JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits

Michael Schwarz, Florian Lackner, Daniel Gruss
Graz University of Technology
{michael.schwarz,daniel.gruss}@iaik.tugraz.at, florian.lackner@student.tugraz.at

***Abstract*—Today, more and more web browsers and extensions provide anonymity features to hide user details. Primarily used to evade tracking by websites and advertisements, these features are also used by criminals to prevent identification. Thus, not only tracking companies but also law-enforcement agencies have an interest in finding flaws which break these anonymity features. For instance, for targeted exploitation using zero days, it is essential to have as much information about the target as possible. A failed exploitation attempt, e.g., due to a wrongly guessed operating system, can burn the zero-day, effectively costing the attacker money. Also for side-channel attacks, it is of the utmost importance to know certain aspects of the victim's hardware configuration, e.g., the instruction-set architecture. Moreover, knowledge about specific environmental properties, such as the operating system, allows crafting more plausible dialogues for phishing attacks.**

In this paper, we present a fully automated approach to find subtle differences in browser engines caused by the environment. Furthermore, we present two new side-channel attacks on browser engines to detect the instruction-set architecture and the used memory allocator. Using these differences, we can deduce information about the system, both about the software as well as the hardware. As a result, we cannot only ease the creation of fingerprints, but we gain the advantage of having a more precise picture for targeted exploitation. Our approach allows automating the cumbersome manual search for such differences. We collect all data available to the JavaScript engine and build templates from these properties. If a property of such a template stays the same on one system but differs on a different system, we found an environment-dependent property.

We found environment-dependent properties in Firefox, Chrome, Edge, and mobile Tor, allowing us to reveal the underlying operating system, CPU architecture, used privacy-enhancing plugins, as well as exact browser version. We stress that our method should be used in the development of browsers and privacy extensions to automatically find flaws in the implementation.

I. Introduction

Today, more than half of the world's population is connected to the internet [35]. Regardless of whether people use websites from a computer or a smartphone, they require a web browser to do so. Most web browsers follow the standards defined by the World Wide Web Consortium (W3C), an international organization responsible for standards concerning the world wide web. Although the standards define many aspects of how websites are rendered and how they behave, they do not define everything on the implementation level.

As a consequence, implementation details differ significantly between different browsers. The differences can be found in supported standardized features, browser-specific features, as well as aspects which are undefined according to the standard [18]. With JavaScript, a scripting language supported by all modern browsers, websites can gather information about the concrete implementation of the browser. Furthermore, JavaScript allows to obtain details about the host system, e.g., the screen resolution, operating system, installed plugins. This can be used to adapt a website to the specific properties of a user's device and environment, providing an optimal user experience. However, the amount of information available to a website can also be abused to create a fingerprint consisting of a set of properties. Such a fingerprint can be used to uniquely identify a browser, and therefore a user, across multiple sessions and even across webpages [19], [51], [39].

Browsers aiming at the protection of the privacy of the user, such as the Tor browser, try to prevent fingerprinting. They do so by removing differences caused by the browser as well as the environment. They also block functionality such as Canvas elements [57]. There are also approaches to prevent fingerprinting by adding randomness instead of removing functionality [38]. The aim is always to prevent the creation of unique fingerprints of a browser and thus also user.

There are many legitimate reasons to prevent tracking and identification, and for certain groups, such as journalists or whistleblowers, it is in many cases even vital. However, for criminal actors, it is undoubtedly also beneficial to prevent tracking and unique identification. Thus, browsers such as the Tor browser are also heavily used for criminal activities [60], [17]. The anti-fingerprinting methods ensure that users cannot be tracked across websites, preventing deanonymization through the user's usage pattern of websites [57]. Thus, attackers trying to reveal the identity of such users cannot rely on simply tracking a user with fingerprinting.

However, an attacker does not necessarily want to uniquely identify a user for the purpose of tracking. For an attacker, it might be even more desirable to gather as much information about the environment as possible to mount a targeted attack [51]. Especially for nation-state actors or law-enforcement agencies, it can already be advantageous if only some information is known about a user. Information fragments can then be used to, e.g., link a suspect to a browser session, or mount a targeted exploit on the user.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23155
www.ndss-symposium.org

In this paper, we propose a method to automate the search for data leakage which reveals information about the user's environment. To automate the leakage detection, we build so-called templates over properties in different environments. A property can be anything which can be read by JavaScript. Multiple runs on one system reveal unstable properties, resulting in a deterministic set of static properties for a specific environment. We analyzed all unstable properties and show that in most cases they do not provide any reliable information about the environment. We also show how our method can be extended to the unstable properties that can be used for fingerprinting. The JavaScript property template we obtain allows us to match a specific target system to one of the environments in our template. Hence, an attacker can deduce what the environment of the target system is, and thus, which attacks can be mounted.

It is well known that law-enforcement agencies actively try to de-anonymize Tor users [61], [24], [16], [78]. Various exploits have already been used to do this, some of which were discovered later on by researchers. The exploits are usually zero-day exploits mainly targetting users with Windows operating system [78], [22]. However, exploits are not limited to zero-day exploits.

Nowadays, there is a repertoire of powerful, software-based side-channel attacks. These side-channel attacks exploit various microarchitectural elements, most prominently caches [56], [55], [33], [81], DRAM [58], or branch prediction [3], [2], [21]. Side-channel attacks are not only able to break cryptographic algorithms [37], [5], [62], but are even able to read arbitrary memory contents [43], [36], [74], [80], [8]. With powerful side-channel attacks, it is plausible that nation-state actors also use side-channel attacks to de-anonymize Tor users.

Although some side-channel attacks can be mounted directly from the browser [29], [30], [64], [26], [41], [25], [23], [36] or even remotely [71], [40], [65], many powerful side-channel attacks require native code execution. Both zero-day exploits, as well as side-channel attacks, require knowledge of the attacked system. Trying to use an attack for a system which is not affected might draw attention to the exploit, and worse, might even leak a zero-day to the public, rendering it useless for future attacks.

Hence, there is an arms race between browser vendors emphasizing on the privacy of the user (e.g., Tor), and attackers and tracking companies trying to learn as much about the system as possible. Attackers try to find new ways to leak information which browser vendors prevent as soon as they become public. This requires considerable effort on both sides. Thus, both parties have an interest in automating this approach. Automated leakage detection has already been used to detect leakage from the cache [32], memory accesses [79], procfs pseudo-file system [68], and Android API [69].

Our fully-automated approach we propose can replace the tedious work of identifying such properties manually. As it is easily integrated into the development and testing chain, it will allow providing strong guarantees for this security and privacy aspect of modern browsers.

Furthermore, we present two new side-channel attacks which can be mounted from JavaScript. They allow an attacker to reveal the instruction-set architecture and the memory allocator. Both properties are essential aspects of both side-channel attacks as well as traditional zero-day exploits.

Contributions. The contributions of this work are:

1) We are the first to propose a fully-automated method to identify browser properties which can be used for fingerprinting.
2) We show that we can deduce information about the host system even in browsers employing anti-fingerprinting techniques.
3) We present two new side-channel attacks in JavaScript to deduce further information about the host system.
4) We show that privacy-enhancing browser extension can leak more information than they disguise and can even be semi-automatically circumvented, leading to a false sense of security.

Outline. The remainder of the paper is organized as follows. In Section II, we provide background information on browser fingerprinting and side-channel attacks. In Section III, we present our fully-automated method to find leakage from browser properties. In Section IV, we present two novel side-channel attacks to deduce information about the environment. In Section V, we apply the method to real-world scenarios and discuss the detected properties which are useful for targeted attacks and fingerprinting. In Section VI, we analyze the coverage we reach with our approach. In Section VII, we discuss the limitations of the approach. We conclude in Section VIII.

II. Background and Related Work

In this section, we provide background about state-of-the-art browser fingerprinting and anti-fingerprinting mechanisms employed in current browsers. Furthermore, we also discuss related work which aims to automatically detect leakage in similar scenarios and give a short overview of side-channel attacks in JavaScript.

A. Browser Fingerprinting

Browser fingerprinting tries to uniquely identify a user across multiple webpages or visits to the same webpage without storing information in the browser. Thus, browser fingerprinting does not rely on classical tracking mechanisms such as cookies, making it hard for a user to prevent tracking.

Fingerprinting is usually done via a script which is executed when a user visits a website. This script collects several properties of the browser, such as the browser version, operating system, screen resolution, or installed plugins. While each of the properties itself does not allow tracking of a user, the combination of properties is unique enough to identify a user [19], [39].

There are many properties that can be used to fingerprint users. These properties include fonts [6], [51], plugins [51], rendering differences [70], [9], the battery status [53], and audio processing [20].

B. Anti-Fingerprinting Mechanisms

To prevent the tracking and identification of users, several software- and research projects try to minimize the fingerprinting surface. There are mainly two approaches to accomplish this goal.

