Fingerprinting Information in JavaScript Implementations

Keaton Mowery[†], Dillon Bogenreif[*], Scott Yilek[*], and Hovav Shacham[†]

[†]Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California, USA

[*]Department of Computer and Information Sciences
University of St. Thomas
Saint Paul, Minnesota, USA

ABSTRACT

To date, many attempts have been made to fingerprint users on the web. These fingerprints allow browsing sessions to be linked together and possibly even tied to a user's identity. They can be used constructively by sites to supplement traditional means of user authentication such as passwords; and they can be used destructively to counter attempts to stay anonymous online.

In this paper, we identify two new avenues for browser fingerprinting. The new fingerprints arise from the browser's JavaScript execution characteristics, making them difficult to simulate or mitigate in practice. The first uses the innate performance signature of each browser's JavaScript engine, allowing the detection of browser version, operating system and microarchitecture, even when traditional forms of system identification (such as the user-agent header) are modified or hidden. The second subverts the whitelist mechanism of the popular NoScript Firefox extension, which selectively enables web pages' scripting privileges to increase privacy by allowing a site to determine if particular domains exist in a user's NoScript whitelist.

We have experimentally verified the effectiveness of our system fingerprinting technique using a 1,015-person study on Amazon's Mechanical Turk platform.

1. INTRODUCTION

A unique fingerprint that identifies a browser and its user has many uses on the Web. Used constructively, such a fingerprint allows sites to recognize returning users both between visits and during a visit, for example to avoid asking for login credentials for every action that requires authentication, or to detect account hijacking and impersonation. Used maliciously, such a fingerprint may allow an attacker to track a user between sites (even when those sites do not cooperate with the attacker) and to identify users who use privacy-enhancing technologies such as Tor.

The traditional means for a user to identify himself to a service is the password. With services and user browsing data moving to cloud-hosted infrastructure, it is increasingly the case that large-scale data compromise makes it hard to have a firm hold on customer identity. Since users reuse passwords at multiple sites, this is true even when a site impeccably secures its own login database: The recent Gawker compromise allowed hackers to compromise many Twitter accounts that reused the same login credentials.

Accordingly, for many sites, and especially those (such as online banking) where fraud is a concern, identifying users by password alone is not sufficient. Increasingly, such sites employ additional means to identify the user and the system she is using. Hardware tokens are appropriate in certain settings, but their cost and inconvenience form a barrier to ubiquitous use. For such sites, browser and user fingerprints form an attractive alternative to hardware tokens. On the (reasonable) assumption that typical users will usually log in from a single machine (or, at most, a handful), JavaScript fingerprinting allows sites to harden passwords against account hijacking with a minimum of user inconvenience.

Destructively, fingerprints can also be used to identify and track users, even when those users wish to avoid being tracked. The most familiar browser fingerprinting technology is the cookie, a client-side datastore partitioned by domain and path. Cookies installed by third-party content included on sites users visit allow users to be tracked between sites. Modern browsers provide many other client-side datastores that can provide cookie-like functionality [16]. The browser history and file cache are effectively unpartitioned by domain and can be queried by timing or DOM APIs [12], and can be used as a fingerprint [15]. Information disclosed by the browser in headers, through JavaScript APIs, and through plugins can provide a fingerprint with more than 18 bits of effective entropy [18, 9]. Even quirks of the browser's JavaScript handling and supported DOM features can be used as a fingerprint [11]. These techniques can be used in concert by sites, yielding a fingerprint that can uniquely identify users and facilitate their being tracked.

Our results.

We describe two new techniques for fingerprinting browsers, both making use of the JavaScript interpreter. The fingerprinting information we obtain can be used alongside previously known fingerprinting technologies, such as those explored in EFF's Panopticlick project, to enable user device identification and supplement user passwords (in a constructive appplication) or to reduce the anonymity set of Web users even further (in a destructive application).

Our first technique times the performance of different operations in the core of the JavaScript language. Unlike previous fingerprinting techniques based on JavaScript [11, 9], ours depends not on functionality differences, but performance differences between browsers. This means that it can be used even when limited functionality is made available to JavaScript. Also, because our technique takes advantage of the execution timing differences between instruction sequences produced by the browser's JavaScript JIT, it can distinguish not only browser versions but also microarchitectural features not normally exposed to JavaScript. As browsers compete largely on JavaScript performance, any countermeasure that involves slowing JavaScript execution or preventing accurate timing will be unacceptable to vendors, making this fingerprinting technique very robust.

Our second technique allows the attacker to query entries in the user's NoScript whitelist. NoScript [17] is a Firefox extension that allows users to whitelist or blacklist domains for JavaScript execution. When a user visits a page, scripts will execute only if the site's domain is whitelisted. Moreover, if page includes a script from a second domain, then that script will execute only when that script-provider domain is also whitelisted. Many sites do not function properly when JavaScript is disabled, so users customize their NoScript whitelist for their favorite sites. We show that a site can include a script from another domain to determine whether that other domain is whitelisted; scripts appropriate for including can be automatically found for a large fraction of the Alexa Top 1000 sites. When tested in aggregate, the domains found in the NoScript whitelist constitute a fingerprint of a user's preferred sites and habits, leaking valuable information about their preferences to attackers. While this attack targets a particular browser extension (which may not be installed by every fingerprinted user), its existence illustrates the possiblity of arbitrary websites probing for the existence and internal state of browser extensions.

Applications and threat model.

For constructive applications, we consider an attacker who has stolen a user's login credentials (via malware or phishing) and is attempting to impersonate the user. Such an attacker can study the victim's system configuration and could mimic the user-agent string, but may not be able to duplicate the victim's hardware and software configuration. Under this model, the attacker may impersonate the user in any self-reported fingerprint scheme (such as the Panopticlick approach of accessing user agent, available plugins, and other data through JavaScript APIs). When creating a fingerprint directly from raw JavaScript performance, however, the attacker will be unable to replicate the user's timings for every conceivable script. This unforgeability property provides an inherent advantage over other browser fingerprinting schemes.

