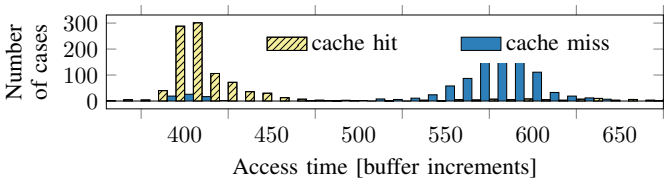


(b) With *Chrome Zero*.

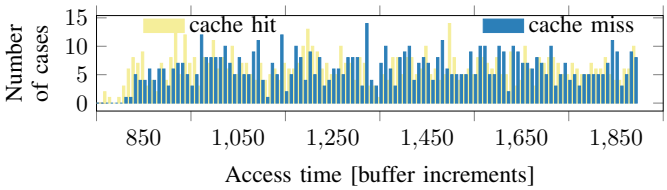
Fig. 9: Running the attack by Vila et al. [47] shows keystrokes among other system and browser activity (Figure 9a). With *Chrome Zero* in place, the `postMessage` timings are delayed and thus keystrokes cannot be detected anymore (Figure 9b).

completely replace `WebWorkers` by a polyfill intended for unsupported browsers. The polyfill [29] simulates `WebWorkers` on the main thread, trying to achieve similar functionality without support for real parallelism. Thus, all attacks relying on real parallelism [13], [40], [47], [20] do not work anymore.

b) Message delay: A different policy to specifically prevent certain timing primitives [40] and attacks on the browser's rendering queue [47] is to delay the `postMessage` function. If the `postMessage` function randomly delays messages (similar to Fuzzyfox [18]), the attack presented by Vila et al. [47] does not work anymore, as shown in Figure 9.



(a) Without *Chrome Zero*.



(b) With *Chrome Zero*.

Fig. 10: Using `SharedArrayBuffer` in combination with web worker to build a high-resolution timing primitive as proposed by Gras et al. [13] and Schwarz et al. [40]. Without *Chrome Zero*, cache hits and misses are clearly distinguishable (Figure 10a). Configuring *Chrome Zero* to delay accesses to the `SharedArrayBuffer` leads to a uniform distribution in timings, thwarting the attacks (Figure 10b).

4) Shared Data: `SharedArrayBuffer` is the only data type which can be shared across multiple workers in JavaScript. This shared array can then be abused to build a high-resolution timer. One worker periodically increments the value stored in the shared array while the main thread uses the value as a timestamp. This technique is the most accurate timing primitive at the time of writing, creating a timer with a resolution in the range of nanoseconds [13], [40].

a) No `SharedArrayBuffer`: At the time of writing, the `SharedArrayBuffer` is by default deactivated in modern browsers. Thus, websites should not rely on this functionality anyway, and a policy can simply keep the `SharedArrayBuffer` disabled if vendors enable it.

b) Slow `SharedArrayBuffer`: On our Intel i5-6200U test machine, we achieve a resolution of 0.77 ± 0.01 ns with the `SharedArrayBuffer` timing primitive. This allows use to clearly distinguish cache hits from cache misses (Figure 10a), and thus mount the attacks proposed by Schwarz et al. [40] as well as Gras et al. [13]. To protect users from these attacks, our policy randomly delays the access to the `SharedArrayBuffer`. Using this policy, we reduce the resolution to 4215.25 ± 69.39 ns, which is in the same range as the resolution of the native `performance.now()` timer. Thus, these microarchitectural attacks, which require a high-resolution timer, do not work anymore as shown in Figure 10b.

5) Sensor API: As mobile devices are equipped with many sensors, JavaScript provides several APIs allowing websites to access sensor data, e.g., accelerometer, ambient light, or battery status. While some of those interfaces allow developers

to build more functional and user-friendly applications, they also facilitate leakage of sensitive information. While modern browsers explicitly ask the user for permission if the running websites want to access the user's geolocation, access to other APIs is silently permitted.

a) Battery Status API: After Olejnik et al. [32] showed the potential privacy risk of the HTML5 Battery Status API as a tracking identifier, Firefox disabled the interface with version 52 [8]. However, Chrome still offers unrestricted access to this API without asking for permission. Thus, we introduce a policy allowing to either randomize the properties of the battery interface, to set them to fixed values, or to disable the interface entirely. With this policy, the Battery Status API can not be used as a tracking identifier anymore.

b) Ambient Light Sensor: The ambient light sensor can be used to infer user PINs [42] or to recover browsing history information [31]. While the API needs to be enabled manually in the Chrome browser, it is enabled by default in the Firefox browser. By introducing a policy that either returns constant values for the ambient light sensor, or disables the interface, an attacker is unable to perform these attacks.

c) Device Motion, Orientation, and Acceleration: The motion sensor data and orientation sensor data of mobile web browsers can be exploited to infer user PIN input [23]. Both are available to websites without any permissions. In order to mitigate such attacks, we introduce a policy that allows to spoof the sensor data or to prohibit access entirely.