First, some applications, such as, e.g., the Tor browser [57], hide the actual values of properties by either not exposing them or replacing them with the same value on all platforms. The Tor browser tries to prevent fingerprinting attempts which rely on browser properties that can be retrieved using JavaScript, plugins, or CSS [57]. Thus, the fingerprint of all Tor browsers in their default configuration is supposed to be the same.

There are also browser extensions for hiding values of properties as well as complete functionality that can be used for fingerprinting from a website. Such extensions include, e.g., Canvas Defender or the WebAPI Manager [66].

Second, some applications, such as, e.g., the FPRandom browser [38] or PriVaricator [50], try to break the stability of fingerprints by randomizing properties. FP-Block [72] spoofs properties in a way that they are the same for subsequent visits to one site, but differ between domains to prevent cross-domain tracking.

However, anti-fingerprinting mechanisms can be detected through additional, missing, or inconsistent values they create [46], [19], [1], [51].

C. Microarchitectural Attacks

Microarchitectural attacks have recently gained a lot of attention. Typically they are timing attacks that exploit the behavior of the microarchitecture, e.g., caches, branch predictors, or DRAM. The cache, in particular, was exploited in many attacks over the past years, leading to different attack techniques. Osvik et al. [55] described Evict+Time, where the attacker measures the influence of evicting a cache set on the runtime of an algorithm run by the victim, and Prime+Probe, where the attacker continuously measures whether the victim evicted a cache line in a specific cache set. Yuval and Falkner [81] described Flush+Reload, where the attacker continuously measures whether the victim reloaded a cache line. Several variations of these attacks were proposed, e.g., Flush+Flush [31], Evict+Reload [32], [42]. The recently discovered Meltdown [43], [74], [80], [8] and Spectre [36] attacks are significantly more powerful microarchitectural attacks. In some cases, they can infer values from arbitrary memory locations from other contexts, e.g., other processes or the operating system kernel.

D. Microarchitectural and Side-Channel Attacks in JavaScript

Although microarchitectural attacks exploit effects on a very low level of the CPU, they can even be exploited from JavaScript. In contrast to native code, JavaScript code is sandboxed and less powerful in terms of multithreading. Thus, there are several challenges an attacker has to overcome [63].

Still, many microarchitectural properties can be inferred from JavaScript [54], [29], [64], [41], [26], [25], [23], [36]. Moreover, sensors found on many mobile devices as well as modern browsers, introduce side channels which can be exploited from JavaScript [67], [44], [52]. It has also been shown that microarchitectual properties can be used for fingerprinting [46].

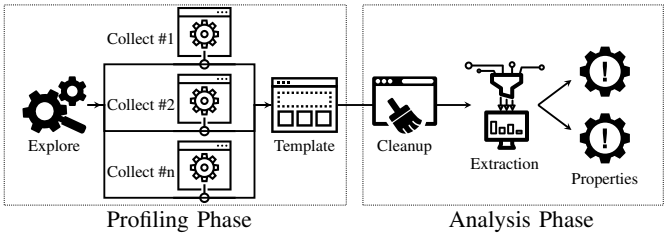

Fig. 1: A JavaScript Template Attack consists of two phases. In the profiling phase, all available properties of a browser are collected multiple times. In the analysis phase, the template is pruned by removing duplicates and changing values. The resulting identifed properties leak properties about certain aspects of the environment.

E. Template-based Leakage Detection

Chari et al. [10] introduced template attacks as a strong form of side-channel attacks. They first collect side-channel traces from an attacker-controlled device, the so-called template. Then, they collect a single trace from an identical device processing an unknown secret. The unknown secret can then be recovered by comparing the trace to the recorded templates.

Brumley and Hakala [7] applied template attacks to cache-based timing attacks. They rely on Prime+Probe to automatically detect and exploit cache leakage. However, their method is limited to an attacker who runs on the same CPU core as the victim. Gruss et al. [32] demonstrated a Flush+Reload-based template attack to detect and exploit cache leakage automatically. As their attack leverages the shared last-level cache, it does not rely on the attacker's ability to run on the same core as the victim. Weiser et al. [79] dynamically instrumented binaries to generate templates consisting of all memory access. By comparing templates for different secret inputs, they can automatically detect whether the binary contains secret-dependent memory accesses.

On a higher level, Spreitzer et al. used template attacks on Android to infer application launches and visited websites via the procfs pseudo-file system [68] as well as the Android API [69]. For both approaches, they first create a template by gathering all available information from the proc file system [68] or Android API [69]. In the analysis phase, they compare templates gathered from different applications to classify application launches and fingerprint websites based on the templates.

III. JavaScript Template Attacks

JavaScript Template Attacks can automatically identify language features of JavaScript that leak information about the environment, e.g., the operating system or hardware. For this purpose, they leverage the well-known concept of template attacks (cf. Section II-E) and apply it to JavaScript. As with all template attacks, JavaScript Template Attacks detect leakage through template differences caused by a secret. For

```
function getProperties(o) {
  var result = [];

  while(o !== null) {
    result = result.concat(Reflect.ownKeys(o));
    o = Object.getPrototypeOf(o);
  }
  return result;
}
```

Listing 1: Using reflections on all objects of the prototype chain results in a list of property names defined either directly in the object or inherited from an object on the prototype chain.

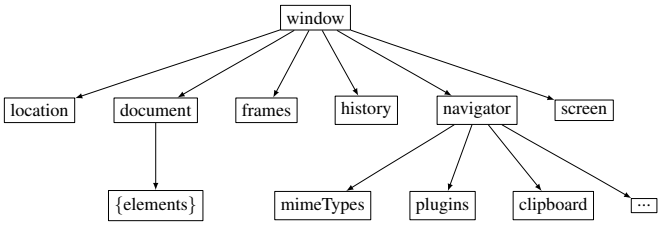

Fig. 2: In the JavaScript object hierarchy, every object is derived from `Object`. The `window` object is the root of all accessible objects and thus, for JavaScript Template Attacks.

JavaScript, the secret is the environment of the website, *i.e.*, the browser, operating system, and underlying hardware. A template is a matrix of properties (rows) for various environments (columns). All properties, e.g., browser properties, are retrieved through JavaScript.

Finding leakage is equivalent to detecting differences in these collected properties of the templates. The advantage of template attacks is that it is not necessary to understand the cause of the information leak. Hence, the template attack works fully automated. If the template contains different properties for different environments, our attack can deduce information about the (inaccessible) environment. This information can then be used by an attacker to mount a targeted exploit.

Our attack works in two phases which are outlined in Figure 1. The first phase is the *profiling phase*, which creates several profiles by collecting a set of properties, which are accessible via JavaScript, in different environments. These profiles are then combined to a template. In the *analysis phase*, we compare the properties of templates to automatically find differences caused by the environment. These discovered differences leak information about the environment which can be used on any webpage to mount a targeted attack.

A. Profiling Phase

The first phase is the profiling phase which builds the templates consisting of multiple profiles. The profiling phase runs entirely inside the browser and is implemented in JavaScript.

As a first step, the profiling code creates a list of properties which are accessible from JavaScript. In JavaScript, the accessible properties are either functions, numbers, strings, booleans, arrays, or objects. We refer to numbers, strings, and booleans as *primitive types*, as they have a single value which can directly be accessed and read. Objects and arrays (which are only a special type of object) are *complex types*, as they do not have a single generically comparable value. Instead, they are comprised of multiple primitive types and possibly further complex types.

Functions are more complex and require at least a certain understanding of the semantics to invoke them. This is an orthogonal problem [34], and thus, the properties that are returned by function calls are subject to future work. Solving this problem also allows applying JavaScript Template Attacks trivially to properties returned by functions. Even though we do not evaluate functions, we can still leverage functions for the templates. First, functions itself have a set of properties, e.g., `name` or `length`. Second, with *artificial properties*, we describe a way to add custom properties to the profiling phase. This allows us to convert simple functions, e.g., the `toString` function, into properties.

We distinguish between *native properties*, which are defined by the language or the browser, and *artificial properties* which can be added manually or automatically before the profiling phase.

- **Native Properties.** Native properties are primitive or complex types which are defined either by the language, *i.e.*, in the ECMAScript standard, or by the browser. Examples include the `length` property of almost every object or the `document` property of the `window` object. Moreover, browsers often introduce own properties to support features which are not yet standardized, or which aid developers in the debugging process of web applications. Examples include the `window.chrome` property in Google Chrome or the `window.sidebar` property in Mozilla Firefox.
- **Artificial Properties.** We introduce the term *artificial properties* for properties which are typically not available in JavaScript. As JavaScript allows adding properties dynamically to any object, additional properties can be added to the profiled objects. These additional properties can, for example, be results of preceding function calls.
  Moreover, accessor properties can be added to the profiled objects. These properties are actually functions, as they do not return a static value but the result of a function. In contrast to functions, these properties do not support arguments. Thus, functions without arguments (e.g., `toString`) can be converted to artificial properties, allowing them to be used in the profiling phase.