For destructive applications, we consider browsers' "private browsing" modes [1] or the Tor network [8] together with the Torbutton extension for Firefox [20]. For the former, we adopt the Web attacker model of Aggarwal et al. [1]; for the latter, we also consider a rogue exit node model; unlike a Web attacker, a rogue Tor exit node can inject content that appears to come from sites the attacker doesn't control — including sites on the default NoScript whitelist such as `addons.mozilla.org`.

Related work.

When operations depend on secret values, the time it takes to complete them often leaks information useful to the attacker. Timing side channels have been used to attack several cryptosystems, notably a remote timing attack on RSA decryption in SSL [5]. In a Web context, timing has been used against clients to determine whether content was in the cache [10] and against servers to learn information such as the number of items in a user's shopping cart [4]. Clock skew, which is correlated with processor activity, has been used to identify Tor hidden services [19]. Finally, we observe that other sources of information, including summaries of the Web traffic they generate, can be used to fingerprint browsers [22].

Our NoScript whitelist fingerprint is closely related to history sniffing [12, 13]. The different decoration applied by browsers to visited and unvisited links, together with ways for sites to determine the decoration applied to page elements, makes it possible for attackers to determine whether a users visiting their site had previously visited any URL of interest. This technique, known since at least 2002 [7], can be used interrogate tens of thousands of URLs per second [14]. Whereas history sniffing applies to specific URLs, NoScript whitelist sniffing applies to domains; whereas history entries are set automatically whenever a user visits a page, NoScript whitelists are changed manually by users. These two differences mean that NoScript whitelist sniffing provides less information than history sniffing; the latter difference may mean that the fingerprint it provides is more stable over time. Moreover, with browsers deploying fixes against traditional history-sniffing techniques (see, e.g., [6, 2] for Firefox), the amount of information available to attackers from history sniffing may be reduced, making other techniques more attractive by comparison.

2. JAVASCRIPT PERFORMANCE FINGERPRINTING

As browsers advance and compete on features and usability, one of the largest areas of development is the JavaScript engine. Benchmark suites, such as V8 and SunSpider, are created to measure scripting speed. As developers apply clever approaches such as just-in-time script compilation, the JavaScript engines improve. These improvements are not uniform, however: different approaches yield different payoffs on certain types of script. The process of incremental improvement produces a tell-tale signature, detectable simply through timing the execution of otherwise innocuous JavaScript. By leveraging these discrepancies, timing information can be used to fingerprint the host machine.

2.1 Methodology

Utilizing the SunSpider and V8 JavaScript benchmarks along with custom code, we constructed a suite of 39 individual JavaScript tests. To create a fingerprint, we simply measure the time in milliseconds to complete each test in our benchmark suite. This produces a 39-dimensional vector, which characterizes the performance of the tested machine.

Naturally, differences in processor architecture and clock speed impact these timings significantly. Many processors also support CPU throttling, which dynamically slows the processor during periods of low utilization. Similarly, benchmark completion times may be affected if external (non-browser) processes cause significant load. To remove these effects from our measurements, we normalize the timing vector before attempting classification.

Oddly, single-test times, even during a single otherwise-idle browser session, can vary widely. This is possibly due to other scripts running in the browser, JavaScript garbage collection, or even the timing infrastructure used by our benchmarks. We take several steps to minimize the effects of this variance. First, we add an 800 ms delay between the end of one test and the start of another. This allows the browser time to handle any cleanup and execute scripts running in other windows of the session. Secondly, we run each test five times, and take the minimum positive time for each test. Intuitively, while the browser may be slowed by external factors, it will never be induced to perform better than its maximum speed. Also, our JavaScript timing framework reported that an indivdual test time took 0 ms an appreciable fraction of the time, even dipping to −1 ms once or twice (which is clearly impossible). Running each test multiple times, then, will smooth out these glitches and random browser slowdowns, in effect decreasing the variance of the final test vector. While these techniques increase the reliability of our tests, they do impose a penalty in terms of execution time.

Our benchmark suite takes 190.8 s to complete on Firefox 3.6. However, due to the per-test 800 ms timeout, the test suite spends approximately 156 s sleeping. By using a smaller timeout, the total time could be reduced significantly. Also, we did not experiment with removing tests — our results may be achievable with less than 39 tests.

(More generally, while we have used off-the-shelf JavaScript benchmark scripts to demonstrate the feasibility of our approach, we believe that custom-written scripts targeting specific JavaScript engine revisions and microarchitectural features would execute more efficiently and provide an even more effective fingerprint than our prototype.)

Once we have a fingerprint vector for a given configuration, we need some way of classifying a test vector in order to infer facts about the user's setup. To do so, we utilize an extremely simple method: matching the fingerprint vector to a representative vector corresponding to a known configuration. From our data, we generate fingerprint vectors of several leading browsers, and use these to classify the browser used to execute the benchmark code.

2.1.1 *Optimization*

One of the largest weaknesses in our approach is that the fingerprinting time is very large — usually over 3 minutes. Much of this time is spent on ways to reduce timing jitter: an 800 ms pause is inserted between each test, and the entire benchmark suite is executed five separate times. As noted recently by the Google Chrome team [3], modern browsers complete most SunSpider benchmarks extremely quickly (under 10 ms), and so a random delay of just a few milliseconds can appear as a momentous difference in test times. By running each individiual test many times in a row (Google chose 50 repititions for their SunSpider variant), timing variance can be greatly reduced. As a bonus, letting each test run longer will mitigate the JavaScript timing idiosyncrasies that result in negative or zero elapsed times. Taken together, these facts suggest that timing many runs of each individual benchmark test (instead of timing each test many times) will reduce timing variance significantly. Heavy-handed measures such as long inter-test delays and gratuitious repetition can be discarded, thereby eliminating our largest sources of delay and shortening our fingerprinting time considerably.

