Fast and Efficient Browser Identification with JavaScript Engine Fingerprinting

Technical Report TR-SBA-Research-0512-01

Martin Mulazzani*, Philipp Reschl, Markus Huber*,
Manuel Leithner*, Edgar Weippl*

*SBA Research
Favoritenstrasse 16
AT-1040 Vienna, Austria
`mmulazzani@sba-research.org`

Abstract. While web browsers are becoming more and more important in everyday life, the reliable detection of whether a client is using a specific browser is still a hard problem. So far, the UserAgent string is used, which is a self-reported string provided by the client. It is, however, not a security feature, and can be changed arbitrarily.
In this paper, we propose a new method for identifying Web browsers, based on the underlying Javascript engine. We set up a Javascript conformance test and calculate a fingerprint that can reliably identify a given browser, and can be executed on the client within a fraction of a second. Our method is three orders of magnitude faster than previous work on browser fingerprinting, and can be implemented in just a few hundred lines of Javascript. Furthermore, we collected data for more than 150 browser and operating system combinations, and present algorithms to calculate minimal fingerprints for each of a given set of browsers to make fingerprinting as fast as possible. We evaluate the feasibility of our method with a survey and discuss the consequences for user privacy and security. This technique can be used to enhance state-of-the-art session management (with or without SSL), as it can make session hijacking considerably more difficult.

1 Introduction

Today, the Web browser is a central component of almost every operating system. Microsoft has Internet Explorer, Apple has Safari, and Google is building an operating system based on its Web browser Chrome [35]. For Web developers, the diversity of browsers has caused headaches, as each browser implements standardized features in a way that often is different from other browser and not one hundred percent according to the standard. This imperfect standard conformance is what we use to derive not only which browser a client uses, but also which version. So far, little research has been done to identify a Web browser reliably [31], as this field of research is relatively new. We will show that it is feasible and beneficial in the context of user privacy and security to identify the browser by one of its core components, the Javascript engine. Our work is originally motivated by *nmap*, which uses TCP/IP stack fingerprinting for determining the operating system of a remote host, except that we determine the Web browser and version of a client instead.

This paper is organized as follows: The technical background of Javascript as well as related work are presented in Section 2. In Section 3, we introduce our approach and discuss the benefits and caveats of browser fingerprinting in general. We show the general feasibility of Javascript fingerprinting in Section 4 and present our experimental results in Section 5. We conclude by proposing various countermeasures against Javascript browser fingerprinting in Section 6.

2 Background

Today's browser market is highly competitive: browser vendors publish new versions in ever-shorter time spans and regularly add new features. Five popular browsers currently compete for market shares, 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 quite recently became part of the Javascript engines in Firefox and Google Chrome's V8, among others. It offers major speed improvements for websites that rely heavily on Javascript. 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 engine, were introduced to increase the overall browser security and to combat the widespread use of flash-based security exploits.

One of the drawbacks of the competition in the Web browser market is that not every feature is implemented uniformly: websites can look completely different viewed with two different browsers. Javascript has been standardized as ECMAScript [11], 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 [14] 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. E.g., the UserAgent for Firefox 10.0.1 on Windows 7 looks like this: *Mozilla/5.0 (Windows NT 6.1; WOW64; rv:10.0.1) Gecko/20100101 Firefox/10.0.1*, where Windows NT 6.1 WOW64 refers to Windows 7 with 64bit architecture, Gecko identifies the browser platform, and Firefox 10.0.1 identifies the exact browser version. The navigator usually contains the same string as the UserAgent string. However, both are by no means security features, and can be set arbitrarily by the user.

Fingerprinting in general has been applied to a broad and diverse set of software, protocols and hardware over the years. Many implementations tried 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* [28], which is still used today and uses slight differences in the implementation of network stacks to identify the underlying operating systems and services. Another passive fingerprinting tool, *p0f* [47], uses fingerprinting to identify communicating hosts from recorded traffic. Recent work concentrates on generating fingerprints in an automated fashion, with the help of machine learning [5] [38]. 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. Physical fingerprinting, on the other hand, allows an attacker to identify (or track) a given device, e.g., by a specific clock skew pattern, which has been shown to be feasible in practice to measure the number of hosts behind a NAT [26] [32].