1) Exploration Step: The first step of the profiling phase is to explore the list of all accessible properties. We leverage both reflections and the JavaScript functionality of iterating through properties of an object. Listing 1 shows our method to collect all properties from a given object. The properties include both inherited properties, which are not defined directly in the object but in the prototype chain, and non-inherited properties.

The goal is to identify as many properties as possible. There is no list of all available objects which can be used in the exploration step. However, in JavaScript, objects are linked with each other in so-called prototype chains. This is similar to class inheritance in other languages such as C++. Thus, from an arbitrary object, we can traverse all child elements and all parent elements. The root object of every object is `Object`.

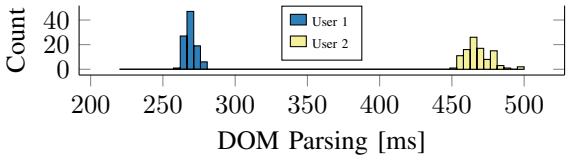

Fig. 3: The histogram of non-static properties (e.g., the DOM parsing time) can be used to, e.g., create fingerprints.

Furthermore, JavaScript has an object hierarchy as illustrated in Figure 2. Accessible objects (e.g., global objects, functions, HTML DOM) are referenced in the `window` object (representing the browser window), or in one of its child elements. Hence, by starting the property exploration step at the `window` object, we reach all accessible properties. The result of the exploration step is a list of accessible properties.

The exploration step has to run only once per environment, as the set of properties is static and does not change.

2) Collection Step: During the collection step, the JavaScript code creates a profile consisting of the properties identified in the exploration step and their values. The collection step runs again inside the browser in JavaScript.

Our property collection algorithm takes a list of properties which were identified in the exploration step. For every property, the collection step acquires the actual value of the property. As we only considered properties which have a concrete value (e.g., no property which first requires a function to be called), we can directly read the value of every such property. Note that this step is not limited to properties with concrete values, as adding properties resulting from function calls works the same if there is a way to call functions in an automated way. We refer to the set of collected properties as a profile. Combining profiles by running the collection step in different environments results in a *template*.

The template still contains properties which are not useful in the further analysis (cf. Section III-B), as they are not static. Examples include the page load time or the render time. These values change every time the page is reloaded. Exploiting such properties requires an understanding of the semantics of the values which is an orthogonal problem. Although semantics could theoretically be inferred using machine learning, our manual investigations already showed that these non-static properties did not contain any information we deemed usable for deducing environment information. Thus, we focussed on the more interesting static properties. For fingerprinting, non-static properties might still be useful and can be exploited by collecting histograms of the values which can then be matched to single users (cf. Figure 3).

To later on detect which properties are not static (cf. Section III-B1), *i.e.*, which properties do not have the same value on every read, the collection step needs to run multiple times. Every run collects the same properties and the hash of the corresponding object. Thus, after multiple runs (typically 3 to 4), there is a list of values for every property from the exploration step, composing the profile.

The profile is finally transmitted to the back-end server (e.g., using AJAX) for incorporation into the template used for further analysis.

B. Analysis Phase

The analysis phase is an offline phase which finds the properties leaking information about the environment. In contrast to the profiling phase, this second phase of the templating process does not run inside the browser.

The input to the analysis is the template generated in the profiling phase. Depending on the profiles contained in the template, the analysis phase can detect properties leaking different aspects of the environment. For example, if all profiles are recorded with the same browser on different operating systems, the analysis phase detects properties leaking the operating system.

The analysis phase is also split into two steps, the cleanup step and the property extraction step.

1) Cleanup Step: In the first step, the template has to be cleaned. Profiles collected in the profiling phase often contain duplicate properties. There are multiple reasons for this.

First, JavaScript objects are often heavily linked to other objects. This creates entries in the profile which appear to have a different name but are the same properties. For example, `window.frames.window.name` is the same property as `window.name`. These properties are detected if the objects have the same hash (which was stored in the collection phase), and are then unified.

Second, due to our method of collecting all properties (cf. Listing 1), the same property for one object might be collected multiple times. As we iterate through the entire prototype chain, we might get properties which are already overwritten by the child object. For example, the `name` property is collected for every object in the prototype chain. However, we can only access the `name` property of the last child, as it overwrites this property for all other objects in the prototype chain. These properties are trivial to remove as they have exactly the same name.

After the pruning of duplicates, the cleanup step has to identify properties which are not static, *i.e.*, properties which have changing values on different reads. For the collected values of every property, we test whether all the values are identical. If at least one of the values is different, we do not consider this property further. For example, the timestamp when the page was fully loaded (`window.performance.timing.responseEnd`) differs between multiple runs of the collection step. Although this property contains information about the environment, we cannot use it in an automated manner, as our automated method does not understand the semantics of properties (*i.e.*, that this is a timestamp).

In all observed cases, it was sufficient to run the collection step 3 to 4 times to filter out non-static properties in the cleanup step.

2) Property Extraction Step: Using the cleaned template, the property extraction step identifies properties which leak information. In this step, we first create the union of all

Browser	Profiling (once)	Profiling (twice)	Analysis	Total
Firefox	0.8 s	3.4 s	<0.1 s	3.5 s
Chrome	1.8 s	5.6 s	<0.1 s	5.7 s
Tor browser	0.7 s	3.2 s	<0.1 s	3.3 s
FPRandom	0.7 s	3.2 s	<0.1 s	3.3 s

TABLE I: The time it takes to run a JavaScript Template Attack for various browsers. As the analysis phase does not run inside the browser, the time difference is due to the number of collected properties. For all browsers, the total time is well below 10 s.

properties from all profiles of the template. This is necessary, as in many cases not all properties are present in all profiles.

For every property in the unified property list, the collected values in the different profiles are compared. If a property has the same value in all profiles, it can be ignored as it does not contain any information. This is the case for the majority of the properties, as properties are in the most cases not influenced by the environment, but only the current page.

However, if the value of a property varies between different profiles in the template, this property contains information that can be used to distinguish the environments. The same holds true if a property cannot be found in a template at all. The absence of a property is treated as a value of `undefined` for this property. In Section V, we show that the absence of properties can, for example, be used to detect whether a browser is used in private-browsing mode.

The final output of the analysis phase is a matrix of properties (rows) and their corresponding values for a set of different environments (columns). For all properties of the template matrix (*i.e.*, for each row), the value differs for at least one environment column. The more templates contain a different value for the property, the higher the entropy of the property, and thus the more it is able to deliver information about the environment. Section V shows the results of the JavaScript Template Attack on various browsers, including the properties which leak information.

C. Performance

In contrast to other template attacks [32], [68], [69], [79], JavaScript Template Attacks are extremely fast. Table I shows the runtime of the profiling and analysis phase for several different browsers. For all browsers, the runtime is well below 10 s, and could still be optimized.

The performance of the profiling phase depends on the performance of the JavaScript engine in the browser, and also on the number of properties provided by the browser. The higher the number of properties collected during the profiling phase, the longer this phase takes. If only native properties, *i.e.*, properties which are provided by the browser, are collected, the time of the profiling phase is below 2 s for all tested browsers. The artificial properties increase the runtime measurably.

The collection step of the profiling phase has to be run at least twice to remove properties which are not static, thus, the real time of the profiling phase increases by the number of runs. However, in all tests, the maximum number of required runs was 4. Moreover, to filter out properties which only change every second (e.g., a timestamp), we wait for 2 s between each run of the collection phase. Still, the profiling phase for most browsers is below 5 s.

As the analysis phase is offline, *i.e.*, it does not run in the browser, and thus, there are only negligible performance differences for different browsers, due to the number of properties and environment provided. The resulting runtime in all our tests was less than 0.1 s.

The total runtime of a JavaScript Template Attack is the sum of the profiling phase(s) and the analysis phase. This time slightly depends on the browser, but for most our tests it is below 5 s.

IV. Low-Level Properties

In this section, we show how the JavaScript Template Attack (cf. Section III) can be augmented with properties reflecting low-level properties of the environment. For this, we add artificial properties (cf. Section III-A) to the browser before running the profiling phase. The artificial properties are not properties per se but the result of functions deriving information about the underlying architecture or even microarchitecture.

Neither architectural nor microarchitectural properties are directly accessible in JavaScript. JavaScript code is platform independent. Thus, environmental properties have to be abstracted by the JavaScript engine. Moreover, for security reasons, JavaScript code runs in a sandbox and has no direct access to the underlying environment.

Still, recent research showed that such low-level properties can be obtained via side channels in JavaScript [54], [29], [64], [41], [26]. In this section, we present 2 new side channels to obtain architectural properties.