2.2 Data Collection

We collected JavaScript fingerprints for 1015 user configurations, where each configuration is an operating system, browser, hardware, and benchmark performance tuple. To achieve data collection on this scale, we conducted a survey on the Amazon Mechanical Turk marketplace, paying users a small monetary sum to report their CPU model, clock speed, RAM, and operating system and to allow our JavaScript to run. We also record the browser-reported User Agent string, which we treat as ground truth (i.e., we assume that user agents are not forged).

The operating systems reported by our users include variants of Windows (7, Vista, XP, NT 4.0), OS X (10.6, 10.5, 10.4) and Linux (Ubuntu, Mint, generic). As for browsers, we observed usage of Chrome, Firefox, Internet Explorer, Opera, Safari, and SeaMonkey. The exact breakdowns are given in Table 1.

Intel processors underly a vast majority of our participants' systems, with the most popular being Core 2 (395), Pentium Dual Core (137), and Pentium 4 (123). Other common platforms include Core i3, Core, Atom, Athlon 64, Core i5, and Athlon II.

Overall, our results appear to contain a representative sample of the browsers, operating systems and hardware configurations in use on the Web today.

2.2.1 *Data Quality*

Our Mechanical Turk survey presented users with three free-form text entry boxes, along with drop-down menus to report operating system and number of CPU cores. We also provide step-by-step instructions, with example screenshots, detailing how to obtain the relevant information for both Windows and OS X.

Nevertheless, improperly filled-out surveys comprised a non-trivial portion of responses. Each of our 1015 samples was hand-verified, and we include every response for which we could decipher a distinct CPU, clock speed, core count, and RAM. We received an extra 185 submissions which did not measure up to this standard and were excluded from further consideration. A fairly reasonable (but unhelpful) answer was "Celeron", as this does not specify a particular CPU model. We accepted their response (and paid the user), then excluded the result later in the process. Most of the rejections, however, were unusable: "Intel" and "x86" were very common. Some of the authors' favorite submissions for CPU architecture included "Normal", "HP Compaq LE1711" (an HP LCD monitor), and two separate participants who reported their architecture as "von Neumann".

Furthermore, among our 1015 accepted data points, 47 of them self-report a different operating system than their HTTP user agent claims. We did not investigate further, but note that this does suggest that some valid-looking but incorrect data might have been submitted by misunderstanding users. However, we ignore these effects and treat every report as correct.

2.3 Results

2.3.1 *Browser Detection*

First, we demonstrate the feasibility of determining browser version through JavaScript performance. To do so, we ran our test suite on a single computer in a variety of browsers. The test machine consisted of an Intel Core 2 Duo processor at 2.66 GHz with 4 GB of RAM. We created test vectors for Firefox 3.6.3, Internet Explorer 8.0, Opera 10.51, and Safari 4.0.5. Internet Explorer was tested on Windows 7, while the remaining three browsers were run on Mac OS X 10.6.3. Each browser returns a 39-dimensional normalized vector of test times, a subset of which are presented in Figure 1. Clearly, significant differences exist in the JavaScript performance profile of each browser, which we can exploit to recognize a browser's unique JavaScript execution signature.

To determine the effectiveness of general browser detection via JavaScript performance, we use the survey-supplied labels to generate browser fingerprint vectors. For each browser under test, we average the first ten survey vectors we received, using 38 tests as vector elements (test generation and selection is discussed in Section 2.4). The resulting vector is the first-level browser fingerprint.

While the top-level fingerprints are generally reliable, in our experience it creates unnecessary misclassification errors between minor versions of Firefox (namely, 3.6 and 4.0). To combat this, we also create a second-level fingerprint, consisting of 37 tests, specifically for finer-grained Firefox classification. Since this fingerprint is only applied if the first-level classification indicates a browser in the Firefox family, we can strip out unneeded tests and produce fingerprint vectors to distinguish between minor variations in the SpiderMonkey JavaScript engine. This pattern of successively more precise tests can produce far better results than a single-level classification alone.

Our data set provides enough benchmarks to create fingerprint vectors for various major versions of Chrome (2.0 through 11.0) and Internet Explorer (7, 8, 9), along with minor versions for Firefox (2.0, 3.0, 3.1, 3.5, 3.6, 4.0b) and Safari (4.0, 4.1, 5.0). We also have both an Opera 9.64 and a SeaMonkey 2.0 sample. Our methods are sufficiently robust to reliably detect the distinct JavaScript execution profile for each of these 23 browser versions. Further work might be able to distinguish between even smaller browser revisions, if those revisions included changes to the JavaScript engine.

To classify an unknown configuration, we simply execute the test suite which creates the 39-element fingerprint vector. Classification then becomes a simple matter of comparing against browser fingerprints and choosing the nearest match, using the Euclidean distance metric. The results of classification for our 1015 samples can be found in Table 2. Notably, we correctly classify 810 browsers, for an overall accuracy of 79.8%.

Our classification results include the reference vectors for each browser, i.e., the first ten samples from which we create the browser fingerprint vector. We use this approach mainly due to the relative excess of small categories in our data set: 15 browser versions contain less than 10 data points. For these browsers, we generate the test vector using every available sample. To demonstrate browser differentiation, then, we have no other samples with which to perform proper classification, so we classify the reference vectors themselves and see if they match their browser vector or not (note that an IE 7.0 sample is misclassified, even though the reference vector is made of itself and one other vector!). By keeping these categories, we also gain the benefit that vectors in the larger browser pools are compared against the smaller versions, allowing for misclassifications if the browsers' performance characteristics are too similar. For consistency, we include the reference vectors of the larger categories in our classification results as well.

Examining our results in detail, we notice that 176 of 205 misclassifications occur between major versions of Chrome, giving us a Chrome detection correctness of 62.5%. We ascribe these failures to two major features. First, Chrome's auto-updater continuously moves most users along the upgrade path. We were able to acquire less than 5 fingerprint vectors for Chrome 2.0 through 7.0, which reduces our confidence in the overall major version fingerprint. Secondly, Chrome's extremely aggressive release schedule means that only about 6 months elapsed between the release of Chrome 6.0 and Chrome 10.0. Our experiments indicate that JavaScript development continued over those six months, but not enough to reliably distinguish between versions using our test suite. This pattern of incremental improvement can be seen in Table 3, which displays the pairwise distance between Chrome fingerprint vectors.