B. Exploits

Although exploits are out-of-scope for the permission system (cf. Section II-C), we investigate the (side-)effect of our policies on JavaScript exploits. For this, we investigate CVEs that are exploitable via JavaScript and were discovered since Chrome 49, as *Chrome Zero* requires Chrome 49 and later.

To evaluate whether *Chrome Zero* protects a user from a specific exploit, we first reproduce the exploit without *Chrome Zero* and then activate *Chrome Zero* to check whether the exploit still works. We reproduced all 12 CVEs[2] for the Chrome JavaScript engine, which were discovered since 2016 for Chrome 49 or later and for which we could find proof-of-concept implementations online. All of the 12 CVEs lead to either a crash of the current tab or to information leakage.

With *Chrome Zero* in place, half of them are prevented, leaving only 6 CVEs that are still exploitable. The prevented CVEs all rely on at least one object which we modify (e.g., `ArrayBuffer`) and thus do not work with the modified object. Furthermore, we expect that actual remote code execution using the working exploits gets more complicated if policies such as array index randomization or buffer ASLR are in place. Thus, *Chrome Zero* provides additional protection against 0-days without requiring explicit policies. Creating policies to specifically target CVEs is left to future research.

VII. Usability Evaluation

In this section, we analyze the usability impact of *Chrome Zero* by performing a performance analysis and a double-blind user study. We first analyze how many websites use functionality which is also used in microarchitectural and side-channel attacks. We then analyze the performance impact on the Alexa Top 10 websites. Finally, we show whether the protection mechanism has any impact on the browsing experience, *i.e.*, whether there are pages that do not work as expected anymore, for the Alexa Top 25 websites.

[2]CVE-2016-1646, 1653, 1665, 1669, 1677, 5129, 5172, 5198, 5200, 9651, 2017-5030, 5053

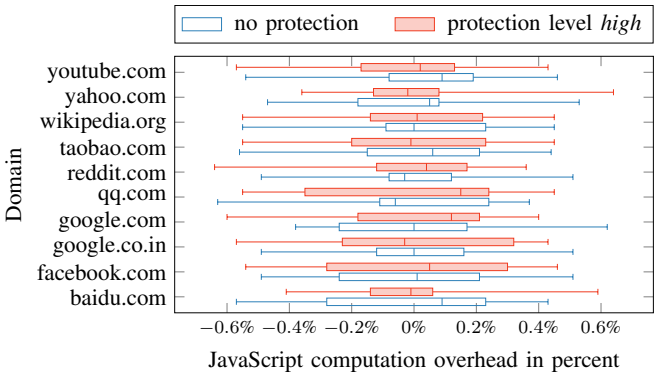


Fig. 11: The computation overhead of the JavaScript engine while loading each of the Alexa Top 10 websites without protection and with protection level *high*. The performance overhead is negligible for most websites.

TABLE IV: Results of the JetStream benchmark.

	Without *Chrome Zero*	With *Chrome Zero*
Latency	71.46 ± 4.43	71.33 ± 2.43
Throughput	220.45 ± 6.80	214.71 ± 3.50
Total	134.90 ± 5.96	132.81 ± 2.92

The higher the score, the better the performance.

A. Performance

We evaluated the performance overhead of *Chrome Zero* in both micro and macro benchmarks.

First, we evaluated the impact of *Chrome Zero* on the loading time of a page. We measured a page loading latency between 10.64 ms if no policy is active, and 89.08 ms if policies protecting against all microarchitectural and side-channel attacks are active. As on every page load the current policies are loaded and injected into the current tab, the latency grows linearly with the number of policies, and delays the actual rendering of the page. On average, we measured a latency of approximately 3.4 ms per active policy, *i.e.*, every policy delays the loading of a newly opened page by 3.4 ms.