A. Instruction-Set Architecture

JavaScript is an interpreted language executed in a sandbox. Thus, the language itself is independent of the instruction-set architecture (ISA) of the machine it runs on. However, for performance reasons, JavaScript functions which are frequently executed are compiled to machine code using a just-in-time (JIT) compiler [73], [15].

Although JavaScript is oblivious to the ISA, the JIT compiler is limited by the ISA of the current platform. Thus, the JIT compiler behaves differently on CPUs with different ISAs. We can exploit this to distinguish one ISA from another ISA in JavaScript.

We craft a code snippet for which the JIT compiler can generate efficient code for one ISA and cannot generate equally efficient code for a different ISA. Then, we compare the runtime of this code snippet to a very similar code snippet for which the JIT compiler can generate efficient code on both ISAs. Using the runtime differences between the two code snippets, we can infer the underlying ISA.

Listing 2 contains two functions which are very similar. Both functions have data-dependent calculations with floating point numbers. However, the first function has one operation less. On x86, the JIT compiler uses the SSE XMM registers for

```
var a = 0.9, b = c = d = e = f = g = 0;
for(var i = 0; i < 10000000; i++) {
  b = 1.0 / a;
  c = 2.2 / b;
  d = 3.4 / c;
  e = 4.1 / d;
  f = 5.8 / e;
  g = 6.6 / f;
  // no operation
  a = a + b + c + d + e + f + g + g;
}
```

```
var a = 0.9, b = c = d = e = f = g = h = 0;
for(var i = 0; i < 10000000; i++) {
  b = 1.0 / a;
  c = 2.2 / b;
  d = 3.4 / c;
  e = 4.1 / d;
  f = 5.8 / e;
  g = 6.6 / f;
  h = 7.1 / g;
  a = a + b + c + d + e + f + g + h;
}
```

Listing 2: Two nearly identical code snippets to detect whether the code runs in a 32-bit or 64-bit environment. In 64-bit environments, both functions have basically the same execution time, whereas in 32-bit environments, the Firefox/Tor browser just-in-time compiler generates slower code for the right function as fewer registers are available to store intermediate results.

```
vaddss %xmm0,%xmm1,%xmm1
vdivsd %xmm7,%xmm6,%xmm6
vmovsd %xmm7,0x8(%esp)
vxorpd %xmm2,%xmm2,%xmm2
vxorpd %xmm7,%xmm7,%xmm7
```

```
vaddsd %xmm0,%xmm1,%xmm0
vdivsd %xmm2,%xmm11,%xmm3
vaddsd %xmm2,%xmm0,%xmm0
vdivsd %xmm3,%xmm10,%xmm4
```

Listing 3: The 32-bit x86 JIT compiler (left) cannot use as many registers as the 64-bit JIT compiler (right) and has to reuse registers and also save them onto the stack.

floating point operations. There are 8 XMM registers available on x86-32 but 16 XMM registers on x86-64.

Thus, on x86-64, all intermediate values can be kept in the registers for both functions. However, on x86-32, all intermediate values can be kept in the registers for the first function but not for the second function. This increases the runtime of the 32-bit code significantly, as registers have to be reused and thus temporarily saved on the stack (cf. Listing 3). As the function is executed multiple thousand times, the runtime difference is accumulated and can easily be measured.

The same approach can also be used to distinguish 32-bit ARM vs. 64-bit ARM environments. There, the number of floating-point registers is the same, however, the number of general registers differ. On 32-bit ARM, only 10 general registers (r0-r9) are used by the JIT compiler, whereas on 64-bit ARM, 32 general registers (r0-r31) are used by the JIT compiler.

We performed the measurement $10\,000$ times each on multiple 32-bit and 64-bit environments. In our tests, 32-bit environments can always be detected, 64-bit environments are in some cases classified as 32-bit due to scheduling or other noise which results in a slower execution of the fast function. However, we can still detect whether it is x86-32 or x86-64 with a probability of $>98\,\%$ for all tested environments.

In fact, the measurement does not even require a high-precision timer. Noise does not play a role, as it can be averaged out by repeating the measurements, and the timer resolution does not matter, as the number of loop iterations (cf. Listing 2) can be increased until it is distinguishable. The `performance.now` function with a resolution of $100\,\mathrm{ms}$ in Tor is already sufficient to measure the difference if combined with edge thresholding [64], [26].

B. Memory Allocator

Many browser exploits rely on the underlying memory allocator [4], [28]. Buffer overflows as well as use-after-free vulnerabilities often require knowledge of the memory layout to craft reliable exploits. As browsers use different memory allocators, reliable exploits require information about the allocation strategy.

Memory allocators differ between browsers, e.g., PartitionAlloc in Chrome [14] and jemalloc in Firefox [4]. Due to platform-specific virtual memory APIs, the memory allocator behavior in one browser can even differ between operating systems [13]. However, all memory allocators have in common that they allocate memory in blocks. The size of such a block is usually a power of 2.

Thus, there are two scenarios if a resizable data structure in JavaScript has to grow. Either, there is still sufficient space in the allocated memory block, and the data structure just uses this space. Or, the memory has to be resized, which can lead to a reallocation of the memory and thus also the data structure. In the latter case, we can measure a timing difference, as this operation requires large amounts of memory to be copied which is a slow process.

Moreover, memory allocators distinguish between small and large allocations. While small allocations are handled directly by the memory allocator, large allocations are delegated to the operating system. The operating system can then directly map the required memory segments, e.g., with `mmap` on Unix or `VirtualAlloc` on Windows. Attacks which require knowledge of physical addresses [30], [64] exploited the fact that memory mapped by the operating system is not initialized. When iterating over the memory, the operating system has to handle a page fault for every page that is accessed for the first time, which takes significantly longer than an access to an already mapped page. Thus, an attacker learns where a new

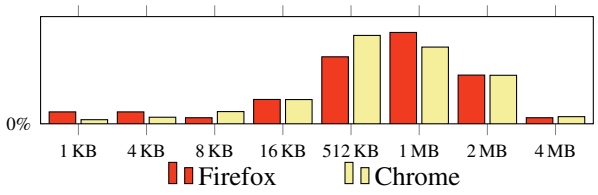

Fig. 4: Iterating over a large array shows timing spikes at different array indices. The distances are caused by the internal memory allocator which has to allocate new memory blocks. The timings which are the easiest to detect (and thus have the highest frequency in the histogram) are slow timings caused by the allocator requesting more memory from the operating system.

page starts, and thus the least significant bits of the physical address.

We only focus on the timing differences from the allocator itself, not on timing differences caused by the operating system. Note that page faults can of course also be used to learn information about the environment. However, as most systems use pages with a size of 4 KB, there is not much information to gain from exploiting this side channel.

To infer information about the memory allocator of the browser, we first allocate a small array of several kilobytes. We then choose a step size of 512 B and continuously resize the array by this step size. For every resize, we measure the time it takes using `performance.now()` in combination with edge thresholding [64], [26]. This results in a sufficiently high timer resolution to see the activity of the memory allocator. The activity manifests itself in slightly higher timings compared to accesses without memory allocator activity.

By comparing the distances between the high timings, we can infer the allocated size of the memory region. Figure 4 shows a histogram of the timing differences for Firefox and Chrome, grouped into typical sizes used by memory allocators. The default allocation size is detected correctly for both Chrome (512 KB) and Firefox (1 MB).

Measurement noise due to the coarse-grained `performance.now` timing function and interrupts leads to spurious high timings and missed high timings. The smaller buckets in the histogram are due to some smaller buckets used by the memory allocators, as well as spurious high timings. If the activity of a memory allocator (*i.e.*, a high timing) is missed, the bucket size is incorrectly identified as too large. However, as we see in the histogram, in the majority of the cases (*i.e.*, the highest peak in the histogram) the allocation size is determined correctly.

C. Graphics

WebGL allows the browser to access low-level properties and functions of the graphics card. The amount of information which can be gathered from the graphics card has already been used as a source for browser fingerprinting [39], [9]. Especially as WebGL does not require any browser permissions, it is an easy-to-use source for properties. In this section, we show that JavaScript Template Attacks can be trivially extended to also detect leaking properties in the WebGL extension.

The WebGL extension is not a static object which is always available through the object hierarchy (cf. Figure 2). Thus, on a blank site, there is no reference to a WebGL object or any of the WebGL extensions. However, by simply creating a WebGL element and attaching it to the `window` object, we can use a JavaScript Template Attack on the WebGL element as well.

```
<canvas id="glCanvas" width="640" height="480"/>
<script type="text/javascript">
  // add artificial property "canvas"
  window.canvas =
    document.querySelector("#glCanvas");
  // add artificial property "gl" for WebGL
  window.gl = window.canvas.getContext("webgl");
</script>
```

Listing 4: Adding the canvas element as well as the WebGL object as an artificial property to the `window` object.