2.1 Related Work

Javascript has recently received a lot of attention with the rise of AJAX [17], cloud computing [30] and especially with respect to client-side security [6,18] and privacy [23]. Novel attacks such as cross-site scripting (XSS) [43] in fact only work because the browser has Javascript enabled.

In recent years, the focus of fingerprinting software shifted from operating system fingerprinting towards browser and traffic fingerprinting. 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 [16] [22]. This trend was 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 [12] [6] [39]. For encrypted data, Web-based fingerprinting methods rely on timing patterns [13] [19] [33], but at higher expenses in terms of accuracy, performance, bandwidth and time. The EFF's Panopticlick project [1] does browser fingerprinting by calculating the combined entropy of various browser features, such as screen size, screen resolution, UserAgent string, as well as supported plugins and system fonts [10]. In recent work, browser fingerprinting has been used effectively to harm the user's privacy solely by using the UserAgent string [46]. History stealing has been shown to be another, effective attack vector to de-anonymize users and could be used for browser fingerprinting as well, even though the fingerprint would be more volatile [45]. Browser fingerprinting in general can be further enhanced by using our proposed method for browser fingerprinting, which has been recently shown to work in a closely related paper [31]. We will show in Section 3.1 how our method differs from the method described in [31].

3 JavaScript Engine Fingerprinting

For our browser fingerprinting method, we compared test results from openly available online Javascript conformance tests and collected various browsers results for fingerprint generation. In essence, our observation is that the test cases that fail for, e.g., Firefox, are completely different to the test cases that fail for Safari. 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 wrongly or not at all. We started to work with the *Sputnik* [2] test cases, but when the website was shut down we changed to *test262* [3]. *test262* is the official TC39 test suite for ECMAScript and is a superset of the *Sputnik* test cases. *test262* is still under development. For our experiments in Section 5 we used *test262* from Mid-January 2012, which includes 11148 unique test cases. These test suites run all the tests from their database, and in the end, the number of failed test cases is presented to the user. The user can then judge to what extent his current browser complies with the official ECMAScript standard - the fewer failed test cases, the better. While current Web browsers only fail in a few test cases, previous versions did not comply with the standard to the same extent. Especially early versions often have hundreds or even thousands of failed test cases, which can then be used for fingerprinting: If a specific test case that fails on old browsers fails for a user, the browser must be from the set of older browsers and cannot be a newer one that complies with this test case.

In total, we stored the results for more than 150 different browser version and operating system combinations, ignoring minor version upgrades of browsers that had no changes in the Javascript engine. While this may not sound like much, this includes all browser releases from the last three years that had modifications

[1] `https://panopticlick.eff.org/`
[2] `http://sputnik.googlelabs.com/`
[3] `http://test262.ecmascript.org`

to the Javascript engine, which accumulates to approximately 98% of browser market share since 2008, as reported by [42]. The Javascript engine is rather static and changes only a few times per year. Most of the minor browser updates were security-related and had no influence on the test results. An excerpt from the list of browsers we tested as well as the number of failed test cases can be found in Table 1 [4].

As the Javascript engines are often not as dynamic as the numbering scheme of browser vendors, similar results are sometimes obtained with consecutive browser versions, as the underlying Javascript engine has not been changed. For example, the major Firefox version numbers 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 removed the Dutch certificate authority Diginotar, which got hacked and was used for issuing rogue certificates for Iran, known as "Operation Black Tulip". The "best" browser regarding Javascript standard conformance in our set of tested browsers was Opera 11.61, with only 4 failed tests, the "worst" was Safari 4.0.3 (which is shipped with Mac OS 10.6) with more than 4,000. As can be seen from the numbers in Table 1, Opera did not change the underlying Javascript engine very much between updates, while Internet Explorer apparently has a different update policy and does not release different versions except for the major releases. Firefox and Chrome improved the engine constantly between releases. Internet Explorer used a different XMLHttpRequest method before version 8 and did not work with *test262*, so we relied on the *Sputnik* tests and test numbers for fingerprint generation in Section 5.2. However, it is not the total number of failed test cases that is of importance, as using this as a basis for identification is not fast enough - all the tests would need to be run, which takes a considerable amount of time. In the best case, only a single test case per browser would be needed to distinguish between two or more browsers. This makes browser identification very fast. The details of our method for fingerprinting can be found in Section 3.1.