Second, we investigated the overhead for Chrome's JavaScript engine by using the internal profiler. Figure 11 shows the overhead for the Alexa Top 10 websites. The runtime increase of the JavaScript engine had a median of 1.82 %, which corresponds to only 16 ms.

Finally, we used the JetStream [3] browser benchmark, which is developed as part of the WebKit engine. We measure a performance overhead of 1.54 % when using *Chrome Zero*. Table IV shows the detailed scores of the benchmark.

A reason for the low overhead of *Chrome Zero* is the JavaScript Just-In-Time (JIT) compiler. Chrome's JIT consists of several compilers, producing code on different optimization levels [39]. As the code is continuously optimized by the JIT, our injected functions are compiled to highly efficient native code, with a negligible performance difference compared to the original native functions. The results of our benchmarks show that *Chrome Zero* does not have a visible impact on the user experience for an everyday usage.

B. Compatibility

For *Chrome Zero* to be usable on a day-to-day basis, it is important that the majority of websites is still usable if policies are active. We analyzed the Alexa Top 100 websites with a protection level of *high*, the second highest protection level (cf. Table III). Out of the 100 pages, all of them used JavaScript, and 57 relied on functions for which the protection level *high* defines policies. For all these pages, we verified that *Chrome Zero* did not cause any error when testing some of the site's basic functionality. For a thorough evaluation, we conducted a double-blind user study to test whether *Chrome Zero* has an impact on the browsing experience. We designed the study to have 24 participants to have a maximum standard error below 15 % at a confidence level of 85 %. The 24 participants, which we recruited by advertising it through word-of-mouth, had different backgrounds, ranging from students without any IT background to information-security post-doctoral researchers.

1) Method: We explained every participant that we developed a browser extension which provides additional protection against attacks. We showed two instances of Google Chrome to the participant, one without *Chrome Zero* (A) and one with *Chrome Zero* set to protection level *high* (B) for every website in the Alexa Top 25. For every website, a fully automated script randomly chose whether browser instance A or B had *Chrome Zero* activated, without any interaction of the study conductor or the study participant. Hence, neither the study participant nor the study conductor knew which of the two browsers had *Chrome Zero* activated, making the study double blind. After 1 minute, the script asked the user whether there was any noticeable difference between the two pages, and if so, whether browser A or browser B had *Chrome Zero* enabled. The results of these questions were saved in a file and automatically evaluated after the user tested all 25 pages. Every correct user answer was counted as a 100 % correct answer, if the user did not notice any difference, we counted it as a 50% probability to make the correct guess, and if the user answered incorrectly, we counted a 0 % correct answer.

2) Results: Figure 12 shows the results of our user study. The overall probability to correctly detect the presence of *Chrome Zero* was 50.2 %. The maximum average success rate of a participant was 60 %; the minimum was 40 %.

The maximum detection rate for a website was 62.5 % for yahoo.com (standard error ±0.05). For all other websites, the detection rate was not better than random guessing. The minimum detection rate for a website was 41.7 % for amazon.com and sina.com.cn (standard error ±0.05).

For the 6 websites where no *Chrome Zero* policy was active, users guessed correctly in 48.3 % on average (standard error ±0.049), *i.e.*, a deviation of 1.7*pp* to random guessing.

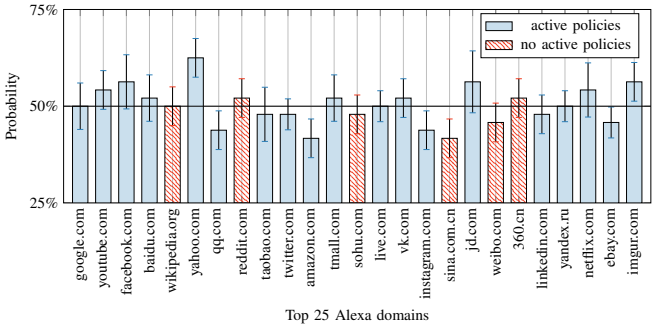


Fig. 12: Probability that a user correctly identifies the browser with *Chrome Zero* and the corresponding standard error. Only for one website (yahoo.com) the users had a chance of identifying the browser with *Chrome Zero* that was clearly above the random guessing probability.

For the 19 websites where at least one *Chrome Zero* policy was active, users guessed correctly in 50.8 % on average (standard error ±0.055), *i.e.*, a deviation of 0.8*pp* to random guessing. This highlights how negligible the differences in the user experience are.