Listing 4 shows the corresponding code to add WebGL as an artificial property to the object hierarchy. WebGL requires an HTML `canvas` element to instantiate the WebGL extension. We also add the `canvas` element to the `window` object as an artificial property as it contains properties as well.

The WebGL object contains 435 properties. 296 out of the 435 properties are only constants which refer to specific WebGL parameters that can be actively queried from OpenGL. Thus, these properties itself do not contain any information. Hence, we have to automatically query the values of all parameters and again add them to `window` object as artificial properties. Querying the value of a parameter is as simple as `window.wgl[param] = gl.getParameter(gl[param])` for every property of the WebGL object.

Adding the base WebGL parameters as artificial properties adds already close to 300 properties accessible to a JavaScript Template Attack. Another large set of parameters corresponding to WebGL, and therefore the underlying hardware and environment, is not directly accessible through the WebGL object but through WebGL extensions. WebGL extensions provide additional functions and parameters of OpenGL to the browser. All specified and not-yet specified extensions are registered in the WebGL Extension Registry [27].

For every WebGL extension which is currently specified, `gl.getExtension(extensionName)` returns either an object of the extension if it is supported, or `null`. If the browser and environment support the extension, we can use it in the same way as the normal WebGL object. Again, every extension provides constant properties which can be used to query the parameter value from the extension. This is fully automated in the same manner as for the WebGL object.

Adding the parameters of all extensions adds around 100 additional properties to the `window` object. While the Tor browser does not provide any WebGL extension, there are 96 parameters from 23 extensions in Chrome and 115 parameters from 24 extensions in Firefox. In Section V, we show that the properties created from WebGL parameters can be used to infer information about the environment.

Device	ISA	Operating System	Browser
PC1	x86-64	Kubuntu 16.04.4 LTS	Chrome, Firefox, Tor
		Windows 10	Chrome, Firefox, Tor, Edge
PC2	x86-64	Kubuntu 18.04 LTS	Chrome, Firefox, Tor
		Windows 7	Chrome, Firefox, Tor, Edge
PC3	x86-64	Kubuntu 16.04.5 LTS	Chrome, Firefox, Tor
		Windows 10	Chrome, Firefox, Tor, Edge
VM1	x86-32	Windows XP	Chrome, Firefox, Tor
VM2	x86-64	Kubuntu 17.04	Chrome, Firefox, Tor
VM3	x86-64	Windows 10	Chrome, Firefox, Tor, Edge
Phone1	AArch64	Android 7.0	Chrome, Firefox, Tor
Phone2	ARMv7	Android 6.0.1	Chrome, Firefox, Tor
Phone3	AArch64	Ubuntu 16.04	Chrome, Firefox, Tor

TABLE II: List of environments used for the case studies.

D. Microarchitectural Elements

There is a variety of other low-level properties which have already been used in side-channel attacks from JavaScript [75], [54], [29], [44], [64], [26], [41], [77], [36]. All these properties can theoretically also be added as artificial properties. However, these attacks are already powerful attacks itself. Furthermore, these attacks are often quite fragile and require information about the system itself, without providing information about the environment, but only about specific secrets. Thus, an attacker would rather use such microarchitectural side-channel attacks to complement a JavaScript Template Attack.

Moreover, as a consequence to the Spectre attacks, which have not only been shown in native code but also in JavaScript, browser manufacturers limited the access to high-precision timers rigorously. This does not only include the provided `performance.now` function but also self-built timers using `SharedArrayBuffers` [26], [64]. As a result, many of the well-known microarchitectural attacks are currently prevented until a new timing source is found, or browser vendors re-enable `SharedArrayBuffers` and precise timers as, e.g., Google plans to do with Chrome [59].

V. Case Studies

In this section, we provide multiple case studies of our JavaScript Template Attack in various environments. We scan all native properties which are in the hierarchy starting at `window` (cf. Figure 2). Additionally, we add the artificial properties described in Section IV, which includes all WebGL properties and WebGL extension properties, the memory allocator and the ISA. As browsers, we used Google Chrome 67.0.3396.99, Mozilla Firefox 61.0.1, Tor 7.5.6, and– if available–Microsoft Edge 42.17134.1.0. Table II shows a table of all the environments we used for testing.

For all case studies, we used our open-source JavaScript Template Attack framework.[1] In the case studies, we tried to automatically infer as much information about the environment as possible.

The collected information can be used directly or indirectly to mount targeted exploits. Directly usable information includes, for example, the operating system and architecture, which is required knowledge for many exploits. Indirectly usable information includes, for example, the use of privacy extensions or private mode which can be used to imitate plausible looking system messages or dialogues, e.g., for phishing [11], [12].

In all use cases, we assume that we cannot simply read the correct information directly from the browser, e.g., from the user agent. The user agent string contains among others operating system, browser name and version. Even if we get this information directly, an attacker cannot rely on this information, as it can easily be modified using browser extensions. Moreover, some browsers such as Tor do not even provide any information about the environment in the user agent.

A. Browser Detection

The major browsers all have their own JavaScript and rendering engine. Thus, exploitable bugs are usually limited to one browser. Especially exploits which heavily rely on the internal functionality of the browser are limited to a specific browser.

The differences between the browsers do not only prevent one browser exploit to work in a different browser, but it also makes it easy to distinguish browsers. Every browser supports a distinct set of functions [18] and also provides browser-specific properties through so-called vendor prefixes [49]. Already the number of documented properties for the major browsers differs significantly, with 2698 for Chrome, 2247 for Firefox, and 1806 for Edge [47].

Moreover, as the JavaScript engine differs between browsers, the values of properties are also different. We added the `toString` representation of functions as simple artificial properties. As the representation is not strictly defined, it differs between browsers. This difference has also been exploited to detect the manipulation of the user-agent string [76].

However, not only the values of properties are different but also the available properties differ between browsers. We compared all accessible native and artificial properties of Firefox and Chrome running in exactly the same environment. Every property which was not implemented was assumed to have the value `undefined`, which is the case for every undefined variable.

In total, our JavaScript Template Attack discovered $14\,544$ properties which differed between Firefox and Chrome. With $60.1\,\%$, the majority of differing properties is the string representation of functions. Without these artificial properties, there are still 5796 properties which differ between the two browsers. Similarly, there are $15\,670$ different properties between Edge and Firefox, and 8913 between Edge and Chrome.

Even between Firefox and the Tor browser (which is based on Firefox) we found 3055 properties with different values. Again, the majority of differing properties ($63.6\,\%$) is the string representation of functions. However, as both browsers share the same code base, the difference is not in the format of the string representing the function. The differences are caused by functions which are only available in one of the two browsers. Without considering functions, there are still 1111 properties with different values between the two browsers.

Summarizing, even browsers which share a common code base can be easily distinguished using our JavaScript Template

[1] The source of the framework can be found in a GitHub repository at https://github.com/IAIK/jstemplate

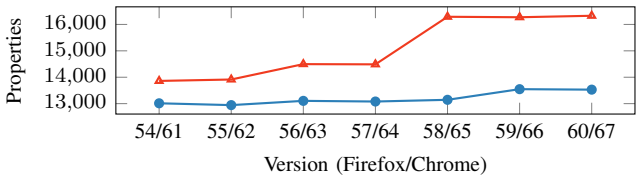

Fig. 5: The number of identified properties from Chrome 60 to 67 (●) and Firefox 53 to 60 (▲). The trend shows that the number of properties increases over time.

Attack. For all tested browsers, there are more than 1000 properties with different values which can be used to uniquely identify a specific browser. We were successfully able to distinguish all of the 40 tested setups (cf. Table II) without any false positives or false negatives. Even in the hypothetical case that native properties do not leak this information anymore, the artificial memory-allocator property (cf. Section IV-B) can be used to distinguish browsers.

1) Browser Version: For many exploits, it is not only necessary to know which browser the victim uses but also the exact browser version. As exploits are disclosed, they are usually fixed by the browser vendor in one of the next versions. Thus, to reliably run an exploit on a browser, knowing the browser version is important for selecting a working exploit.

Figure 5 shows the number of properties discovered using a JavaScript Template Attack for every Firefox and Chrome version since 53 and 60 respectively. For all versions of Firefox and Chrome, there are many unique properties. We further compared the number of properties between all versions of the browsers. There is always at least one property which has changed between any two versions. For all tested browsers in all setups, we were able to distinguish the versions of the browsers. We can see a clear trend to an increasing number of properties, although in some versions properties are removed due to changes in the standards or deprecation of functions.

Summarizing, for all major browsers, it is easy to detect the actual browser version by counting the number of implemented properties, even without inspecting the values of the properties. As the trend is to continuously add more features instead of removing features, we expect the browser version detection to work on newer versions of the browsers as well.