Mowery et al. [31] implemented and evaluated Javascript fingerprinting based on timing and performance patterns. In their paper, they 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 [37], 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. Our approach is superior in multiple ways: (1) It is **more than three orders of magnitude faster** (less then 200ms compared to 190s on average), while having a comparable overhead for creating and collecting fingerprint samples.

[4] Comment for reviewers: we will publish the database as well as the tools and the resulting fingerprints under an open source license, and will include the link here once the paper is accepted

Browser	**Win7**	**WinXP**	**MacOS**	**Browser**	**Win7**	**WinXP**	**MacOS**
Chrome 7	1022	1022	1022	Firefox 3.5.19	3959	3959	3959
Chrome 8	1022	1022	1022	Firefox 3.6.20	3955	3955	3955
Chrome 9	1020	1020	1020	Firefox 3.6.26	3955	3955	3955
Chrome 10	715	715	715	Firefox 4	290	290	290
Chrome 11	489	489	489	Firefox 5	264	264	264
Chrome 12	449	449	—	Firefox 6	214	214	214
Chrome 13	427	427	—	Firefox 7	190	190	190
Chrome 14	430	430	430	Firefox 8	167	167	167
Chrome 15	421	421	—	Firefox 9	167	167	167
Chrome 16	420	420	420	Firefox 10	163	163	163
Chrome 17	210	210	210				
Chrome 18 beta	35	35	35	IE 6 (Sputnik)	—	468	—
				IE 7 (Sputnik)	—	472	—
Opera 10.54	3834	3834	3834	IE 8 (Sputnik)	—	473	—
Opera 10.63	3834	3834	3834	IE 9	394	—	—
Opera 11.01	3828	3828	3828				
Opera 11.11	3828	3828	3828	Safari 4.0.3	—	—	4074
Opera 11.51	3827	3827	3827	Safari 5.0.5	777	1585	1513
Opera 11.52	3827	3827	3827	Safari 5.1	777	853	—
Opera 11.61	4	4	4	Safari 5.1.2	777	777	776

Table 1. Excerpt of our collected data – Number of failed test cases from *test262* (and *Sputnik*)

(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. So far, there is no known technique that detects or blocks our method. (3) Many recent browser versions introduced the stalling of Javascript execution from tabs and browser windows that are not currently visible to the user to increase the responsiveness of the currently active windows. This could severely distort the timing patterns generated and needs to be investigated.

Our method, however, has the drawback that we are unable to identify the underlying operating system, as all browsers that can run on multiple operating systems usually share the same codebase and Javascript engines are not developed separately for different operating systems due to their complexity. It takes multiple person-years to develop a modern Javascript engine, and they are usually compiled for several different architectures and operating systems. See Section 4.3 for details on how our method does (not) work for OS detection. We believe that a combined approach with timing pattern as well as Javascript engine fingerprinting should be used in case OS detection is needed.

3.1 Efficient Javascript Fingerprinting

While the Javascript conformance tests above consist of thousands of independent test cases, not all of them are necessary for browser fingerprinting. In fact, for specific questions, a single test case may be sufficient. The complexity increases with the number of browsers that are fingerprinted. For instance, Opera 11.61 only fails in 4 out of more than 10,000 tests, while the most recent version of Internet Explorer 9 fails in almost 400 test cases. If the testset 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 **testset** 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 testset). For efficiency, the fingerprint for each browser needs to be as small as possible. The details of our implementation and the algorithm in pseudocode for generating the minimal fingerprints can be found in Section 4.1.