	6.0	7.0	8.0	9.0	10.0	11.0
Chrome 6.0	0.00	0.18	0.19	0.17	0.25	0.25
Chrome 7.0	0.18	0.00	0.09	0.16	0.25	0.24
Chrome 8.0	0.19	0.09	0.00	0.17	0.27	0.26
Chrome 9.0	0.17	0.16	0.17	0.00	0.17	0.18
Chrome 10.0	0.25	0.25	0.27	0.17	0.00	0.09
Chrome 11.0	0.25	0.24	0.26	0.18	0.09	0.00

Table 3: Pairwise distances of Chrome major version fingerprints

Overall, our methodology is extremely robust for browser family detection (Chrome, Firefox, etc), with a correctness rate of 98.2%.

2.3.2 Operating System Detection

In the previous subsection, we described techniques for fingerprinting browser versions via JavaScript performance. In this subsection, we extend our techniques to fingerprint operating system versions. This is more difficult to do, but also provides information that is more difficult to gather reliably from adversarial users by means of the User Agent string or other forms of browser self-reporting. Indeed, several alternative methods exist for detecting browser version, such as inspection of browser-specific JavaScript capabilities, whereas scripts must rely on browser self-reporting for such platform details as operating system version or underlying architecture.

The effects of operating system on JavaScript performance are quite small. In fact, they are vastly overshadowed by differences in the JavaScript engines themselves. To combat this, operating system detection is only feasible within a particular browser version. We chose to examine the effects of operating system on Firefox 3.6, as it was reliably detectable and had the most cross-platform data points in our survey (397 on Windows, 19 on OS X, and 8 on Linux). We followed the same procedure as in browser detection: using the same 38 tests from the first ten samples in each category to form the fingerprint, then using Euclidean distance for classification. The results of this classification are in Table 4.

Notably, while we lack a significant number of Linux and OS X samples, our Windows identification rate is 98.5%, and we did not misclassify a single OS X example, demonstrating that operating system detection is quite possible through JavaScript benchmarking. Also, given the even distribution across successive Windows versions, these results indicate that detecting versions within an operating system family through JavaScript is difficult. Further targeted work is needed in this area.

	Windows 7	Windows Vista	Windows XP	NT 4.0	OS X 10.6	OS X 10.5	OS X 10.4	Linux
Chrome 2.0	1	-	-	-	-	-	-	-
Chrome 3.0	1	-	2	-	-	-	-	-
Chrome 4.0	1	-	-	-	-	-	-	-
Chrome 5.0	2	1	1	-	-	-	-	-
Chrome 6.0	-	-	2	-	-	-	-	1
Chrome 7.0	-	-	2	1	-	-	-	1
Chrome 8.0	2	1	7	-	-	-	-	3
Chrome 9.0	98	25	87	-	9	-	-	3
Chrome 10.0	74	25	68	-	8	1	-	6
Chrome 11.0	6	-	5	-	2	-	-	-
Firefox 2.0	1	1	3	-	-	-	-	-
Firefox 3.0	1	1	12	-	1	1	-	-
Firefox 3.1	1	-	-	-	-	-	-	-
Firefox 3.5	6	2	13	-	-	-	-	1
Firefox 3.6	162	45	190	-	13	4	2	8
Firefox 4.0	3	-	-	-	-	-	-	-
IE 7.0	1	1	-	-	-	-	-	-
IE 8.0	34	6	10	-	-	-	-	-
IE 9.0	3	-	-	-	-	-	-	-
Opera 9.64	-	-	1	-	-	-	-	-
Safari 4.0	-	-	-	-	-	3	1	-
Safari 4.1	-	-	-	-	-	-	1	-
Safari 5.0	1	-	1	-	21	13	-	-
Seamonkey 2.0	-	-	1	-	-	-	-	-

Table 1: Fingerprints Collected for Classification: OS and Browser Breakdown

	Chrome										Firefox						IE			Opera	Safari			SM
	2	3	4	5	6	7	8	9	10	11	2	3	3.1	3.5	3.6	4	7	8	9	9.64	4	4.1	5	2
Chrome 2.0	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Chrome 3.0	-	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Chrome 4.0	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Chrome 5.0	-	-	-	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Chrome 6.0	-	-	-	-	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Chrome 7.0	-	-	-	-	-	3	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Chrome 8.0	-	-	-	-	-	1	5	7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Chrome 9.0	-	-	1	4	6	17	11	171	8	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Chrome 10.0	-	-	-	1	1	6	1	25	79	69	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Chrome 11.0	-	-	-	-	-	-	-	1	4	8	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Firefox 2.0	-	-	-	-	-	-	-	-	-	-	5	-	-	-	-	-	-	-	-	-	-	-	-	-
Firefox 3.0	-	-	-	-	-	-	-	-	-	-	-	16	-	-	-	-	-	-	-	-	-	-	-	-
Firefox 3.1	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-
Firefox 3.5	-	-	-	-	-	-	-	-	-	-	-	-	-	21	-	-	-	-	-	-	-	-	-	1
Firefox 3.6	-	1	-	-	-	-	-	-	-	-	-	3	-	7	403	6	-	-	-	-	-	-	1	3
Firefox 4.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3	-	-	-	-	-	-	-	-
IE 7.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	1	-	-	-	-	-	-
IE 8.0	-	-	-	-	-	-	-	-	-	-	-	3	-	-	-	-	-	45	-	-	-	2	-	-
IE 9.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3	-	-	-	-	-
Opera 9.64	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-
Safari 4.0	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3	-	-	-
Safari 4.1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-
Safari 5.0	-	-	-	-	-	-	-	-	-	-	-	4	-	-	-	-	-	-	-	2	-	2	28	-
Seamonkey 2.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1