B. Privacy-Extension Detection

There are several privacy-enhancing extensions for browsers, e.g., ad blocker or anti-tracking extensions. Some of them modify the information sent to servers (e.g., FP-Block [72]) or overwrite JavaScript functionality (e.g., Chrome Zero [63]). Often, such plugins change properties which are accessible from JavaScript. Thus, a JavaScript Template Attack can detect the presence of such plugins.

Note that the detection of such plugins can have various uses. First, it allows an attacker to create dialogues which look as if they are coming from such a browser extension, tricking the user into interacting with them. For example, a user might be tricked into clicking on a fake update dialogue from an extension, which actually triggers, e.g., a switch to fullscreen mode or a file download. Second, exploits can be automatically adapted to avoid functions which are modified by a browser extension such as Chrome Zero [63]. Finally, as already described by Mowery et al. [46], Eckersley [19], Acar et al. [1], or Nikiforakis et al. [51], such plugins are a source for fingerprinting, as they lead to inconsistencies.

We evaluated Chrome Zero [63], Chameleon [45], Canvas Defender[2], CyDec Platform AntiFingerprint[3], Ghostery[4], and WebAPI Manager [66]. For Chrome Zero, we are not only able to detect that it is active but also the current protection level (cf. Table III).

Mounting a JavaScript Template Attack with the WebAPI Manager extension [66] active leads to similar results. Again, we can detect that the extension is active as it modifies properties. Similar to the Chrome Zero extension, we can also detect which protection level is used (lite, conservative, aggressive) as shown in Table IV. As with Chrome Zero, it is not possible to access the references to the original functions.

For Canvas Defender, we cannot only determine that it is used (105 distinguishing properties) but also semi-automatically circumvent it. Canvas Defender replaces functions which are used or can potentially be used for fingerprinting with its own functions. However, as it requires the original functionality as well, it stores references to the original functions as properties of the `window` object. Thus, a JavaScript Template Attack does not only discover the use of the extension, but it also reveals the original functions. From an attacker's perspective, the function references are conveniently named the same as the original functions and just prefixed with a random string. Thus, JavaScript Template Attacks cannot only detect the tested extension. It can even be used to circumvent it, leaving more than 30 000 users who have this extension installed with a false sense of security.

Mounting a JavaScript Template Attack with the WebAPI Manager extension [66] activated leads to similar results. Again, we can detect that the extension is active as it modifies between 1472 and 2307 properties, depending on the protection level. We can also easily detect whether Chameleon or CyDec are active. Our JavaScript Template Attack identified 13 properties which are modified by Chameleon and 2365 properties which are modified or added by CyDec. Each of these properties can be used to detect that the user has Chameleon installed and activated. Interestingly, Ghostery is only detectable when installed in Firefox. Ghostery adds Ghostery-specific elements to every page in Firefox, revealing the usage of this extension. In Chrome, there are no differences, making Ghostery in Chrome not detectable with our automated method.

We can conclude that JavaScript Template Attacks are a valuable method for developers of privacy-enhancing extensions to test their extension. If extensions try to hide references instead of making them inaccessible, they can be easily revealed again, allowing an attacker to easily circumvent

[2]https://multiloginapp.com/canvasdefender-browser-extension/
[3]https://addons.mozilla.org/en-US/firefox/addon/cydec-platform-antifingerprint/
[4]https://www.ghostery.com/

vs.	Medium	High	Tin Foil Hat	Sample Expression
Low	27	29	27	`!!((Worker&&Worker.toString().indexOf(''postMessage'')==-1)|0)`
Medium	-	28	28	`!!((addEventListener&&addEventListener.toString().indexOf(''block'')!=-1)|0)`
High	-	-	28	`!!((performance.now&&performance.now.toString().indexOf(''fuzz'')!=-1)|0)`
Tin Foil Hat	-	-	-	`!!((Array&&Array.toString().indexOf(''Proxy'')!=-1)|0)`

TABLE III: Every row of the table represents a protection level of Chrome Zero [63]. On the left side of the table is the number of properties which have a different value compared to the protection level in the corresponding column. The right side of the table shows one sample expression which is only true if the corresponding protection level is active.

vs.	Lite	Conservative	Aggressive
None	1492	1539	2381
Lite	-	67	894
Conservative	-	-	843

TABLE IV: Every row of the table represents a protection level of the Web API Manager [66]. The table contains the number of properties with a different value compared to the protection level in the corresponding column.

the extension. JavaScript Template Attacks can easily uncover such leaked references during development.

C. Private Mode Detection

Similarly to privacy-enhancing extensions, Firefox, Chrome, and Edge provide a built-in private-browsing mode. In this mode, the browser does not keep any tracks of visited websites, such as cookies or history. Furthermore, private browsing also includes some tracking protection [48].

We mounted a JavaScript Template Attack to detect whether there are any differences between normal mode and private-browsing mode. In Chrome, there are no detectable differences when using the browser in private-browsing mode. Similarly, we cannot detect differences between normal mode and guest mode, a feature similar to private-browsing mode.

For Firefox, however, there are properties revealing whether the current window is a private-browsing window or a normal window. For example, service workers are not available in private-browsing mode. Thus, all 73 properties corresponding to service workers are only detected in normal mode and not available in private-browsing mode.

An additional hint that a Firefox window is in private-browsing mode is the value of the `doNotTrack` property. Per default, this flag is set to "unspecified" and only gets an actual value if the user specifies one in the browser settings. In private-browsing mode, however, this flag is always set to "1" if not configured differently by the user. Thus, if this value is not "1", the window is probably not in private-browsing mode.

For Edge, we can also detect whether the window is in private-browsing mode or normal mode. We detected 72 properties corresponding to local databases and Microsoft-specific properties, such as `MSCredentials`. These features are only available in normal mode. Moreover, Edge handles the `doNotTrack` property in the same way as Firefox, providing another hint about the current mode.

D. Operating System Detection

If exploits interact with the environment, e.g., access operating-system specific resources, an attacker has to know which operating system is used. The same is true if an attacker tries to create fake system messages [11], [12]. Most browsers are available for all major platforms and provide the same functionality on all platforms. Thus, for a legitimate website, there is usually no reason to detect the operating system for any functionality except for statistics.

We mounted a JavaScript Template Attack to detect whether any property would reveal the underlying operating system. For Microsoft Edge, this would be trivial, as it only runs on Microsoft Windows. Thus, we did not include this browser in our tests. Furthermore, to eliminate influences which are not from the operating system, wherever possible, we mounted the attack on the same hardware for the different operating systems.

The Tor browser actively tries to eliminate all differences among operating systems. Still, some properties differ between operating systems. An interesting difference in properties we detected is the `window.innerWidth`/`window.innerHeight` pair. Although the Tor browser warns the user not to resize the window to prevent fingerprinting using these properties, they are not always the same. For example, `window.innerWidth` is 1000 on Linux (Kubuntu 16.04.4) but 1001 on Windows 10. The reason for this is that Windows 10 has native support for high-density displays and automatically scales application such that they have a usable size. For the browser, the screen appears to be smaller than the actual screen resolution. However, this scaling seems to introduce rounding errors, which results in this difference in the `window.innerWidth` property. On Android (with Orfox), this property is also different with a value of 980.

The font rendering causes another difference between operating systems. The list of installed fonts is already known to provide reliable fingerprints [6]. Due to different available fonts as well as differences in the font rendering code, the same text has different dimensions on different operating systems [51]. For example, in Tor, a default heading on Windows 10 is 1 pixel higher than on Linux. Such differences do not only exist for the Tor browser but also for Chrome.

For Firefox, we detected additional properties which give an even better indication about the underlying operating system. Firefox has experimental support for virtual-reality displays (e.g., `window.navigator.activeVRDisplays`). However, in the current version (61.0.1), only Windows is fully supported. Linux is not supported, and macOS is only partially

supported. Thus, by detecting which functions are available for virtual-reality displays, the operating system can be detected.

Moreover, we detected differences in WebGL properties which allow distinguishing the operating system for both Firefox and Chrome. One property which reveals whether the underlying operating system is Windows is the `UNMASKED_RENDERER_WEBGL` property of the `WEBGL_debug_renderer_info` extension. This property contains the OpenGL renderer used for WebGL. On Windows, this string always contains `ANGLE`, which stands for *Almost Native Graphics Layer Engine*, the OpenGL compatibility layer on Windows [76]. The string `Iris` refers to Intel Iris Graphics, a GPU which is mostly found in MacBook Pros and iMacs, thus indicating that the browser is running on macOS.

The Android operating system can also be distinguished from other operating systems mostly by the lack of functions (and thus properties). For example, Firefox on Android does not support speech synthesis (e.g., `window.SpeechSynthesis`). Chrome on Android, for example, does not support support inline installation of extensions (e.g., `chrome.webstore.install`). Both browsers do not support shared workers (e.g., `window.SharedWorker`) and plugins on Android.