While their classification of the instance was many times incorrect, participants stated loading time, cookie-policy dialogues and website redirections as the reason for selecting the instance as the one using *Chrome Zero*.

Although our implementation is only a proof-of-concept implementation, the results of the study confirm that *JavaScript Zero* is practical, and our implementation of *Chrome Zero* is usable on a day-to-day basis.

VIII. Related Work

In this section, we discuss related work on protecting users from the execution of potential harmful JavaScript code. While there are several proposed solutions, *JavaScript Zero* is the only technique fully implemented as a browser extension only (*Chrome Zero*) without negatively affecting the browsing experience. *Chrome Zero* does not rely on any changes to existing source code or the system's environment and thus does not require support by developers or browser vendors.

a) Browser extensions: Browser extensions such as NoScript [10] or uBlock [37] allow users to define policies to permit or prohibit the execution of JavaScript depending on their origin, *i.e.*, a page can either completely block JavaScript or execute it without any restrictions. In contrast, *Chrome Zero* offers a more fine-grained permission model that operates on function level and does not interfere with dynamic website content. Furthermore, *JavaScript Zero* directly targets attack prevention, whereas existing browser extensions aim primarily at blocking advertisements and third-party tracking code.

In concurrent work, Snyder et al. [41] proposed a browser extension to protect against exploits in general, based on the same generic idea as *JavaScript Zero*. They first exhaustively evaluate the usage statistics of JavaScript APIs and their connection to CVEs and then, similar to our approach, selectively block the corresponding JavaScript APIs. Based

on this approach they block 52% of all CVEs, while only impacting the usability of 4% to 7% of the tested websites. When a usability impact on 16% of the tested websites is still acceptable, they can even block 72% of all CVEs. This is a significantly lower usability impact than previous approaches like NoScript [10] or uBlock [37]. With our focus on mitigating microarchitectural and side-channel attacks, we complement the work by Snyder et al. [41] and show that the underlying generic idea is not only applicable to CVEs or side-channel attacks, but to both types of attacks. This highlights the strength of the underlying idea of both papers.

b) Modified browsers: Meyerovich et al. [24] modified the JavaScript engine of Internet Explorer 8 to enforce fine-grained application-specific runtime security policies by the website developer. In contrast, *JavaScript Zero* is implemented as a browser extension and does not rely on any developer to define security policies. Patil et al. [35] analyzed the access control requirements in modern web browsers and proposed JCShadow, a fine-grained access control mechanism in Firefox. JCShadow splits the running JavaScript into groups with an assigned isolated copy of the JavaScript context. A security policy then defines which code is allowed to access objects in other shadow contexts to separate untrusted third-party JavaScript code from the website. Stefan et al. [43] proposed COWL, a label-based mandatory access control system to sandbox third-party scripts. Bichhawat et al. [5] proposed WebPol, a fine-grained policy framework to define the aspects of an element accessible by third-party domains by exposing new native APIs. All these approaches assume a benign website developer protecting their website from untrusted—and possibly malicious—third-party libraries trying to manipulate their website. In contrast, *JavaScript Zero* does not make any assumptions in this direction. Any website or library developer may be malicious, trying to attack the user. *JavaScript Zero* neither relies on website developers nor requires any modifications of the browser or the JavaScript engine.

Kohlbrenner et al. [18] proposed Fuzzyfox, a modified version of Firefox that mediates all timing sources by degrading the resolution of explicit timers and implicit clocks to $100\,\mathrm{ms}$. In contrast to Fuzzyfox, *JavaScript Zero* successfully prevents not only timing attacks but also attacks which do not require high-resolution timing measurements. Mao et al. [21] studied timing-based probing attacks that indirectly infer sensitive information from the website. Their tool only allows to identify malicious operations performing timing-based probing attacks based on generalized patterns, e.g., the frequency of timing API usage. *JavaScript Zero* directly prevents the attack by either disallowing timers or making them too coarse-grained.

c) Code rewriting: Reis et al. [38] implemented BrowserShield, a service that automatically rewrites websites and embedded JavaScript to apply run-time checks to filter known vulnerabilities. Yu et al. [56] proposed to automatically rewrite untrusted JavaScript code through a web proxy, in order to ask the user how to proceed on possible dangerous behavior, e.g., opening many pop-ups or cookie exfiltration attacks. Their model only covers policies with respect to opening URLs, windows, and cookie accesses, and does not protect against side-channel attacks. Moreover, *JavaScript Zero* does neither rewrite any existing code nor rely on any possibly platform-dependent service such as a web proxy.