Another use case of our method is to calculate a **decision tree**: instead of fingerprinting a particular browser with respect to the testset as fast as possible, we could build a decision tree to reveal the exact browser version used, but iteratively and without the need to generate a fingerprint for every browser in the testset. This method allows a much larger testset of browsers than when generating a fingerprint for each browser. The pseudocode for calculating a minimal decision tree can be found in Section 4.2.

3.2 Threats and Threat Model

While the UserAgent string is traditionally used to identify the Web browser to a server, this is often not sufficient as the user can change it arbitrarily. Especially with respect to privacy and security, new methods are needed to ensure secure browser detection, accountability and overall user security. In the context of browser **security**, current malware relies on browser identification for launching zero-day exploits, especially exploit kits like Blackhole [21] that are sold in the underground economy to reliably exploit client-side vulnerabilities. It is well known in the security community that Javascript and drive-by-download attacks can be used to endanger client security and privacy [6,44,7]. Tools like Wepawet [15] were built to detect such malicious code. Furthermore, it has been shown that current malware already tries to detect sandboxed environments like Anubis [27], and tools like Wepawet are likely to experience sandbox detection mechanisms in malware in the near future as well [36]. This is why we believe it is of great importance to develop countermeasures against well-known fingerprinting techniques, as well as to use them to improve the security of protocols and software. For the implications for privacy, we use the security model of Tor and the definition of an anonymity set [34], which could be decreased by a malicious website. Moreover, it has been shown recently that the wrong usage of complex systems, such as a peer-to-peer filesharing protocol, can endanger the user's privacy [4]. Web browsers and the combination of complex protocols and languages like HTML and HTTP are even harder to protect against sophisticated deanonymization attacks.

We identify the following threat model where attackers and security experts alike might use Javascript fingerprinting: 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 by renting advertisement space with embedded iframes, or by social engineering attacks where the user is tricked into opening a malicious website. This is already happening with malware on a large scale, and in the underground economy, everything necessary can be purchased relatively easily. This is of relevance for our work, as malware authors could use browser fingerprinting to thwart sandboxed environments, 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.

4 Evaluation

To evaluate the possibility and power of Javascript fingerprinting we implemented the methods outlined above, and collected more than 150 browser version and operating system combinations for fingerprint generation. An excerpt of the data can be seen in Table 1, but not every minor update of a browser changed the underlying Javascript engine. We took special care to avoid redundant test results for equivalent browser versions for Javascript fingerprinting and included only relevant browsers that are distinguishable with Javascript fingerprinting.

4.1 Minimal Fingerprint

We define the minimal fingerprint as the smallest number of tests needed per testset to fingerprint each browser uniquely. This is a relative measure that depends on the number and selection of browsers, and in the ideal scenario, there is one test case per browser. In the worst-case scenario, it is also possible that fingerprints are undefined if the browsers are not uniquely distinguishable. The minimal fingerprint is useful, e.g., if a website wants to assess whether the UserAgent string of a visitor corresponds to their actual browser.

We use a simple greedy algorithm to find a minimal fingerprint for a given testset. We then run *test262* for each of the browsers in the testset, and calculate the number of browsers that fail for each test case. We call the total number of browsers that fail in a particular test case the *uniqueness* u (with respect to the testset). We then select any test case with $u = 1$, and can use this test case to be the minimal fingerprint for that browser as it is the only browser in the testset that fails in that test case. The uniqueness u is then recalculated for the remaining test cases, but without the browser that has just been assigned a fingerprint. 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 compare browsers and failed test cases in pairs to see if a unique fingerprint can be found as a combination of multiple test cases, and iteratively increase the number of browsers per test case until a solution is found. If no solution can be found, we start over again and use a different test case with $u = 1$ for the first browser, which uses a minimal fingerprint with a single test for another browser.

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 testset do, the browser is uniquely identifiable with respect to the testset. However, a basic assumption here is that the browser is included in the testset during fingerprint calculation. In case the browser is not in the testset, false positives could occur if the engine is similar to one of the fingerprints (with respect to the minimal fingerprint). To overcome this problem, larger fingerprints could be calculated to enhance the resilience against new and unknown browsers, by including multiple test cases with $u = 1$. Some kind of scoring function could be used to keep the number of tests per fingerprint low, and would assign some confidence value to the outcome of the fingerprinting. If the browser was not in the testset beforehand, recalculation could happen on the fly to determine whether the UserAgent correctly identifies the browser: Instead of using the precalculated fingerprints, the browser is added to the testset, fingerprints are recalculated, and the identification process starts again with new minimal fingerprints for all browsers in the testset.

