Fast and Reliable Browser Identification with JavaScript Engine Fingerprinting

Martin Mulazzani*, Philipp Reschl†, Markus Huber*,
Manuel Leithner*, Sebastian Schrittwieser* and Edgar Weippl*

*SBA Research
Vienna, Austria
†FH Campus Wien
Vienna, Austria

***Abstract*—Web browsers are crucial software components in today's usage of the Internet, but the reliable detection of whether a client is using a specific browser can still be considered a nontrivial problem. Reliable browser identification is crucial for online security and privacy e.g., regarding drive-by downloads and user tracking, and can be used to enhance the user's security. So far the UserAgent string is often used to identify a given browser, but it is a self-reported string provided by the client and can be changed arbitrarily.**

In this paper we propose a new method for identifying web browsers based on the underlying Javascript engine, which can be executed on the client side within a fraction of a second. Our method is three orders of magnitude faster than previous work on Javascript engine fingerprinting, and can be implemented with well below a few hundred lines of code. We show the feasibility of our method with a survey and discuss the consequences for user privacy and browser security. Furthermore, we collected data for more than 150 browser and operating system combinations, and present algorithms to make browser identification as fast as possible. UserAgent string modifications become easily detectable with JavaScript engine fingerprinting, which is shown exemplarily on the Tor browser bundle as it uses a uniform UserAgent string across different browser versions. Finally, we propose to use our results for enhancing state-of-the-art session management (with or without SSL), as reliable browser identification can be used to increase the complexity of session hijacking attacks considerably.

***Keywords*-Browser Fingerprinting, Privacy, Security**

I. Introduction

With the rise of ever more sophisticated Web applications that nowadays even compete with native software, the web browser became the dominant interface connecting the user to a computing system. Platforms such as Gmail or Zoho.com were designed from the ground up to be primarily accessed via web browser, replacing their native counterparts (email client and office suite). Due to the immense importance of the web browser for the interaction with the user, it became a central component of almost every modern operating system: Microsoft has Internet Explorer, Apple has Safari, and Google is building ChromeOS, an operating system based entirely on its web browser Chrome. Furthermore, system-independent browsers such as Opera also contribute to the highly competitive and diverse browser market.

While today's browsers interpret a website's code in similar ways (based on standards), the actual implementations of these standards differ. This diversity of browsers has always caused headaches for Web developers, as the same website can vary across different browsers with respect to functionality or appearance, requiring additional testing and debugging of a website's code in order to ensure correct functionality in relevant browsers. However, this can also have severe implications on privacy and security. In this paper, we propose a novel concept for browser identification, which exploits exactly these imperfect implementations of standards in the different browsers. Our work was originally motivated by the security scanner *nmap*, which uses TCP/IP stack fingerprinting to determine the operating system of a remote host. In a very similar way, we use the browser's underlying JavaScript engine for browser identification. While implementation differences in HTML5 or CSS could also be used for fingerprinting, we decided to base our approach on JavaScript as it is well established, supported by all major browsers, and works on mobile devices such as smartphones and tablets. JavaScript is furthermore used by a very high percentage of websites, and enabled by default on all major browsers. While other methods for server-side browser identification exist (in particular and by design, the User-Agent string), our approach can be considered more robust. While the User-Agent string can be set to an arbitrary value by the user, the JavaScript fingerprint is authentic for each browser and cannot be easily ported to a different browser. It is not easily possible to use a modified version of e.g., Internet Explorer with SpiderMonkey, the JavaScript engine of Mozilla's Firefox in order to obfuscate the actual browser in use.

In particular, the contributions of this paper are as follows:

- We propose a new method to reliably identify a browser based on the underlying JavaScript engine. Our method is more than three orders of magnitude faster than previous work.
- We show the feasibility and reliability of our method with a survey.

- We show how this can be used to detect modified UserAgent strings, used, for example, by the Tor browser bundle to increase the size of the anonymity set of its users.
- We propose an iterative protocol for server-side detection of session hijacking using browser fingerprinting.
- Raise awareness for such advanced fingerprinting methods, and discuss measures to protect users.

The rest of the paper is organized as follows: Section II gives the technical background. Our method for browser identification based on fingerprinting the JavaScript engine is introduced in Section III. We show the feasibility of browser identification with JavaScript engine fingerprinting in Section IV and discuss our results as well as possible countermeasures in Section V. Related work is presented in Section VI before we conclude in Section VII.

II. Background

Today's browser market is highly competitive. Browser vendors publish new versions in ever-shorter time spans and regularly add new features, with especially mobile browsers for smartphones and tablets on the rise. Many of these updates increase the overall performance of the browser in order to enhance the user experience and reduce loading times: just-in-time compilation (JIT) of Javascript, for example, allows dynamic compilation of Javascript and became part of the Javascript engines in Firefox and Google Chrome's V8 quite recently, among others. Using the GPU for rendering in Internet Explorer's Chakra engine is yet another feature that was introduced recently and increased browser performance considerably. Sandboxing the browser or specific parts, like the Flash plugin, was introduced to increase the overall browser security and to combat the widespread use of Flash-based security exploits.