Table 2: Browser Detection Results
Columns correspond to classification, rows represent actual version (as reported by user agent)

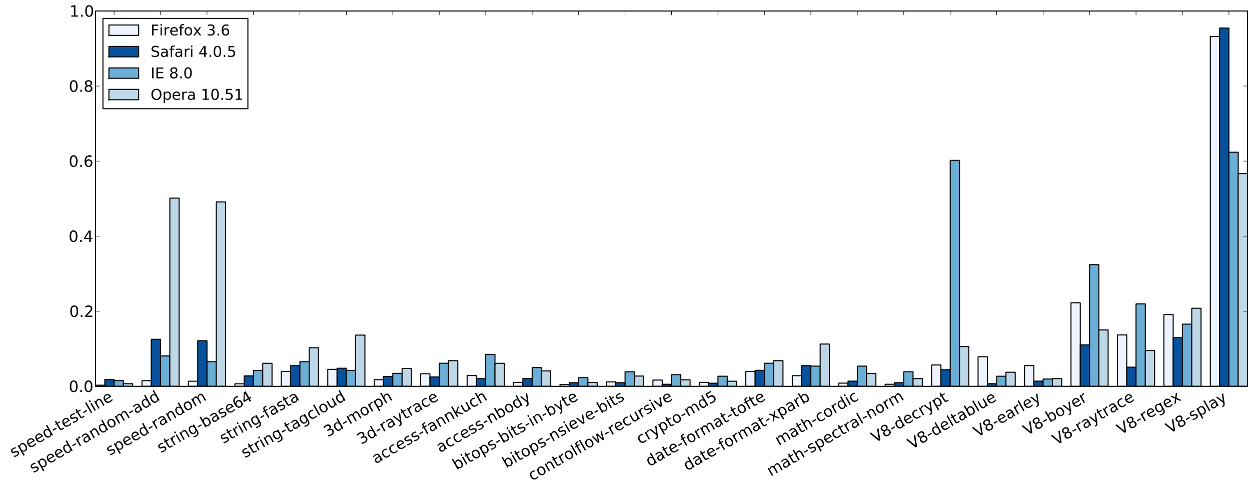

Figure 1: Comparison of relative browser performance: Normalized test time vectors for different browsers on a single machine

	Linux	OS X 10.4	OS X 10.5	OS X 10.6	Windows 7	Windows Vista	Windows XP
Linux	2	3	-	-	-	-	3
OS X 10.4	-	2	-	-	-	-	-
OS X 10.5	-	-	2	2	-	-	-
OS X 10.6	-	2	1	10	-	-	-
Windows 7	1	-	-	-	76	68	17
Windows Vista	-	-	-	-	17	23	5
Windows XP	1	1	3	3	51	80	51

Table 4: Detected Operating Systems under Firefox 3.6

2.3.3 *Architecture Detection*

Continuing our exploration of Firefox 3.6 behavior, we examine the effects of processor architecture on browser performance. In our data set, the major architectures running Firefox 3.6 are Core 2 (150), Pentium Dual Core (51), Pentium 4 (53), Core i3 (26), and Athlon 64 (22).

Our detection methodology for architecture differs slightly than our previous classification strategies. We use a 1-nearest neighbor strategy, based on the same 38-test benchmark vector used in first-level browser classification. This approach represents a departure from our previous classification strategies; we discuss why shortly. While 1-nearest neighbor gives reasonable results, it performs very badly on underrepresented platforms with very few samples. Therefore, we exclude any architecture with 5 samples or fewer from consideration.

The results from the classification can be found in Table 5. Overall, we achieve a 45.3% classification success rate.

Examining the results in detail, we see clusters of misclassifications across similar processors. For example, processors sold under the “Pentium Dual Core” name are detuned versions of Core or Core 2 chips, and over half (71 of 120) of our misclassifications occur between the Core 2 and Pentium Dual Core categories. Core i3 and Core i5 processors can be extremely similar, with Intel Clarkdale and Arrandale processors being sold under both those names. Our Pentium D samples are mainly misclassified as Pentium 4, which we posit is due both to the minimal number of Pentium D examples as well as the design similarity between those processors. Our survey procedures collected manufacturer marketing names, like “Core 2” and “Athlon”, while the data suggests that a more precise approach, such as determining individual manufacturer code names, could allow for better and finer-grained processor detection.

We chose to use a 1-nearest neighbor classification scheme due to this marketing name mismatch. For example, a test vector made from samples from both Yonah and Penryn-based Pentium Dual Core CPUs will average to somewhere between the two, leaving each individual Dual Core sample lying closer (and classified as) to the (Yonah) Core and (Penryn) Core 2 test vectors, respectively. The 1-nearest neighbor classification scheme avoids this averaging issue, at the expense of slightly higher noise in the results. With a larger and more precise data set, we posit that the test-vector-generation approach would also work for detecting CPU architectures.

Overall, our JavaScript fingerprinting techniques are able to infer underlying hardware details which were previously unexposed to JavaScript APIs. We believe that more targeted work in this area will allow the inference of processor types with much higher reliability and accuracy.

	Atom	Pentium				Core	Pentium	Core			Athlon			Phenom II	Sempron
		3	4	D	M		Dual Core	2	i3	i5	Classic	64	II		
Atom	9	-	-	-	-	-	2	1	-	-	-	-	-	-	-
Pentium 3	-	5	-	-	-	-	1	1	-	-	-	-	-	-	-
Pentium 4	-	-	40	3	-	-	3	5	-	-	1	1	-	-	-
Pentium D	1	-	5	-	-	-	-	-	-	-	-	-	-	-	-
Pentium M	-	-	-	-	-	4	4	2	-	-	-	-	-	-	-
Core	-	-	-	-	1	1	5	8	-	1	1	1	-	-	-
Pentium Dual Core	-	-	2	-	1	4	11	30	-	2	-	-	-	1	-
Core 2	2	1	9	-	1	8	41	78	3	3	3	-	1	-	-
Core i3	-	-	-	-	-	1	1	1	17	4	1	1	-	-	-
Core i5	-	-	-	-	-	2	-	2	7	-	1	1	-	-	-
Athlon	-	-	-	-	-	-	-	4	-	1	1	2	-	1	1
Athlon 64	-	-	-	-	-	1	1	1	2	1	2	10	-	1	3
Athlon II	1	-	-	-	-	1	-	2	1	-	-	-	-	1	1
Phenom II	-	-	-	-	-	1	-	-	-	-	-	-	1	3	2
Sempron	-	-	-	-	-	2	-	-	1	-	-	-	-	-	7