However, we detected one feature which is only available on Chrome for Android. The `window.MediaSession` allows a mobile website to show information about the currently played multimedia content in the notification bar. If this property is available, the underlying operating system is Android.

For some of the properties, the operating system can be directly inferred, and by combining the detected properties, we can reliably detect any of the major operating systems.

E. Architecture Detection

For exploits running binary code, it is vital to know the current ISA. Assuming a wrong ISA (e.g., x86 instead of ARM) results in an unsuccessful exploit. In both cases, the exploit attempt does not only fail, but it might also be detected.

As with all other properties, the Tor browser tries to provide the same functionality and properties on all architectures. On all desktop operating systems, the Tor browser reports the platform as `Win32`, independent of the actual operating system or ISA. However, we detected a difference when running a JavaScript Template Attack on Orfox, the official Android version of the Tor browser. There, the platform is not reported as `Win32` but the actual platform is reported (`armv8l` on an ARMv8 phone and `armv7l` on an ARMv7 phone). We also disclosed this issue to the developers, and it will be fixed in one of the future versions.

Another property which indicates the underlying ISA is again the renderer information as well as the vendor information from WebGL. `Adreno`, `Mali`, and `Tegra` renderer are only available for ARM. Thus, if this string is contained in the renderer information, the underlying ISA is ARM. Similarly, on Linux, the renderer information can even contain the specific microarchitecture. For example, on a Lenovo T460s with an Intel Skylake CPU, the vendor string contains `Intel` and the renderer property value is `Mesa DRI Intel(R) HD Graphics 520 (Skylake GT2)`.

Browser	MDN	JavaScript Template
Firefox	2247	15 709
Chrome	2698	13 570
Edge	1806	9666
Firefox Android	2104	15 612
Chrome Android	2676	13 119
Tor browser	2247†	15 639

† As the MDN does not distinguish between the Tor browser and Firefox, we used the Firefox numbers, as the Tor browser is based on Firefox.

TABLE V: The number of properties documented in the MDN Web Docs compared to the number of properties found using a JavaScript Template Attack.

Finally, the artificial property presented in Section IV-A can be used to distinguish 32-bit and 64-bit x86. We achieve a classification rate which is close to 100 %. Moreover, it has the huge advantage that it cannot easily be hidden from an attacker, whereas the values of properties can be anonymized by the browser vendors.

F. Virtual Machine Detection

Although virtual machines should not be distinguishable from native machines, we still detected one property which has a distinct value inside a virtual machine. In Firefox, the WebGL extension can reveal that Firefox is running inside a virtual machine. The `UNMASKED_VENDOR_WEBGL` property of the `WEBGL_debug_renderer_info` extension is set to `VMWare, Inc.` when running inside VirtualBox or VMWare. For the Tor browser and Chrome, we could not detect any property which immediately reveals that the environment is a virtual machine.

However, there are two properties which can give a hint that the underlying environment is a virtual machine. First is the reported screen resolution (`window.screen.availWidth` / `window.screen.availHeight`). If the value is an odd value, *i.e.*, not one of the usually used resolutions of screens, it is a strong indicator that the browser is running in a virtual machine. For example, on our test machine, the screen resolution is 1920x1080, and the reported resolution inside the VM is 1920x944. Second, the number of reported CPUs can be easily queried using `navigator.hardwareConcurrency`. For a native environment, this value is usually a power of two on consumer hardware. A small number which is not a power of two (e.g., 3) is also an indicator that the browser is running inside a virtual machine.

VI. Coverage Analysis

In this section, we analyze the coverage of JavaScript Template Attacks. As a baseline, we parsed the MDN Web Docs [47]. We then compared all our detected properties to the properties extracted from the MDN Web Docs.

Table V shows the number of properties we parsed from the MDN Web Docs as well as the number of properties detected with a JavaScript Template Attack for Firefox, Chrome (both on Linux and Android), Edge, and the Tor browser (Linux only). Interestingly, the number of detected properties for every browser is much higher than the number of properties officially documented. One reason for this is that the documentation is apparently not complete. Moreover, we access several internal,

Browser	Exploration	Without duplicates	Usable
Firefox	18 443	16 450	15 709
Chrome	15 585	13 604	13 570
Edge	13 752	11 850	9666
Firefox Android	18 214	16 296	15 612
Chrome Android	15 556	13 608	13 119
Tor browser	17 217	15 645	15 639

TABLE VI: The number of properties found using a JavaScript Template Attack and the number of properties which were left after the cleanup step of the analysis phase (cf. Section III-B1).

undocumented properties. This is an interesting aspect, as our JavaScript Template Attack also allows to find completely new properties which might not have been considered for fingerprinting before as they are not documented. Another reason is that we access the same property for multiple objects, e.g., the `length` property. Properties from the prototype chain are not documented if they are already documented for the parent object. Thus, this property is counted twice although it is in principle the same property.

Still, we do not achieve a $100\,\%$ coverage for multiple reasons. The majority of the documented properties does not belong to static objects, *i.e.*, objects which always exists in the browser. Many objects have to be dynamically created, e.g., exceptions, or instances of elements. Thus, we cannot automatically explore the properties of these objects. It is, however, possible to create such objects and add them to the hierarchy manually. We showed this for WebGL (cf. Section IV-C) and the `toString` function (cf. Section V-A). Future work has to research whether this step can be automated to achieve an even higher coverage. Nonetheless, as shown in Section V, the coverage is already sufficient to find many properties which reveal information about the environment.

Another reason for missing properties is that some browser-specific properties are not referenced by the `window` root object and are thus not in the hierarchy illustrated in Figure 2.

Table VI shows that most of the detected properties were actually usable for the property extraction step (cf. Section III-B2). The cleanup step (cf. Section III-B1) removed only a small percentage ($<15\,\%$) of the properties as they were duplicates. From the remaining properties, only a few ($<9\,\%$) had to be discarded as they changed their value when read multiple times. These properties were mostly timestamps.

For all browsers, we found around $10\,000$ usable properties. This massive number of automatically detected, partly undocumented and usable properties stresses the need for automated leakage detection.

VII. Discussion

In this section, we discuss the differences between JavaScript Template Attacks and traditional fingerprinting, its limitations, and possible future improvements.

A. Difference to Fingerprinting

Although JavaScript Template Attacks look similar to fingerprinting, they have a different goal. In traditional fingerprinting, attackers try to identify properties or combinations of properties which are unique for a *user*. For JavaScript Template Attacks, we try to identify properties or combinations of properties which are unique for an *environment*. In contrast to fingerprinting, it is preferable that the identified properties do not change for different users, but only for environments.

The overlap between JavaScript Template Attacks and fingerprinting lies in the fact that many detected properties can be used for fingerprinting. This makes JavaScript Template Attacks also a powerful method to automatically search for new fingerprinting sources. It detects differences in properties within seconds, without requiring any manual analysis. Thus, this also reduces the time to search for new fingerprints. As shown in Section V, several of the properties we used to detect the environment are indeed useful for fingerprinting.

B. Limitations and Future Work

We currently focussed mainly on properties, and only added the `toString` function and the functions to query WebGL parameters. Thus, many properties which are hidden behind function calls are not identified. We expect that the results of function calls provide more information about the environment, similar to function calls in Android [69].

The most simple case are functions which do not take any argument. Still, adding these functions as artificial properties is not as straightforward as it seems at first glance. Several functions have to be blacklisted, as they would abort the script (e.g., `window.close()` or `document.location.reload()`) or pause the script until the user actively continues execution (e.g., `alert()`). Moreover, cycles have to be detected to not be stuck in endless loops (e.g., the result of `toString` is again a string which provides a `toString` function).

Future research has to investigate how this approach can be applied to functions with parameters. In contrast to Java [69], getting the number and types of arguments for a function in JavaScript is not straightforward. Moreover, choosing sane values is a hard problem. It would be interesting to combine techniques from fuzzing which select sane values with JavaScript Template Attacks to automatically test the return values of functions. However, fuzzing JavaScript APIs with a high coverage is still an open research problem [34].

An interesting direction would also be to target certain web standards, such as Web USB or Web NFC. To get useful results, a JavaScript Template Attack would require some manual initialization and possibly user interaction to grant the corresponding permission. Thus, this is not in the scope of this paper, as it requires more research into automatically understanding the semantics of functions and calling them.

C. Countermeasures

Most browsers do not have the goal to prevent identification of the environment. While some properties which leak information about the environment cannot easily be removed, others can be anonymized as it is, e.g., done in the Tor browser. From our experiments, we have seen that Tor's anti-fingerprinting design [57] also prevents that an attacker can leak a lot of information about the environment. Thus, anti-fingerprinting techniques–if implemented correctly–are a viable method to also prevent the detection of the environment.