Javascript has been standardized as ECMAScript [8], and all major browsers implement it in order to allow client-side scripting and dynamic websites. Traditionally, Web developers use the *UserAgent* string or the navigator object (i.e., *navigator.UserAgent*) to identify the client's Web browser, and load corresponding features or CSS files. The UserAgent string is defined in RFC2616 [11] as a sequence of product tokens and identifies the software as well as significant subparts. Tokens are listed in order of their significance by convention. The navigator object contains the same string as the UserAgent string. However, both are by no means security features, and can be set arbitrarily by the user.

Mowery et al. [22] recently implemented and evaluated browser identification with Javascript fingerprinting based on timing and performance patterns. In their paper, the authors used a combination of 39 different well-established Javascript benchmarks, like the SunSpider Suite 0.9 and the V8 Benchmark Suite v5, and generated a normalized fingerprint from runtime patterns. Even though these artificial Javascript benchmarks, such as SunSpider, do not necessarily reflect real-world Web applications [28], using their patterns for fingerprint generation is a convenient approach. In total, the runtime for fingerprinting was relatively high, with 190 seconds per user on average (caused partly by an intentional delay of 800ms between tests). Our approach is superior in multiple ways: (1) It's runtime is **more than three orders of magnitude faster** (less than 200ms on average compared to 190s), while having a comparable overhead for creating and collecting fingerprint samples. (2) It can be implemented in just a few hundred lines of Javascript and is undetectable for the user, as the CPU is not stalled noticeably. (3) Many recent browser versions stall the execution of Javascript from tabs and browser windows that are currently not visible to the user to increase the responsiveness of the currently active windows. This, however, could severely distort the timing patterns generated from [22] and was not addressed in the paper.

For the rest of the paper we will use the following terminology due to sometimes ambiguous usage of the term browser fingerprinting in the literature: the fingerprinting in our approach refers to Javascript fingerprinting, not the browser. We use Javascript engine fingerprinting to reliably identify a given browser, and for identifying the browser itself as well as the major version number. Related work (like Panopticlick [7]) uses the term browser fingerprinting for identifying a particular browser instance.

III. Design

For our fingerprinting method, we compared test results from openly available Javascript conformance tests and collected results from different browsers and browser versions for fingerprint generation. These tests cover the ECMAScript standard in version 5.1 and assess to what extent the browser complies with the standard, what features are supported and specifically which parts of the standard are implemented incorrectly or not at all. In essence, our initial observation was that the test cases that fail in, e.g., Firefox, are completely different from the test cases that fail in Safari. We started working with Google's *Sputnik* test cases, but later switched to *test262*[1]. *test262* is the official TC39 test suite for ECMAScript, it is still improved regularly and is a superset of the *Sputnik* test cases. For ensuring comparability within our results from the experiments in Section IV we used *test262* from mid-January 2012, which includes 11,148 unique test cases for desktop browsers, while for the mobile browsers we used an updated version of *test262* with 11,570 test cases. However, the ECMAScript standard as well as the test suite are constantly updated, leaving enough future testing capabilities for Javascript engine fingerprinting. Running the full *test262* suite takes approximately 10 minutes on a desktop PC, while on smartphones and tablets it takes between 45 minutes and an hour, depending on the underlying hardware.

[1] http://test262.ecmascript.org

A. Efficient Javascript Fingerprinting

While Javascript conformance tests like *Sputnik* or *test262* consist of thousands of independent test cases, not all of them are necessary for browser identification. In fact, a single test case may be sufficient, e.g., to distinguish two specific browsers - if one of the browsers fails in a particular test case, while the other one does not, and assuming a priori that only these two browsers are within the set of browsers to test, this single test case is already enough to distinguish them. An example: Opera 11.64 only fails in 4 out of more than 10,000 tests cases from mid-January, while the most recent version of Internet Explorer 9 at that time failed in almost 400 test cases. If the test set contains only those two browsers, and the goal is to distinguish whether the client is using Opera 11.61 or Internet Explorer 9, a single test from the 400 failed test cases of Internet Explorer 9 (that are not within the set of 4 failed test cases from Opera) is sufficient to reliably distinguish those two browsers, and can be executed within a fraction of a second.

To formalize this approach: the **test set** of browsers is the set of browsers and browser versions that a given entity wants to make reliably distinguishable, in our case with Javascript engine fingerprinting. First, each browser is tested with *test262*. The results are then compared, and a **minimal fingerprint** is calculated for each browser (in relation to the other browsers in the test set). The use case for the minimal fingerprint is a web server that wants to assess whether a UserAgent string from the client is forged with respect to the other browsers in the test set. The web server can **verify** the browser. For efficiency, one of the requirements is that the fingerprint for each browser is as small as possible. The details of our implementation and the algorithm for generating the minimal fingerprints can be found in Section III-B.

Another use case of our method is to calculate a **decision tree**: instead of fingerprinting a particular browser with respect to the test set, we propose to build a binary decision tree to iteratively **identify** the browser in multiple rounds. The use case for this method is that the web server wants to identify the browser used under the assumption that the UserAgent might be forged. This method allows a larger test set than using minimal fingerprints while reducing the execution time on the client side. The pseudocode for calculating a minimal decision tree can be found in Section III-C.