Table 5: Detected CPU Architecture under Firefox 3.6

Strangely, in our tests, the total time taken to complete our benchmarks did not scale linearly with clock speed, even given an identical browser, operating system, and processor architecture. Since we expected overall performance to be a fairly good indicator of processor clock speed, these discrepancies may point toward additional sources of system-specific information, which could further enhance the user's architectural fingerprint.

2.4 JavaScript Test Selection

In our fingerprints, we use all 26 tests from SunSpider 0.9 and 9 tests constructed from the V8 Benchmark Suite v5 along with 4 custom tests, for a total of 39 individual benchmarks. The custom tests excercise straight-line code, nested for loops, random number generation, and basic arithmetic. Our fingerprinting techniques, however, use 38 of these tests to perform classification. We chose to discard the "splay" test from V8 due to its excessive completion time (over 2000 ms; other tests might take tens or hundreds of milliseconds). As demonstrated in Figure 1, left unchecked, "V8-splay" dominates the normalized vector of test times for both Firefox and Safari, marginalizing any interesting effects we might observe through the other tests. Therefore, we discard the splay test when computing fingerprint vectors (and note that best normalization results are achieved when all test timings are of similar magnitudes).

As previously noted in Section 2.3.1, this particular set of 38 tests performs badly when differentiating between minor versions of Firefox. After experimentation, we discovered that better results were achieved after removing the V8 test "regex" from consideration as well, leaving a 37-element vector for differentiating minor Firefox versions.

3. NOSCRIPT WHITELIST FINGERPRINTING

NoScript is a Firefox browser plug-in aimed at protecting users against malicious JavaScript on the web. The recommended usage model is to whitelist trusted sites: the NoScript website suggests that JavaScript be blocked by default, and recommends that the user allow scripts "only by trusted web sites of your choice (e.g. your online bank)". As the web becomes more interactive, allowing JavaScript is becoming increasingly necesssary to preserve website functionality. The NoScript whitelist, then, may contain potentially sensitive information about a user's browsing habits, such as her preferred news sources, banks, webmail provider, and entertainment. Individual whitelist entries can thus reveal sensitive personal information. Just as importantly, the fact that users who visit different sites will allow JavaScript from different sites means that whitelists can be used as additional form of fingerprint.

The NoScript whitelist is very flexible, allowing entries to be either Full Addresses (`http://www.noscript.net`), Full Domains (`www.noscript.net`), or Base 2nd Level Domains (`noscript.net`). By default, NoScript will populate its whitelist with Base 2nd Level Domains, allowing JavaScript for a domain and all of its subdomains. With this in mind, in this paper we consider all subdomains to be in the same protection domain as their parent. However, note that should a user switch to using either Full Addresses or Full Domains in their whitelist, she will still whitelist addresses which correspond to scripts she wishes to run. The JavaScript allowed by these entries is therefore executable in her browser and can be detected by our fingerprinting methods.

Technically speaking, the NoScript plugin blocks not only the execution of untrusted scripts, but even the fetching of such scripts from the server. Within a trusted JavaScript execution context, an untrusted script simply does not exist. No exceptions are thrown; the contents of the untrusted script are simply missing. As we will show, by using these facts a malicious website, once allowed to execute any JavaScript at all, can probe the contents of the user's NoScript whitelist.

Lastly, this attack is only mountable if a NoScript user allows malicious JavaScript to execute. However, many sites fail to display content or work properly without the use of JavaScript. The current best framebusting defense [21] even hides page content by default, requiring JavaScript execution to display anything at all. The attacker could also lure user cooperation via the offer of a game or service, either of which could require JavaScript functionality.

3.1 Attack Methodology

We begin by describing how an attacker can determine whether a particular domain is in the user's NoScript whitelist, and how she can then combine multiple such checks to compute a whitelist fingerprint for the user. Performing these checks requires locating suitable *indicator* JavaScript scripts on the sites to query; we describe an effective automated spidering approach for finding such scripts. As noted above, we grant our attacker the capabilities provided by the Web attacker or rogue Tor exit node model, as appropriate.

3.1.1 Domain Checking

To create a page which checks for a given domain in a NoScript whitelist:

1. Find a URL in the domain containing JavaScript, suitable for inclusion in a `<script>` tag.
2. Inspect the JavaScript for the name of a defined object, such as a variable or function.
3. Create a page with two elements:
    - A `<script>` tag referencing the script URL.
    - JavaScript code checking for the existence of the defined object.

When a user visits this page with JavaScript enabled, one of two things occurs: Either the object exists or is undefined. In the first case, JavaScript is enabled for the tested domain. Otherwise, the domain is blocked (via NoScript or some other means). Note that other browser plugins, such as AdBlock Plus, may block the script's execution. We ignore the effects of such plugins, as they effectively modify the NoScript blacklist.

Many scripts, when removed from their native locations, throw errors or otherwise fail during this test. Generally, this is due to missing components in the execution environment, such as other JavaScript elements from their domain or an unexpected DOM tree. If such an error occurs, execution of the script stops immediately. However, the functions and variables it defines are still inserted into the global `window` object, and thus our domain test will be successful.

3.1.2 Site Spidering

Any effective fingerprinting solution needs to encompass a significant number of possible domains. Manual creation of a domain-checking page is fairly trivial, but the creation of hundreds or thousands of these pages represents significant human effort. Fortunately, automation of this process is fairly easy.