For the four most common browsers a sample of failed test cases is shown in Table 2. The browsers in the testset are Firefox 10.0.1, Opera 11.61, Internet Explorer 9 and Chrome 17, which are at the time of writing the most recent versions of these browsers. 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. According to the algorithm explained above, we calculate the minimal fingerprints as follows: For every test case, the uniqueness among the testset is calculated. 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 10. Next, Opera gets 15.4.4.4-5-c-i-1 as fingerprint, and Chrome S10.4.2.1_A1.

Larger fingerprints are always possible, but the example above identifies a minimal fingerprint for those four browsers, and shows that the most recent versions of these major web browsers can be identified easily and very cheaply in terms of computational costs.

Web Browser	15.4.4.4-5-c-i-1	13.0-13-s	S15.2.3.6_A1	10.6-7-1	S10.4.2.1_A1
Opera 11.61	✓	✗	✗	✗	✗
Firefox 10.0.1	✓	✗	✗	✓	✗
Internet Explorer 9	✗	✓	✗	✗	✓
Chrome 17	✗	✗	✓	✗	✓
Uniqueness u	2	1	1	1	2

Table 2. Suitability of *test262* test cases for fingerprinting

4.2 Building a Decision Tree

To find out the browser of a user without relying a-priori on the UserAgent, we would like to build a binary decision tree for a given testset, and see if the browser is included in it by running multiple rounds of tests. 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 at that test or not. Instead of calculating a unique fingerprint for each browser in the testset, we need to identify the test cases that can be used to split the number of browsers that fail (respectively pass) equally. It is not necessary to run all the minimal fingerprints from all the browsers in the testset. The decision tree can reduce the total number of executed test cases considerably, and a reliable identification of the underlying Javascript engine can be done much faster. The decision tree is especially useful if the testset and the total number of test cases for the minimal fingerprints is rather large.

To calculate a decision tree, we slightly adapt the algorithm above. 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 in two parts, and recursively repeat this until we have built a complete decision tree with all the browsers from the testset. No assumptions can be made for the distribution of failed test cases, which means that in the worst case, it is a linear list instead of a tree. Again, if a tree cannot be found with the statistical mode, we can slightly vary the choice of test cases for the inner nodes and rerun the algorithm. However, in the ideal case, every inner node in the tree splits the subset of browsers in half, and the total number of tests needed is only $O(logn)$ instead of $O(n)$ for running the minimal fingerprint for each browser.

Referring to the example from Section 4.1, we can construct a decision tree as follows: We start again by calculating the uniqueness for every test case of every browser that fails. We sort it, 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.