B. Minimal Fingerprint

We use a greedy algorithm to find a (possibly minimal) fingerprint for a given test set: We start by running *test262* for each of the browsers in the test set, and calculate the number of browsers that fail for each test case. As the JavaScript engine is a static part of the browser, this needs to be done only once per browser. We then compare the results of *test262* for the browsers within the test set and calculate for each failed test case the number of browsers that fail. We call the total number of browsers that fail a particular test case the *uniqueness* u (with respect to the test set). We then select a test case with $u = 1$ at random and remove the browser from the test set, as this test uniquely identifies this browser. The uniqueness u is then recalculated for the remaining test cases. The process is repeated until either a unique fingerprint has been found for every browser, or no test case with $u = 1$ is found. In the latter case, we change our selection and choose a different test case until either a minimal test set is found or no solution can be found. An alternative approach would be to use some form of machine learning to find minimal fingerprints, but our results indicate that this is not (yet) necessary and our simplistic, greedy algorithm works well in practice. With the resulting set of fingerprints it becomes possible to assess whether a browser is what it claims to be: if all the tests of the minimal fingerprint for that browser fail, and no minimal fingerprints for the other browsers from the test set do, the browser is uniquely identifiable with respect to the test set. To make the algorithm and, in consequence, browser fingerprinting more resilient against errors, multiple tests could be used per browser (in case the user's browser is not part of the test set and the UserAgent string is not used to check this beforehand).

However, a basic assumption here is that the browser is included in the test set during fingerprint calculation in the first place. If the browser is not in the test set, false positives could occur if the engine is similar to one of the fingerprints (with respect to the minimal fingerprint). It is also possible to dynamically extend the test set: If a new UserAgent string is encountered that was not part of the test set, fingerprints could be recalculated on the fly to determine whether the UserAgent correctly identifies the browser: Instead of using the precalculated fingerprints, the browser is added to the test set, fingerprints are recalculated, and the identification process starts again with new minimal fingerprints for all browsers in the test set. This would allow *relative fingerprinting* over time and could be used to verify only the, e.g., five most popular browser versions for the previous month or day.

The minimal set of failed test cases for the four common browsers from 2012 shown in Table I to illustrate minimal fingerprints. The browsers in the test set are Firefox 12, Opera 11.64, Internet Explorer 9 and Chrome 20, with a resulting minimal fingerprint consisting of only 4 tests. With the algorithm explained above, we calculate the minimal fingerprints as follows: For every test case, the uniqueness in the test set is calculated. If a test fails for a specific browser, it receives a check mark in the table, and if the browser does not fail that test, it is crossed out. While this seems counter-intuitive, the check mark highlights the potential to use this particular test case for fingerprinting, as the number of failed test cases is much smaller than the number of tests passed. One of the test cases with $u = 1$ is selected at random, in the example this is 13.0-13-s. This test then becomes the minimal fingerprint for Internet Explorer 9, and

Internet Explorer is removed from the set of browsers that do not yet have a fingerprint. The uniqueness is recalculated, and another test case is selected at random with $u = 1$, e.g., 10.6-7-1, which becomes the minimal fingerprint for Firefox 12. Next, Opera gets 15.4.4.4-5-c-i-1 as fingerprint, and Chrome S15.8.2.16_A7. If a web server now tries to verify a given UserAgent, all 4 tests are sent for execution to the client, and the web server can verify the UserAgent with respect to the test set if only one test fails (in this example).

C. Building a Decision Tree

To identify a user's browser without relying a priori on the UserAgent, we build a binary decision tree for a given test set and assess if the browser is included in it by running multiple test rounds. For every test, we step down one level of the decision tree until we finally reach a leaf node. Inner nodes in this decision tree are test cases, while the edges show whether the browser fails that test or not. Instead of calculating a unique fingerprint for each browser in the test set, we need to identify the test cases that can be used to split the number of browsers that fail (respectively pass) equally. Multiple rounds of discriminating test cases can thus be used instead of calculating the minimal fingerprints for large test sets. The decision tree can reduce the total number of executed test cases considerably for such large test sets, making browser identification much faster. The decision tree is especially useful if the test set and the total number of test cases for the minimal fingerprints are rather large.