As shown in Section V, JavaScript Template Attacks can detect leakage in privacy-enhancing browsers and extensions. Thus, the main use case of JavaScript Template Attacks is to provide an automated augmentation for the development process of defense mechanisms. If used in the development process of privacy-enhancing browsers and extensions, they can detect overlooked properties, as, e.g., in the case of the Orfox browser (cf. Section V-E). This also shows shortcomings in the implementation of extensions, e.g., the original function references are still accessible (cf. Section V-B).

VIII. Conclusion

In this paper, we presented JavaScript Template Attacks, a fully automated novel technique to detect subtle differences in browser engines caused by the environment. Furthermore, we showed two new side-channel attacks on browsers, allowing to detect the instruction-set architecture and the used memory allocator. Our techniques even work in the presence of anti-fingerprinting mechanisms in the browser. By leveraging the found differences in the browser engine, an attacker learns details about the environment and can get a clearer picture of a system for a targeted exploit. Moreover, our technique is applicable to identifying new fingerprints automatically.

We found environment-dependent properties in all major browsers, including Tor for Android, allowing us to reveal the underlying operating system, CPU architecture, used privacy-enhancing plugins, and the exact browser version. Furthermore, we showed that privacy-enhancing extensions can provide a false sense of security as they can be circumvented semi-automatically using our technique if not implemented correctly. Thus, we stress that our method should be used in the development process of browsers and privacy extensions to automatically find flaws in the implementation.

Acknowledgments

We would like to thank our anonymous reviewers for their feedback. This work 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] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, "Fpdetective: dusting the web for fingerprinters," in *CCS*, 2013.

[2] O. Acıiçmez, c. K. Koç, and J.-p. Seifert, "On the Power of Simple Branch Prediction Analysis," in *AsiaCCS*, 2007.

[3] O. Acıiçmez, J.-P. Seifert, and c. K. Koç, "Predicting secret keys via branch prediction," in *CT-RSA 2007*, 2007.

[4] P. Argyroudis and C. Karamitas, "Exploiting the jemalloc memory allocator: Owning firefox's heap," *Blackhat USA*, 2012.

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

[6] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre, "User tracking on the web via cross-browser fingerprinting," in *Nordic Conference on Secure IT Systems*, 2011.

[7] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2009.

[8] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," *arXiv:1811.05441*, 2018.

[9] Y. Cao, S. Li, and E. Wijmans, "Browser fingerprinting via os and hardware level features," in *NDSS*, 2017.

[10] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *CHES*, 2002.

[11] G. Chatzisofroniou, "Efficient wi-fi phishing attacks," 2016. [Online]. Available: https://census-labs.com/media/effective_wifi_phishing_33c3.pdf

[12] ——, "Extra phishing pages," 2018. [Online]. Available: https://github.com/wifiphisher/extra-phishing-pages

[13] Chromium, "Key concepts in chrome memory," 2018. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/lkgr/docs/memory/key_concepts.md

[14] ——, "Partitionalloc design," 2018. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/lkcr/base/allocator/partition_allocator/PartitionAlloc.md

[15] L. Clark, "A crash course in just-in-time (jit) compilers," 2017. [Online]. Available: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

[16] S. Cortes, "Legalizing domestic surveillance: The role of mutual legal assistance treaties in deanonymizing torbrowser technology," 2015.

[17] J. Dalins, C. Wilson, and M. Carman, "Criminal motivation on the dark web: A categorisation model for law enforcement," *Digital Investigation*, 2018.

[18] A. Deveria. (2018) Can i use... support tables for html5, css3, etc. [Online]. Available: http://caniuse.com/

[19] P. Eckersley, "How unique is your web browser?" in *PETS*, 2010.

[20] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *CCS*, 2016.

[21] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[22] A. Fobian and C.-B. Bender, "Firefox 0-day targeting tor-users," 2016.

[23] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *IEEE S&P*, 2018.

[24] B. Gellman, C. Timberg, and S. Rich, "Secret nsa documents show campaign against tor encrypted network," *The Washington Post*, p. 4, 2013.

[25] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *ACNS*, 2018.

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

[27] K. Group, "Webgl extension registry," 2018. [Online]. Available: https://www.khronos.org/registry/webgl/extensions/

[28] S. Groß. (2017) Exploiting a cross-mmap overflow in firefox. [Online]. Available: https://saelo.github.io/posts/firefox-script-loader-overflow.html

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

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

[31] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[32] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015.

[33] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *S&P*, 2011.

[34] R. Hodován and Á. Kiss, "Fuzzing javascript engine apis," in *International Conference on Integrated Formal Methods*, 2016.

[35] S. Kemp, "Digitnal in 2018: World's internet users pass the 4 billion mark," 2018. [Online]. Available: https://wearesocial.com/blog/2018/01/global-digital-report-2018

[36] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *S&P*, 2019.

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

[38] P. Laperdrix, B. Baudry, and V. Mishra, "Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques," in *ESSoS*, 2017.

[39] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *S&P*, 2016.

[40] M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing rowhammer faults through network requests," *arXiv:1711.08002*, 2017.

[41] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard, "Practical Keystroke Timing Attacks in Sandboxed JavaScript," in *ESORICS*, 2017.

[42] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security Symposium*, 2016.

[43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[44] 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.

[45] A. Miagkov, "Chameleon - browser fingerprinting protection for everybody," 2015. [Online]. Available: https://github.com/ghostwords/chameleon

[46] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in javascript implementations," in *W2SP*, 2011.

[47] Mozilla, "mdn-browser-compat-data," 2018. [Online]. Available: https://github.com/mdn/browser-compat-data

[48] ——, "Private browsing - use firefox without saving history," 2018. [Online]. Available: https://support.mozilla.org/en-US/kb/private-browsing-use-firefox-without-history

[49] ——, "Vendor prefix," 2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Vendor_Prefix

[50] N. Nikiforakis, W. Joosen, and B. Livshits, "Privaricator: Deceiving fingerprinters with little white lies," in *WWW*, 2015.

[51] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *Security and privacy (SP)*, 2013.

[52] 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/

[53] L. Olejnik, S. Englehardt, and A. Narayanan, "Battery status not included: Assessing privacy in web standards," in *Workshop on Privacy Engineering (IWPE)*, 2017.

[54] 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*, 2015.

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

[56] C. Percival, "Cache missing for fun and profit," in *BSDCan*, 2005.

[57] M. Perry, E. Clark, S. Murdoch, and G. Koppen. (2018, 05) The design and implementation of the tor browser. [Online]. Available: https://www.torproject.org/projects/torbrowser/design/

[58] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security Symposium*, 2016.

[59] C. Reis, "Mitigating spectre with site isolation in chrome," 2018. [Online]. Available: https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html

[60] D. S. Rudesill, J. Caverlee, and D. Sui, "The deep web and the darknet: A look inside the internet's massive black box," 2015.

[61] B. Schneier, "Attacking tor: how the nsa targets users' online anonymity," *The Guardian*, vol. 4, 2013.

[62] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks ," in *DIMVA*, 2017.

[63] M. Schwarz, M. Lipp, and D. Gruss, "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks," in *NDSS*, 2018.

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

[65] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read arbitrary memory over network," *arXiv:1807.10535*, 2018.

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

[67] 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.

[68] R. Spreitzer, F. Kirchengast, D. Gruss, and S. Mangard, "Procharvester: Fully automated analysis of procfs side-channel leaks on android," in *AsiaCCS*, 2018.

[69] R. Spreitzer, G. Palfinger, and S. Mangard, "Scandroid: Automated side-channel analysis of android apis," in *11th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2018.

[70] P. Stone, "Pixel Perfect Timing Attacks with HTML5," Jun. 2013. [Online]. Available: http://www.contextis.com/files/Browser_Timing_Attacks.pdf

[71] A. Tatar, R. Krishnan, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *USENIX ATC*, 2018.

[72] C. F. Torres, H. Jonker, and S. Mauw, "Fp-block: Usable web privacy by controlling browser fingerprinting," in *ESORICS*, 2015.

[73] V8 Team, "Launching ignition and turbofan," 2017. [Online]. Available: https://v8project.blogspot.com/2017/05/launching-ignition-and-turbofan.html

[74] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security Symposium*, 2018.

[75] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *CCS*, 2015.

[76] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "Fp-scanner: The privacy implications of browser fingerprint inconsistencies," in *USENIX Security Symposium*, 2018.

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

[78] B. Vitaris, "Firefox zero-day can be used to deanonymize tor users," 2016. [Online]. Available: https://www.deepdotweb.com/2016/12/11/firefox-zero-day-can-used-deanonymize-tor-users

[79] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "DATA - differential address trace analysis: Finding address-based side-channels in binaries," in *USENIX Security Symposium*, 2018.

[80] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution," *Technical report*, 2018.

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