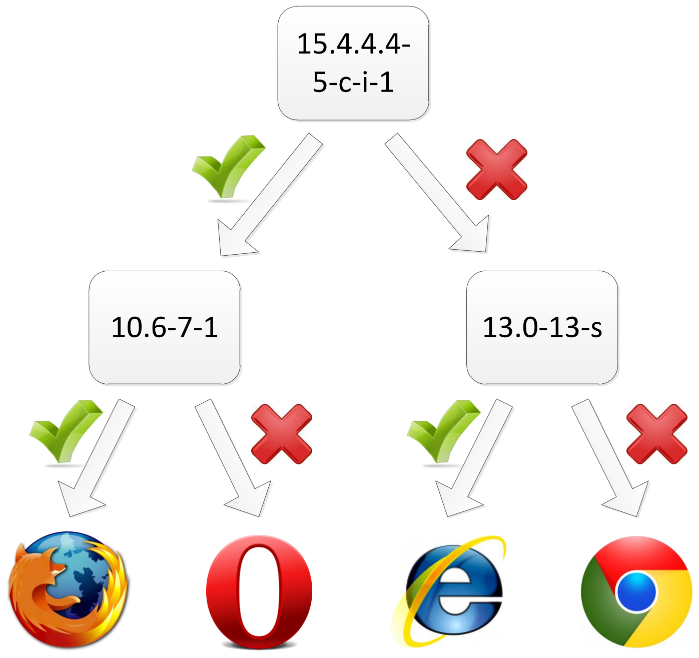

Fig. 1. Decision tree for table 2

4.3 Platform Detection

While it is possible to detect a specific browser with the algorithms discussed above quite fast, our method can, unfortunately, not be used to detect the underlying operating system. Other means are necessary to identify it, as well as the underlying computing architecture (x86, x64, ...). All the latest browser versions at the time of writing that run on different operating systems and platforms appear to have the same Javascript engine, with the most recent version *test262* results shown in Table 3.

	Win 7	Ubuntu 11.10	OS X Lion
Chrome 17	210	210	210
Firefox 10	163	163	163
Opera 11.61	4	4	4
Safari 5.1.2	777	—	777

Table 3. Number of failed test262 tests per OS

The only exception we could find were historic versions of Safari, as apparently the same version number on different operating systems relied on different Javascript engines, as can be seen in Table 1. For all the other browsers we tested, the version number convention across operating systems seems to correlate with the Javascript engine. However, this does not necessarily apply to mobile versions of the browsers, and future work is needed to assess whether browsers for smartphones can be identified with our method as well, i.e., whether it is possible to distinguish a smartphone from a regular computer.

5 Experimental Results

To evaluate our method with respect to the threat model discussed in Section 3.2, we first evaluated if it is possible to determine the actual browser behind a modified UserAgent with the Tor Browser Bundle. We also conducted a survey to measure the performance of our method.

5.1 Tor Browser Bundle

The Tor network [9] is an overlay network that provides online anonymity to its users by hiding the user's IP address. Users connect through a circuit of Tor relays, usually three, and while the relay closest to the client sees the user's IP address, the last relay can only see the connection from the previous relay. It has been shown that the majority of Tor users do not browse the Web securely [20] [4], and Javascript engine fingerprinting can be used to further increase the attack surface for sophisticated deanonymization attacks. Browsing the Web anonymously is an important feature of Tor, and is used heavily [29]. The *Tor Browser Bundle* (TBB) is a convenient way to use the Tor anonymization network without the need for the user to install any software, and is available for Windows, Linux and MacOS. It has all important privacy-enhancing features, like TorButton or HTTPS Everywhere, prepackaged and preconfigured, making

Version TBB	Browser	UserAgent	test262	exp. test262	Attackable?
2.2.35-3	Firefox 9.0.1	Firefox 5.0	167	264	✓
2.2.34-3	Firefox 8.0.1	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	✗
2.2.25-1	Firefox 4.0.1	Firefox 4.0	290	290	✗
2.2.24-1	Firefox 4.0	Firefox 3.6.3	290	3956	✓

Table 4. Detectability of UserAgent manipulation: Actual and expected Tor Browser Bundle test262 results

it the recommended way for using 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 [8].

In the most recent versions of the Tor Browser Bundle, starting at the beginning of 2011, the UserAgent is uniformly changed to Firefox 5.0, while the shipped browser uses often a more recent version. By running *test262*, it is easy to detect that the underlying Javascript engine is not the one the UserAgent string suggests. It is detectable by an attacker running a website or able to inject Javascript into the user's browser on the fly. As such, the recent change in policy from Mozilla to release new versions of Firefox every few months actually harms the Tor network. The numbers of the actual and the expected results from *test262* running on Windows 7 can be seen in Table 4. However, as already discussed in Section 4.3, we cannot detect the underlying operating system, which leaves the Tor Browser Bundle users some form of protection. A decision tree similar to the example from Section 4.2 (Figure 1) can be constructed to minimize the number of tests needed to accurately identify the browser used with the Tor Browser Bundle.

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 in fact is using a different version of Firefox than indicated. The webserver can already easily determine that the user is using the Tor network by comparing the IP address it sees and the public list of all Tor 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.