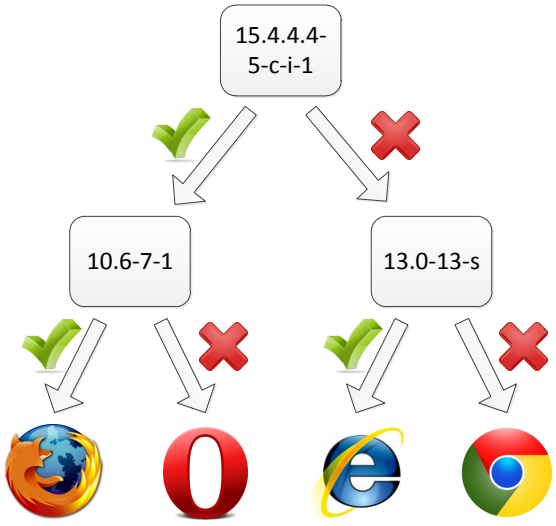

Fig. 1. Decision tree for Table I

To calculate a decision tree, we adapt the algorithm above slightly. We start again by calculating the *uniqueness* u for each *test262* test case that fails, sort the list and pick the test that splits the set into halves as the first test in our tree. If there is no such test, we select the statistical mode. We then continue to split the array of browsers into two parts, and recursively repeat this until we have built a complete decision tree with all the browsers from the test set. No assumptions can be made for the distribution of failed test cases, which means that in the worst case the tree can become a linear list instead of a tree if all failed tests have uniqueness $u = 1$. Again, if no tree can be found using the statistical mode, we can slightly vary the choice of test cases for the inner nodes and rerun the algorithm. In the ideal case, every inner node in the tree splits the subset of browsers in the test set in half, and the total number of tests that need to be executed at the client is only $O(logn)$ compared to $O(n)$ for executing the minimal fingerprints. Referring to the example from Section III-B, we can construct a decision tree as follows (cf. Table I): We start again by calculating the uniqueness for every test case of every browser that fails. We sort the results, and pick test 15.4.4.4-5-c-i-1 as our root note, because it splits the test set perfectly into halves. We then select the tests 10.6-7-1 and 13.0-13-s as the child nodes, and can identify the browser by running only two test cases, instead of four with the minimal fingerprinting approach. The resulting decision tree is shown in Figure 1. As with the algorithm for calculating minimal fingerprints, the algorithm is straightforward and fast to implement and execute on the client as well as on the server - it works well across different browsers and versions, thus negating the need for a more complex algorithm.

D. Implications on Security and Privacy

While the UserAgent string is traditionally used to report the web browser and version to a server, this is often not sufficient as the user can change it arbitrarily. In the context of browser **security**, current malware often relies on vulnerabilities in browsers (besides plugins like Flash) for launching exploits. Especially exploit kits like Blackhole [16] have been shown to use the UserAgent String to exploit client-side vulnerabilities. It is furthermore well known in the security community that Javascript and drive-by-download attacks can be used to endanger client security and privacy [3], [32], [4]. For the implications to privacy we use the security model of Tor [6] and the definition of an anonymity set [27], which could be decreased by a malicious website using JavaScript engine fingerprinting. Section VI discusses numerous recent papers that have been using browser fingerprinting to endanger user's online privacy.

In our threat model we assume that an attacker has the capabilities to host a website and direct users to it. The victim then fetches and executes Javascript code on the client side. This can be done e.g., by renting advertisement space, or with social engineering attacks where the user is tricked into opening a malicious website. This is already happening with malware on a large scale, and everything necessary to conduct such attacks can be purchased relatively easily. This is of relevance for our work, as malware authors could use browser fingerprinting to use it for increasing reliability of

Web browser	15.4.4.4-5-c-i-1	13.0-13-s	S15.8.2.16_A7	10.6-7-1	15.2.3.6-4-410
Opera 11.64	✓	✗	✗	✗	✗
Firefox 12.0	✓	✗	✗	✓	✗
Internet Explorer 9	✗	✓	✗	✗	✗
Chrome 20	✗	✗	✓	✗	✓
Uniqueness u	2	1	1	1	1

TABLE I
TESTS FROM *test262* AND THEIR USABILITY FOR BROWSER IDENTIFICATION

their exploits, thwart sandboxed environments like Wepawet[2] and to increase the stealthiness of their malware: instead of relying on the UserAgent string to find out if a victim is exploitable, Javascript fingerprinting could be used. The bar to incorporate this is low, and could be of significance for the arms race between malware authors and security research in the future. Detection of such malicious code would be considerably harder, and we aim to increase awareness for security researchers of such sophisticated browser fingerprinting methods. More work is needed to assess if this or similar fingerprinting is already used by malware in the wild.

E. Benign Uses of Fingerprinting

Here we discuss some benign use cases in addition to the sections discussing the framework and our results, respectively. To protect against session hijacking, web servers could use JavaScript engine fingerprinting to verify or refute validity of HTTP sessions, as session hijackers usually clone all possibly identifying plaintext information like session cookies (e.g., Firesheep[3] or FaceNiff[4] do) or the complete HTTP header. With JavaScript engine fingerprinting such attacks become detectable at the server side, as modified UserAgents can be detected. Another way to secure HTTP sessions would be to constantly challenge the browser and add Javascript engine fingerprinting as an additional security layer: At the beginning of a session the browser is identified with minimal fingerprinting. For every latter request the webserver chooses a subset of random test cases and includes them in the JavaScript code, thus challenging the client. The overhead would be minimal and not noticeable to the client. If the responses do not correlate with the expected outcome, the session is terminated. While the attacker is able to see the challenges, he might not know the correct responses - the attacker is forced to (1) either use the same browser down to the very same version (which may be not possible, e.g., to run an Internet Explorer on Android), or (2) collect the fingerprints for his victim beforehand to fake the replies, which would be very time consuming. Thus Javascript fingerprinting can be used to raise the bar for session hijacking in the arms race against attackers. This method could also be used for connections that are secured with HTTPS to prevent HTTPS MITM attacks. Recently hacked CAs like DigiNotar or Comodo and "Operation Black Tulip"[5] have shown that HTTPS alone is simply not enough to secure online communication anymore. However, it cannot completely defy session hijacking as the attacker might for example simply relay the challenges to the actual client. We believe though that this would be a valid countermeasure against session hijacking as this can be added easily to existing web applications.