To produce our domain-checking pages, we built a simple web spider. Given a domain, it simply crawls the first ten in-domain URLs, looking for `<script>` tags. Any scripts included from domains other than the one we are targeting are ignored, as presumably these are not necessary to the operation of the site in question. When an appropriate script URL is found, we execute the code using the V8 JavaScript engine[1] with an EnvJS[2] environment. If this execution finds a properly defined object, such as a variable or function, the spider has all it needs to create a NoScript test page for the domain. For an example of an automatically-created domain test page for `google.com`, see Figure 2.

[1]Online: `http://code.google.com/p/v8/`

[2]Online: `http://www.envjs.com/`

```
<html>
  <head>
    <script type="text/javascript"
      src="http://www.google.com/accounts/hosted
       /helpcenter/js/tooltips/TooltipLoader.js">
    </script>
    <script type="text/javascript">
      if ("XML_STATUS_OKAY" in window) {
        location.hash = "enabled";
      } else {
        location.hash = "disabled";
      }
    </script>
  </head><body></body>
</html>
```

Figure 2: Whitelist test page for `google.com`

3.1.3 Whitelist Fingerprinting

Once a suitable number of domain test pages are created, we can turn our attention to delivering the tests to the machine being fingerprinted and collecting the results into some usable format. To achieve this, we create a test suite from the individual test pages.

The test suite consists of an HTML page and associated JavaScript. To test a given domain, an iframe, containing the test page for the domain to test, is created and inserted into the page. Once the test page reaches a determination, it modifies its `location.hash` attribute. As each test completes, its iframe is destroyed, releasing its resources and, importantly, stopping all JavaScript execution in the frame. Since each iframe potentially includes arbitrary JavaScript, ceasing its execution as quickly as possible reduces browser load and allows for faster testing.

To prevent serious performance degradation, the test suite limits the number of active tests at once. This prevents a slow-to-respond site from blocking the fingerprinting progress, while not noticably reducing browser responsiveness. Currently, we simply set this limit to a small constant. Conceivably, this could be automatically adjusted for network lag and browser speed to reduce testing times even further.

This approach does have its limitations. Most notably, the testing process creates continual change in the browser's chrome. As each iframe loads from the test server, the text on the tab associated with the test flashes between its HTML page title and "Loading...". Also, test progress is noted in the status bar, with messages such as "Connecting to www.google.com" and "Transferring data from www.google.com". When NoScript blocks a script, it adds a yellow information bar to the bottom of the browser window. However, once the last iframe is removed, this NoScript information bar disappears as well. These limitations suggest that the optimal way to run the fingerprinting process is in a popunder or otherwise hidden tab, where these notifications will go unnoticed by an unsuspecting user.

Note that the attacker's site must itself be authorized to execute JavaScript for the fingerprinting to succeed. A Web attacker may be able to trick the user into temporarily adding her site to the whitelist, for example through the promise of a JavaScript-enabled game or other interesting content. A rogue Tor exit node will simply inject the script into a trusted-by-default domain.[3]

[3]An earlier version of the Torbutton FAQ recommended

3.2 Prevalence of Testable JavaScript

To measure the effectiveness of our techniques, we attempt to create whitelist probes for each of the Alexa Top 1,000 sites. Starting the spider at the root of each site, we investigate 10 pages in a breadth-first fashion. As described in Section 3.1.2, we stop when we find an appropriate script, which we define as any includable script available in the domain or any subdomain thereof.

Out of the Alexa Top 1,000 sites, we find 706 sites which fit our criteria, and generate NoScript whitelist test scripts for each of these domains.

There are several reasons why a site might not be testable:

1. No JavaScript
2. JavaScript only embedded in HTML pages (no `.js` files)
3. JavaScript files not accessible within first 10 pages
4. All JavaScript served from a different domain
5. Crawling forbidden by `robots.txt`

(Note that real attackers would not be constrained by the `robots.txt` file and could crawl many more pages in each site, looking for smaller scripts that would load and execute more quickly. Our findings are thus a lower bound on the effectiveness of our fingerprinting technique.)

For example, `yahoo.com` serves all of its external JavaScript files from `yimg.com`, while `facebook.com` disallows crawling, requires a login to get past the first few pages, and uses `fbcdn.net` to serve their JavaScript. Manual intervention may be required in these cases to recognize site-specific content distribution networks (CDNs), or high-value sites that request logins before presenting any interactive content (such as banks or webmail providers).

Once we acquired the set of test scripts, we had to manually remove 17 misbehaving tests. Most of these scripts contained elements such as `alert()` calls, framebusting code, attempts to set `document.domain`, or illegal characters. One of these scripts appears to be programmatically generated, and changes every few seconds (rendering our preinspection worthless). Another attempts to fetch a resource from our test server, and throws an error before creating the variable of interest. Lastly, our JavaScript execution engine simply fails on one script, leaving our test page useless. We did not inspect the affected sites for alternative scripts, although issues such as these could potentially be detected and discarded during the site crawl phase.

Therefore, our minimal crawler is able to generate usable test scripts for 68.9% of the Alexa Top 1000 sites. We note again that this is a lower bound, and additional (and less polite) searching would likely reveal suitable scripts at more sites.

3.3 Fingerprinting Speed

For user fingerprinting systems, speed is paramount. More elements tested means a more unique, and therefore more useful, fingerprint. To determine the speed of probing NoScript whitelists, we created a test suite (as described in Section 3.1.3) to examine all 689 checkable domains we found in the Alexa Top 1000. The test suite creates an iframe for each tested domain and loads a domain test page. This page attempts to fetch a script from the remote site and can then test for success. Once an answer has been determined, the domain's iframe is deleted, which stops any further script execution. To reduce the impact of high-latency servers, the test suite runs five independent iframes simultaneously.

We tested our benchmarks on Firefox 3.6.11 with NoScript 2.0.3.5. The test machine consisted of an Intel Core 2 Duo processor at 2.66 GHz with 4 GB of RAM. The test suite was served off of a machine on the UC San Diego network.