5.2 Representative Survey

To evaluate the performance and practicability of our fingerprinting method, we conducted a survey among colleagues, students and friends for multiple weeks in summer 2011 to find out (1) whether our method was working reliably, and to (2) measure the time and bandwidth needed for fingerprinting. The testset consisted of Firefox 4, Chrome 10 and Internet Explorer 8 & 9, which were the top browsers at that time and had a cumulative market share of approx. 66% [42]. For each of the browsers in our testset we selected 10 failed test cases from the Sputnik test suite to be run on the client. As a result, every client executed 40 test cases in total, and the failed test cases where then used to determine the user's Javascript engine. The details of the testset can be seen in Table 5, including the total number of failed *Sputnik* test cases. 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. 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 the fingerprints per browser were between 2,500 and 3,000 bytes in size. 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.

Web Browser	JS Engine	# Sputnik Errors
IE 8	JScript	471
Chrome 10	V8	136
Firefox 4	SpiderMonkey	181
IE 9	Chakra	72

Table 5. Browser testset in the survey

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 testset, resulting in an overall coverage of the testset by more than 90%. 14 submissions were made with browsers not in the testset, mainly smartphone Web browsers and only a few older browser versions. We compared the results from 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 testset. In one case Javascript fingerprinting identified a UserAgent 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 testset; 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 whole test was very short, with 90ms on average for PCs and 200ms on average for smartphones.

6 Countermeasures

The naive approach to countermeasures would suggest to make the different Javascript engines as uniform as possible, so that they would all conform to the standard. As this is very unlikely to happen in the near future, we propose preventing Javascript fingerprinting by preventing or detecting it on the client side. A simple solution would be to block all Javascript, but as more and more websites rely on Javascript, this is hardly feasible either. Client-side protection could be done either by the browser itself [7], or by using a proxy [39] that can detect and block fingerprinting patterns similar to TCP/IP stack fingerprinting prevention methods [40]. 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.

6.1 Future Work

Further enhancements to browser fingerprinting could include other novel Web browser features 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 work with clients that have Javascript disabled for security reasons. We also plan to assess if current advertising networks [25] are already using Javascript fingerprinting, just as they were recently found to already use tricks to spawn almost undeletable cookies like *evercookie* [24], Flash cookies [41] or ETag respawning [3].

Browser fingerprinting could also be used to enhance current session security by raising the bar for the attacker: instead of only cloning the session cookie (like Firesheep [5] does), the attacker would also need to clone the characteristics of the underlying Javascript engine. This method could also be used on connections that are secured with SSL, to prevent SSL MITM attacks. Deployed Web authentication frameworks like SessionLock [2], OpenID [6] or BrowserID [7] could include browser fingerprinting to harden their sessions by including the "what the user has" factor for a particular browser. Session hijacking would become detectable simply through the browser changing within a session, while other factors like the cookie or the IP (in case the attacker is within the same LAN) stay the same.

[5] `http://codebutler.com/firesheep`
[6] `http://openid.net`
[7] `https://browserid.org`

7 Conclusion

In this paper, we introduced a method for browser fingerprinting based on the underlying Javascript engine, and evaluated its feasibility in multiple ways. In a survey with around 190 participants, our method identified all browsers within the testset correctly. We also evaluated the impact on systems like the Tor Browser Bundle, which relies on changing the UserAgent string on purpose to increase the anonymity of users, and collected data for generating fingerprints for up to 150 browser and operating system combinations. We showed that this method can be used very efficiently in terms of bandwidth and computational overhead, takes less than a second to run, 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. B. Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceeding of the 17th international conference on World Wide Web*, pages 517–524. ACM, 2008.
3. M. AYENSON, D. WAMBACH, and A. Soltani. Flash Cookies and Privacy II: Now with HTML5 and ETag Respawning. 2011.
4. S. L. Blond, P. Manils, A. Chaabane, M. A. Kaafar, C. Castelluccia, A. Legout, and W. Dabbous. One bad apple spoils the bunch: Exploiting p2p applications to trace and profile tor users. In *Proceedings of the 4th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET 2011)*, March 2011.
5. J. Caballero, S. Venkataraman, P. Poosankam, M. Kang, D. Song, and A. Blum. Fig: Automatic fingerprint generation. *Department of Electrical and Computing Engineering*, page 27, 2007.
6. 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.
7. C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, 2011.
8. 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.
9. 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.
10. P. Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.
11. 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`.
12. 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.