IV. RESULTS AND EVALUATION

To evaluate the possibility and power of Javascript fingerprinting, we implemented the methods outlined above. We collected different browser version and operating system combinations for desktop browsers as well as for mobile browser versions on smartphones and tablets for fingerprint generation in a database. An excerpt of the data can be seen in Table II[6]. To evaluate our method with respect to the security and privacy implications discussed in Section III-D, we first evaluate if it is possible to determine the actual browser behind a modified UserAgent as used by the Tor Browser Bundle on a large scale. We also conducted a survey and measure the performance impact of our method on the client side.

A. Destkop and Mobile Browsers

In total, we stored the *test262* results for more than 150 different browser version and operating system combinations, ignoring minor version upgrades of browsers that contained no changes in the underlying Javascript engine. While this may not sound like much, it includes all major browser releases from the last three years, which accumulates to approximately 98% of browser market share since 2008[7]. For desktop browsers we collected test results for fingerprint generation from the five most popular browsers on three different operating systems: Windows, OS X and Linux. Different mobile browser versions and their test results can be seen in Table III. Results for mobile browsers are focused on Android and iOS devices. While the setup files for desktop browsers are often freely available and were easy to collect, it was much more difficult for us to get access to a broad spectrum of older mobile browsers as it is not possible to revert the running operating system of a smartphone or a tablet to an older software version, among other reasons as

[2]https://wepawet.iseclab.org
[3]http://codebutler.com/firesheep
[4]http://faceniff.ponury.net
[5]http://www.enisa.europa.eu/media/news-items/operation-black-tulip/
[6]Please see the author's homepage for the full data set
[7]http://www.w3schools.com/browsers/browsers_stats.asp

Browser	Win 7	WinXP	Mac OS X		Browser	Win 7	WinXP	Mac OS X
Firefox 3.6.26	3955	3955	3955		Chrome 8	1022	1022	1022
Firefox 4	290	290	290		Chrome 10	715	715	715
Firefox 5	264	264	264		Chrome 11	489	489	489
Firefox 6	214	214	214		Chrome 12	449	449	—
Firefox 7	190	190	190		Chrome 13	427	427	—
Firefox 12	165	165	165		Chrome 14	430	430	430
Firefox 15	161	161	161		Chrome 16	420	420	420
Firefox 17	171	171	171		Chrome 17	210	210	210
Firefox 19	191	191	191		Chrome 18	35	35	35
					Chrome 19	18	18	18
IE 6 (Sputnik)	—	468	—		Chrome 21	9	9	9
IE 8 (Sputnik)	—	473	—		Chrome 23	10	10	10
IE 9	611	—	—		Chrome 25	17	17	17
IE 10	7	—	—					
					Safari 5.0.5	777	1585	1513
Opera 11.52	3827	3827	3827		Safari 5.1	777	853	—
Opera 11.64	4	4	4		Safari 5.1.2	777	777	776
Opera 12.02	4	4	4		Safari 5.1.7	548	548	547
Opera 12.14	9	9	9					

TABLE II
SELECTION OF BROWSERS AND THEIR FAILED TEST CASES FROM *test262* (AND *Sputnik*)

Browser	OS	Device	# of fails
Safari	iOS 5.1.1	iPhone 4S	988
Safari	iOS 6.1.2	iPhone 4	28
Browser	Android 2.2	GalaxyTab	2130
Browser	Android 2.3.7	HTC Desire	1328
Browser	Android 4.0.3	GalaxyTab2	588
Browser	Android 4.0.4	Nexus S	591
Browser	Android 4.1.2	Nexus S	23
Chrome 18	Android 4.0.3	GalaxyTab2	46
Firefox 19	Android 4.0.3	GalaxyTab2	191

TABLE III
NUMBER OF FAILED TEST CASES FOR MOBILE BROWSERS

protection against jailbreaking.

As the Javascript engines are not as dynamic as the numbering scheme of browser vendors, equal results are obtained with consecutive browser versions if the underlying Javascript engine has not been changed. For example, the major Firefox version numbers often indicate changes in the underlying principles, while the minor numbers are used for, e.g., security updates: Updates 6.0.1 and 6.0.2, for example, were used to solely remove the Dutch certificate authority Diginotar, which got hacked and was used for issuing rogue certificates [25]. The results are discussed in detail in Section V.

B. Tor Browser Bundle

While modifying the UserAgent can be used to hide a user's browser and version, JavaScript engine fingerprinting can be used to reliably identify the web browser of a user. The Tor Browser Bundle is using modified UserAgent strings on a large scale, and we will show how these modified UserAgents can be detected by web servers. The Tor network [6] is an overlay network that provides online anonymity to its users by hiding the user's IP address. At the time of writing it is estimated to be used by more than 500,000 users every day[8]. It has been previously shown that the majority of Tor users do not browse the Web securely [15], and Javascript engine fingerprinting can be used to further increase the attack surface for sophisticated de-anonymization attacks. The *Tor Browser Bundle* (TBB) is a convenient way to use the Tor anonymization network with a known, secure configuration and without the need for the user to install any software. It is available for Windows, Linux and OS X and has many important privacy-enhancing features enabled by default, e.g., TorButton or HTTPS Everywhere, prepackaged and preconfigured, making it the recommended way to use the Tor network securely at the time of writing. By default, the Tor Browser Bundle changes the UserAgent string to increase the size of the anonymity set [5]. In the Tor Browser Bundle the UserAgent is uniformly changed to Firefox 5.0 while the shipped browser often uses a more recent version. Mozilla releases new versions of Firefox every six weeks. The numbers of the actual and the expected results from *test262* running on Windows 7 can be seen in Table IV. A decision tree similar to the example in Section III-C can be constructed to minimize the number of tests needed to

[8]https://metrics.torproject.org/users.html

accurately identify the browser used with the Tor Browser Bundle. In the most recent versions of TBB the browser was changed to Firefox 17 with long-term support, and the UserAgent is correctly identifying the browser as Firefox 17.