d) JavaScript frameworks: Agten et al. [1] presented JSand, a client-side JavaScript sandboxing framework that enforces a server-specified policy to jail included third-party libraries. Phung et al. [36] proposed to modify code in order to protect it from inappropriate behavior of third-party libraries. Their implementation requires website developers to manually add protection code to their website. However, their protection does not apply to scripts loaded in a new context, *i.e.*, with `<frame>`, `<iframe>`, or refresh directives. Guan et al. [16] studied the privacy implications of the HTML5 `postMessage` function and developed a policy-based framework to restrict unintended cross-origin messages. As our countermeasure is implemented solely as a browser extension, it does not rely on any website developer to use a certain library or to apply any changes to the code.

IX. Conclusion

In this paper, we presented *JavaScript Zero*, a highly practical and generic fine-grained permission model in JavaScript to reduce the attack surface in modern browsers. *JavaScript Zero* leverages advanced JavaScript language features, such as virtual machine layering, closures, proxy objects, and object freezing, for security and privacy. Hence, *JavaScript Zero* is fully transparent to website developers and users and even works with obfuscated code and higher-order scripts. Our proof-of-concept Google Chrome extension, *Chrome Zero*, successfully protects against 11 unfixed state-of-the-art microarchitectural and side-channel attacks. As a side effect, *Chrome Zero* successfully protects against $50\,\%$ of the published JavaScript 0-day exploits since Chrome 49. *Chrome Zero* has a low-performance overhead of only 1.82% on average. In a double-blind user study, we found that for 24 websites in the Alexa Top 25, users could not distinguish browsers with and without *Chrome Zero* correctly, showing that *Chrome Zero* has no perceivable (negative) effect on most websites. Our work shows that transparent low-overhead defenses against JavaScript-based state-of-the-art microarchitectural attacks and side-channel attacks are practical.

Acknowledgment

We would like to thank our anonymous reviewers for their valuable feedback and our study participants for their time. This work has been supported by the Austrian Research Promotion Agency (FFG), the Styrian Business Promotion Agency (SFG), the Carinthian Economic Promotion Fund (KWF) under grant number 862235 (DeSSnet) and has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

References

[1] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "Jsand: complete client-side sandboxing of third-party javascript without browser modifications," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[2] Alex Christensen, "Reduce resolution of performance.now." 2015. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=146531

[3] Apple, "JetStream 1.1," Aug. 2017. [Online]. Available: http://browserbench.org/JetStream

[4] D. J. Bernstein, "Cache-Timing Attacks on AES," 2004. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[5] A. Bichhawat, V. Rajani, J. Jain, D. Garg, and C. Hammer, "Webpol: Fine-grained information flow policies for web browsers," in *ESORICS'17*, 2017, (to appear).

[6] Boris Zbarsky, "Reduce resolution of performance.now." 2015. [Online]. Available: https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab

[7] L. Cai and H. Chen, "TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion," in *USENIX Workshop on Hot Topics in Security – HotSec*, 2011.

[8] Chris Peterson, "Bug 1313580: Remove web content access to Battery API," 2016. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1313580

[9] Chromium, "window.performance.now does not support sub-millisecond precision on Windows," 2015. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110

[10] Giorgio Maone, "NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!" Jul. 2017. [Online]. Available: https://noscript.net

[11] Google, "Chrome DevTools Protocol Viewer," 2017. [Online]. Available: https://developer.chrome.com/devtools/docs/debugger-protocol

[12] ——, "chrome.debugger," 2017. [Online]. Available: https://developer.chrome.com/extensions/debugger

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

[14] D. Gruss, D. Bidner, and S. Mangard, "Practical memory deduplication attacks in sandboxed javascript," in *ESORICS'15*, 2015.

[15] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA'16*, 2016.

[16] C. Guan, K. Sun, Z. Wang, and W. Zhu, "Privacy breach by exploiting postmessage in html5: Identification, evaluation, and countermeasure," in *ASIACCS'16*, 2016.

[17] P. C. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," in *CRYPTO'96*, 1996.

[18] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times," in *USENIX Security Symposium*, 2016.