13. E. Felten and M. Schneider. Timing Attacks on Web Privacy. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 25–32. ACM, 2000.
14. 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.
15. S. Ford, M. Cova, C. Kruegel, and G. Vigna. Wepawet, 2009.
16. S. Frei, T. Duebendorfer, G. Ollmann, and M. May. Understanding the Web browser threat. Technical Report 288, TIK, ETH Zurich, June 2008. Presented at DefCon 16, Aug 2008, Las Vegas, USA. `http://www.techzoom.net/insecurity-iceberg`.
17. J. Garrett et al. Ajax: A new approach to web applications. *Adaptive path*, 18, 2005.
18. O. Hallaraker and G. Vigna. Detecting Malicious Javascript Code in Mozilla. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 85–94. Ieee, 2005.
19. A. Hintz. Fingerprinting websites using traffic analysis. In *Proceedings of the 2nd international conference on Privacy enhancing technologies*, pages 171–178. Springer-Verlag, 2002.
20. M. Huber, M. Mulazzani, and E. Weippl. Tor http usage and information leakage. In *Communications and Multimedia Security*, pages 245–255. Springer, 2010.
21. 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`.
22. C. Jackson, A. Bortz, D. Boneh, and J. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international conference on World Wide Web*, pages 737–744. ACM, 2006.
23. 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.
24. S. Kamkar. evercookie–never forget. *New York Times*, 2010.
25. F. R. T. Kohno and D. Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the Symposium on Networked System Design and Implementation (NSDI)*. USENIX Association, 2012.
26. T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, 2005.
27. M. Lindorfer, C. Kolbitsch, and P. Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection (RAID)*, 2011.
28. G. Lyon and Fyodor. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning.* Insecure, 2009.
29. D. McCoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker. Shining light in dark places: Understanding the tor network. In *Privacy Enhancing Technologies*, pages 63–76. Springer, 2008.
30. P. Mell and T. Grance. The NIST Definition of Cloud Computing. *NIST special publication*, 800:145, 2011.
31. 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.
32. S. Murdoch and G. Danezis. Low-cost traffic analysis of tor. In *Security and Privacy, 2005 IEEE Symposium on*, pages 183–195. IEEE, 2005.

33. A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*, pages 103–114. ACM, 2011.
34. A. Pfitzmann and M. Köhntopp. Anonymity, unobservability, and pseudonymity—a proposal for terminology. In *Designing privacy enhancing technologies*, pages 1–9. Springer, 2001.
35. S. Pichai and L. Upson. Introducing the google chrome os. *The Official Google Blog*, 2009. Online at `http://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html`.
36. M. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt. Trends in circumventing web-malware detection.
37. P. Ratanaworabhan, B. Livshits, and B. 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.
38. D. Richardson, S. Gribble, and T. Kohno. The limits of automatic os fingerprint generation. In *Proceedings of the 3rd ACM workshop on Artificial intelligence and security*, pages 24–34. ACM, 2010.
39. K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 31–39. ACM, 2010.
40. M. Smart, G. Malan, and F. Jahanian. Defeating tcp/ip stack fingerprinting. In *Proceedings of the 9th USENIX Security Symposium*, volume 24, 2000.
41. A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. Hoofnagle. Flash Cookies and Privacy. *SSRN preprint (August 2009) http://papers. ssrn. com/sol3/papers. cfm*, 2009.
42. w3school. Browser statistics, 2012. Online at `http://www.w3schools.com/browsers/browsers_stats.asp`.
43. G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 171–180. IEEE, 2008.
44. Z. Weinberg, E. Chen, P. 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.
45. 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.
46. 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.
47. M. Zalewski. p0f: Passive os fingerprinting tool, 2006. Online at `http://lcamtuf.coredump.cx/p0f3`.