Care has to be taken when interpreting the implications of Javascript engine fingerprinting on the Tor network: Even though Javascript is not disabled by default in the Tor Browser Bundle [1], the only information the malicious website operator obtains is that the user is, in fact, using a different version of Firefox than indicated. The web server can already easily determine that the user is using the Tor network by comparing the client's IP address to the public list of Tor exit relays. However, Javascript fingerprinting can reduce the size of the anonymity set of all Tor users, and can harm anonymity to a yet unknown extent.

C. Experimental Survey

To evaluate the performance and practicability of our fingerprinting method, we conducted a survey among colleagues, students and friends for a duration of several weeks in 2011 to find out (1) whether our method was working reliably, and (2) to measure the time and bandwidth needed for fingerprinting. The test set consisted of Firefox 4, Chrome 10 and Internet Explorer 8 & 9, which were the top browsers at that time and had a cumulative worldwide market share of approx. 66% at that time. For each of the browsers in our test set, we manually selected 10 failed test cases from the Sputnik test suite to be run on the client instead of the minimal fingerprint, to increase accuracy and decrease the possibility of false positives. As a result, every client executed 40 test cases in total, and the failed test cases were then used to determine the user's Javascript engine. Due to the automatic update function of Google Chrome, the version number changed from 10 to 12 during the testing period, but the 10 test cases we had selected for Chrome did not change, so the updates did not skew our results and Chrome was still correctly identified even though the Javascript engine changed with the updates (see Table II). Users were directed to a webpage where they were asked to identify their web browser manually using a dropdown menu and to start the test. As a ground truth to evaluate our fingerprinting, we relied on the UserAgent string in combination with the manual browser classification by the users. The Javascript file containing the 40 tests as well as the testing framework had a size of 24 kilobytes, while each of the 10 tests per browser were only between 2,500 and 3,000 bytes in size. The results were written to a database. We used a cookie to prevent multiple test runs by the same browser, and also blocked submissions with the same UserAgent string and IP address that originated in close temporal vicinity in case the browser was configured to ignore cookies.

In total, we were able to collect **189** completed tests. From those 189 submissions, 175 were submitted by one of the four browsers covered by the test set, resulting in an overall relative coverage of more than 90%. 14 submissions were made with browsers not in the test set, mainly smartphone web browsers. We compared the results of Javascript fingerprinting with the UserAgent string as well as the user choice from the dropdown menu, and Javascript fingerprinting had the correct result for all browsers in the test set. In one case our method identified a UserAgent string manipulation, as it was set to a nonexistent UserAgent. In 15 cases, the users made an error identifying their browser manually from the dropdown menu, but the UserAgent and the results from fingerprinting matched. There were no false positives for the browsers within the test set; the algorithm for fingerprinting identified browsers if and only if all the test cases for that browser failed and all tests for the other browsers did not fail. The runtime for the entire test was short, with 90ms on average for PCs and 200ms on average for smartphones (even though smartphones were not part of the test set).

V. Discussion

The results above show that JavaScript engine fingerprinting is a feasible approach to identify or verify a given browser, even for mobile devices like smartphones, with only small overhead regarding execution time on the client and bandwidth. On the server side the impact is negligible, as it can be implemented as a small number of database lookups. The "best" browser regarding Javascript standard conformance in our set of tested browsers was Opera, with only 4 failed tests in its most recent versions. Firefox and Chrome improved the engine constantly between releases, which happen at a much higher pace. Internet Explorer used a different XMLHttpRequest method before version 8 and thus did not work with *test262*, so we relied on the *Sputnik* tests and test numbers for fingerprint generation in Section IV-C. Please note that it is not the total number of failed test cases that is of importance, but if there is a difference between the browsers in the test set. For browser identification and with respect to the chosen test set, a single test case per browser is often sufficient to distinguish between two or more browsers. Also, these results and the number of failed tests are not static in nature: browsers, ECMAscript and the test suites are under active development and are constantly improved, with ECMAscript currently preparing version 6 of the standard (codename "Harmony").