[19] E. Lavoie, B. Dufour, and M. Feeley, "Portable and efficient run-time monitoring of javascript applications using virtual machine layering," in *European Conference on Object-Oriented Programming*, 2014.

[20] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard, "Practical Keystroke Timing Attacks in Sandboxed JavaScript," in *ESORICS'17*, 2017, (to appear).

[21] J. Mao, Y. Chen, F. Shi, Y. Jia, and Z. Liang, "Toward Exposing Timing-Based Probing Attacks in Web Applications," in *International Conference on Wireless Algorithms, Systems, and Applications*, 2016.

[22] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," in *NDSS'17*, 2017.

[23] M. Mehrnezhad, E. Toreini, S. F. Shahandashti, and F. Hao, "Touchsignatures: identification of user touch actions and pins based on mobile sensor data via javascript," *Journal of Information Security and Applications*, 2016.

[24] L. A. Meyerovich and B. Livshits, "Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser," in *S&P'10*, 2010.

[25] Mike Perry, "Bug 1517: Reduce precision of time for Javascript." 2015. [Online]. Available: https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517

[26] Mozilla Developer Network, "ArrayBuffer," 2017. [Online]. Available: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer

[27] ——, "debugger," 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/debugger

[28] ——, "Proxy," 2017. [Online]. Available: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Proxy

[29] Nolan Lawson, "A tiny and mostly spec-compliant WebWorker polyfill," Nov. 2016. [Online]. Available: https://github.com/nolanlawson/pseudo-worker

[30] J. Nymann, "On the probability that k positive integers are relatively prime," *Journal of Number Theory*, 1972.

[31] L. Olejnik, "Stealing sensitive browser data with the W3C Ambient Light Sensor API," 2017. [Online]. Available: https://blog.lukaszolejnik.com/stealing-sensitive-browser-data-with-the-w3c-ambient-light-sensor-api/

[32] L. Olejnik, G. Acar, C. Castelluccia, and C. Diaz, "The leaking battery," in *Revised Selected Papers of the 10th International Workshop on Data Privacy Management, and Security Assurance - Volume 9481*, 2016.

[33] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications," in *CCS'15*, 2015.

[34] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.

[35] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang, "Towards fine-grained access control in javascript contexts," in *31st International Conference on Distributed Computing Systems (ICDCS)*, 2011.

[36] P. H. Phung, D. Sands, and A. Chudnov, "Lightweight self-protecting javascript," in *ASIACCS'09*, 2009.

[37] Raymond Hill, "uBlock Origin - An efficient blocker for Chromium and Firefox. Fast and lean." Jul. 2017. [Online]. Available: https://github.com/gorhill/uBlock

[38] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: Vulnerability-driven filtering of dynamic html," in *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[39] Ross McIlroy, "Firing up the Ignition Interpreter," Aug. 2017. [Online]. Available: https://v8project.blogspot.co.at/2016/08/firing-up-ignition-interpreter.html

[40] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *FC'17*, 2017.

[41] P. Snyder, C. Taylor, and C. Kanich, "Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security," in *CCS'17*, 2017.

[42] R. Spreitzer, "Pin skimming: Exploiting the ambient-light sensor in mobile devices," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, 2014.

[43] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, "Protecting users by confining javascript with cowl," in *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

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

[45] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *CCS'15*, 2015.

[46] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in Xen," in *CCSW'11*, 2011.

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

[48] W3C, "Battery Status API," 2016. [Online]. Available: https://www.w3.org/TR/battery-status/

[49] ——, "Geolocation API Specification 2nd Edition," 2016. [Online]. Available: https://www.w3.org/TR/geolocation-API/

[50] ——, "Ambient Light Sensor," 2017. [Online]. Available: https://www.w3.org/TR/ambient-light/

[51] ——, "DeviceOrientation Event Specification," 2017. [Online]. Available: https://www.w3.org/TR/orientation-event/

[52] ——, "Generic Sensor API," 2017. [Online]. Available: https://www.w3.org/TR/2017/WD-generic-sensor-20170530/

[53] ——, "Javascript APIs Current Status," 2017. [Online]. Available: https://www.w3.org/standards/techs/js

[54] W3Techs, "Usage of JavaScript for websites," Aug. 2017. [Online]. Available: https://w3techs.com/technologies/details/cp-javascript/all/all

[55] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[56] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," in *ACM SIGPLAN Notices*, 2007.