While it is possible to detect a specific browser version with the algorithms discussed above, our method cannot be used to detect the underlying operating system (compared to the approach used in [22]). Other means are necessary to identify it as well as the underlying computing architecture (x86, x64, ...). Due to their complexity, JavaScript engines reuse their engine across different operating systems, as it nowadays takes multiple man-years to develop a modern JavaScript engine. All the latest browser versions at the time of writing that run on different operating systems and platforms seem to use the same Javascript engine. The only exception we

Version TBB	Browser	UserAgent	test262	exp. test262	Detectable
2.3.25-4	Firefox 17esr	Firefox 17	171	171	✗
2.3.25-2	Firefox 10esr	Firefox 10	172	172	✗
2.2.35-9	Firefox 12.0	Firefox 5.0	165	264	✓
2.2.35-8	Firefox 11.0	Firefox 5.0	164	264	✓
2.2.35-3	Firefox 9.0.1	Firefox 5.0	167	264	✓
2.2.33-2	Firefox 7.0.1	Firefox 5.0	190	264	✓
2.2.32-3	Firefox 6.0.2	Firefox 5.0	214	264	✓
2.2.30-2	Firefox 5.0.1	Firefox 5.0	264	264	✗
2.2.24-1	Firefox 4.0	Firefox 3.6.3	290	3956	✓

TABLE IV
DETECTABILITY OF USERAGENT STRING MANIPULATIONS IN TBB

could find were (mostly historic) versions of Safari, where the same version number on different operating systems used different versions of the Javascript engine (see Table II). For all the other browsers we tested, the version number convention across operating systems seems to correlate with the Javascript engine. We could show on the other hand that operating system detection for smartphones and tablet PCs is possible, and that we can easily distinguish between e.g., Android or iOS with our method. Due to the larger update cycles compared to desktop browsers, and due to the fact that there are still a lot of old Android versions in use, JavaScript engine fingerprinting is thus especially dangerous for mobile devices. It is furthermore possible to distinguish between a mobile browser and one running on a regular computer easily, if both are included in a test set. However, our sample size for mobile devices is much smaller compared to our desktop browser dataset - more work is needed in this area.

A. Countermeasures

It is naive to believe that Javascript engines across different browsers will conform uniformly with the standard in the future due to the complexity of the Javascript engines. As this is unlikely to happen in the near future, we propose preventing or detecting fingerprinting on the client side. Client-side protection could be done either by the browser itself [4], a browser extensions looking for fingerprinting of any kind, or by using a proxy server that can detect and block fingerprinting patterns similar to TCP/IP stack fingerprinting prevention methods [30]. We are currently working on a browser extension that can detect Javascript fingerprinting, and hope to work on a proxy solution in the near future as well.

B. Future Work

Future work towards browser fingerprinting includes other core features of browsers that are not yet uniformly implemented, such as HTML5 or CSS3. We plan to add these to the fingerprint generation process, to decrease overall runtime and the computational overhead even further, and to make our approach work with browsers that have Javascript disabled. We also plan to assess whether current advertising networks [29] are already using Javascript fingerprinting, just as they were recently found to already use tricks to spawn almost undeletable cookies like *evercookie* [18], Flash cookies [31] or ETag respawning [2]. We are also working on a framework that can detect and prevent session hijacking on insecure connections (with or without SSL alike), as proposed in Section III-E.

VI. RELATED WORK

Javascript has recently received a lot of attention with the rise of AJAX as a new programming paradigm and especially with respect to client-side security [3], [13] and privacy [17]. Cross-site scripting (XSS) as one of the most prevalent online security vulnerability in fact only works when a browser has Javascript enabled.

Fingerprinting in general has been applied to a broad and diverse set of software, protocols and hardware over the years. Many implementations try to attack either the security or the privacy aspect of the test subject, mostly by accurately identifying the exact software version in use. One of the oldest security-related fingerprinting software is *nmap* [12], which is still used today and uses slight differences in the implementation of network stacks to identify the underlying operating systems and services. OS fingerprinting is often a crucial stepping stone for an attacker, as remote exploits are not uniformly applicable to all versions of a given operating system or software. Another passive fingerprinting tool, *p0f*[9], uses fingerprinting to identify communicating hosts from recorded traffic. Physical fingerprinting, on the other hand, allows an attacker to identify (or track) a given device, e.g., using specific clock skew patterns, which has been shown to be feasible in practice to measure the number of hosts behind a NAT [19], [24]. History stealing [26], [33] has been shown to be another, effective attack vector to de-anonymize users and could be used for browser fingerprinting as well. User tracking is yet another threat to the privacy of users, which is, e.g., used heavily by advertising networks [29], [21].

[9]http://lcamtuf.coredump.cx/p0f3

In recent years, the focus shifted from operating system fingerprinting towards browser and HTTP traffic fingerprinting in the area of security and privacy research. On the one hand, this was caused by the widespread use of firewalls as well as normalized network stacks and increased awareness of administrators to close unused ports. On the other hand, the browser has become the most prevalent attack vector for malware by far. This trend has been further boosted by the advent of cloud computing (where the browser has to mimic or control operating system functionality), online banking and e-commerce, which use a web browser as the user interface. Recent malware relies on fingerprinting to detect if the victim's browser is vulnerable to a set of drive-by-download attacks [9], [3]. For encrypted data, Web-based fingerprinting methods rely on timing patterns [10], [14], but at higher expenses in terms of accuracy, performance, bandwidth and time. The EFF's Panopticlick project[10] does browser fingerprinting by calculating the combined entropy of various browser features, such as screen size, screen resolution, UserAgent string, and supported plugins and system fonts [7]. Mayer was among the first to discuss technical features that can be used for browser fingerprinting [20]. In recent work, browser fingerprinting with the aim of harming the user's privacy has been used effectively solely by using the UserAgent string [34]. Another recent paper uses novel HTML5 features and WebGL to accurately fingerprint browsers [23] and the underlying hardware (GPU).

VII. Conclusion

In this paper, we introduced a method for reliable browser identification based on the underlying Javascript engine, and evaluated its feasibility in multiple ways. In a survey with 189 participants, our method identified all browsers within the test set correctly. We also evaluated the impact on systems like the Tor Browser Bundle that use a modified UserAgent string on purpose to increase the anonymity of users, and collected data for generating fingerprints for more than 150 browser and operating system combinations. We showed that this method can be used efficiently in terms of bandwidth and computational overhead, takes less than a second to run on the client, and can reliably identify a web browser without relying on the UserAgent string provided by the client.

References

[1] T. Abbott, K. Lai, M. Lieberman, and E. Price. Browser-based attacks on tor. In *Privacy Enhancing Technologies*, pages 184–199. Springer, 2007.

[2] M. D. Ayenson, D. J. Wambach, and A. Soltani. Flash Cookies and Privacy II: Now with HTML5 and ETag Respawning. 2011.

[3] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th international conference on World Wide Web*, pages 281–290. ACM, 2010.

[4] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. ZOZZLE: fast and precise in-browser JavaScript malware detection. In *USENIX Security Symposium*, 2011.

[5] R. Dingledine and N. Mathewson. Anonymity loves company: Usability and the network effect. In *Proceedings of the Fifth Workshop on the Economics of Information Security (WEIS 2006), Cambridge, UK*, June 2006.

[10]https://panopticlick.eff.org/

[6] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, pages 21–21. USENIX Association, 2004.

[7] P. Eckersley. How Unique is Your Web Browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.

[8] E. ECMAScript, E. C. M. Association, et al. ECMAScript Language Specification. Online at http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[9] M. Egele, E. Kirda, and C. Kruegel. Mitigating drive-by download attacks: Challenges and open problems. *iNetSec 2009–Open Research Problems in Network Security*, pages 52–62, 2009.

[10] E. W. Felten and M. A. Schneider. Timing Attacks on Web Privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 25–32. ACM, 2000.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol–HTTP/1.1, June 1999. *Status: Standards Track*, 1999.

[12] F. Gordon Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, 2009.

[13] O. Hallaraker and G. Vigna. Detecting Malicious Javascript Code in Mozilla. In *Proceedings 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, pages 85–94. Ieee, 2005.

[14] A. Hintz. Fingerprinting websites using traffic analysis. In *Proceedings of the 2nd international conference on Privacy enhancing technologies*, pages 171–178. Springer-Verlag, 2002.

[15] M. Huber, M. Mulazzani, and E. Weippl. Tor HTTP usage and information leakage. In *Communications and Multimedia Security*, pages 245–255. Springer, 2010.

[16] Imperva. Imperva Data Security Blog–Deconstructing the Black Hole Exploit Kit, 2011. Online at http://blog.imperva.com/2011/12/deconstructing-the-black-hole-exploit-kit.html.

[17] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 270–283. ACM, 2010.

[18] S. Kamkar. evercookie–never forget. *New York Times*, 2010.

[19] T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, 2005.

[20] J. R. Mayer. Any person... a pamphleteer: Internet anonymity in the age of web 2.0. *Undergraduate Senior Thesis, Princeton University*, 2009.

[21] J. R. Mayer and J. C. Mitchell. Third-party Web tracking: Policy and technology. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[22] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting Information in JavaScript Implementations. In *Proceedings of Web 2.0 Security and Privacy 2011 (W2SP)*, San Franciso, May 2011.

[23] K. Mowery and H. Shacham. Pixel Perfect: Fingerprinting Canvas in HTML5. *Proceedings of Web 2.0 Security and Privacy (W2SP) 2012*, 2012.

[24] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *Security and Privacy, 2005 IEEE Symposium on*, pages 183–195. IEEE, 2005.

[25] E. E. Network and I. S. Agency. Operation Black Tulip: Certificate authorities lose authority. *Press Release*, 2011.

[26] L. Olejnik, C. Castelluccia, and A. Janc. Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns. *5th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2012)*, 2012.

[27] A. Pfitzmann and M. Köhntopp. Anonymity, unobservability, and pseudonymity—a proposal for terminology. In *Designing privacy enhancing technologies*, pages 1–9. Springer, 2001.

[28] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 3–3. USENIX Association, 2010.

[29] F. Roesner, T. Kohno, and D. Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. In *Proceedings of the 9th USENIX Conference on Networked systems design and implementation (NSDI 2012)*. USENIX Association, 2012.

[30] M. Smart, G. R. Malan, and F. Jahanian. Defeating TCP/IP stack fingerprinting. In *Proceedings of the 9th USENIX Security Symposium*, volume 24, 2000.

[31] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle. Flash Cookies and Privacy. *SSRN preprint (August 2009) http://papers. ssrn. com/sol3/papers. cfm*, 2009.

[32] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson. I Still Know What You Visited Last Summer: Leaking Browsing History via User Interaction and Side Channel Attacks. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 147–161. IEEE, 2011.

[33] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel. A practical attack to de-anonymize social network users. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 223–238. IEEE, 2010.

[34] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi. Host Fingerprinting and Tracking on the Web: Privacy and Security Implications. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012)*, February 2012.