We ran the test suite twice, once with NoScript configured to block all domains and once with NoScript installed but configured to allow all scripts.

3.3.1 NoScript Disabled

With NoScript disabled, Firefox behaves normally, fetching all requested resources and executing all scripts. We execute the test five times. Each time, we clear all caches and restart Firefox to ensure that all necessary resources were fetched from their remote servers.

The mean time to completely test all 689 domains is 118.6 seconds, with a maximum of 131.9 seconds. Since our benchmark machine and test server are on the same network, fetching each of the domain test pages is a minimal cost. Most of the overall test time is spent fetching and executing the remote scripts. The CDF of domain test times (including iframe creation, test page fetch, remote script fetch, and script execution) across all five runs is shown in Figure 3.

3.3.2 NoScript Enabled

With NoScript enabled, Firefox refuses to run all scripts that don't appear on the NoScript whitelist. Helpfully, NoScript will disable even fetching a blocked script. This removes the network round trip associated with loading a domain's script (although not the original test page fetch). In accordance with our threat model, we whitelist our test suite, allowing JavaScript to run only for us.

We execute the test suite five times. Each time, we clear all caches and restart Firefox.

The mean time to completely test all 689 domains is 22.2 seconds, with a maximum of 23.3 seconds. Since no external requests are made, most of this time consists of iframe setup, test page fetch, and test suite delays. The CDF of domain test times across all five runs is shown in Figure 4.

Notably, as there are no external script fetches, this fingerprinting session runs much faster than before. Also, as none of the sites being tested receive any network traffic from the system being fingerprinted, they are unable to detect fingerprint attempts through log inspection.

3.3.3 In Practice

Our fingerprinting methodology expects NoScript to be installed and in use on a user's browser. As the user repeatedly visits sites of interest which require JavaScript, the user will likely whitelist those sites to streamline their browsing experience. When the user encounters a fingerprinting attempt, then, we expect that most domains are blocked while a select few are allowed. For the allowed domains, we suffer network fetch and script execution time penalties. In this model, then, the total time taken to fingerprint a particular user lies somewhere between the fully-blocked and fully-allowed cases.

against the use of NoScript because it would "allow malicious exit nodes to compromise your anonymity via the default whitelist (which they can spoof to inject any script they want)."

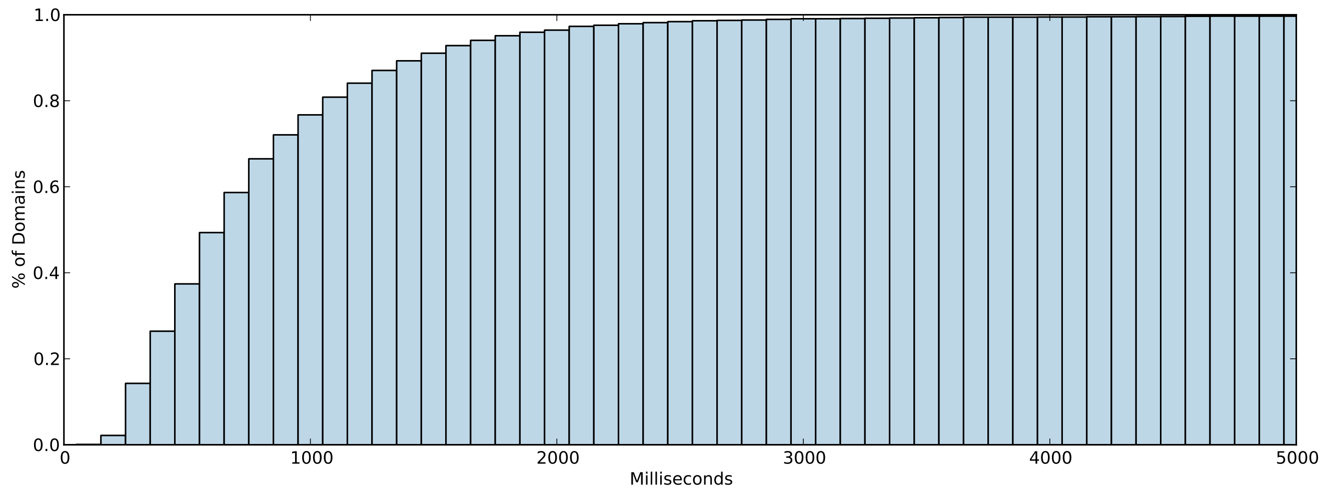

Figure 3: Completion Time CDF for domain whitelist check (NoScript Off)

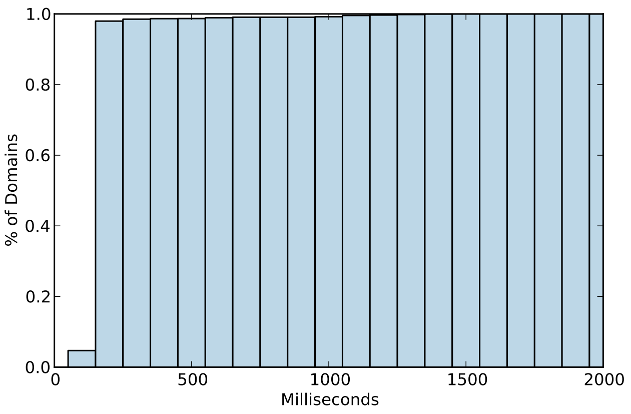

Figure 4: Completion Time CDF for domain whitelist check (NoScript On)

In this paper, we do not attempt to optimize the fingerprinting process. However, there are a few simple ways in which optimizations could be applied. For example, test pages could each test multiple domains, instead of just one. This would cut down on the number of iframes created, along with the number of elements fetched from the test suite server. Care must be taken to eliminate potential interactions between the unrelated scripts, but it is easy to imagine methods of making this possible. If we expect that each whitelist contains relatively few domains, then a less careful approach to testing will give almost as much speedup even when scripts might interact negatively with each other: test several domains in one iframe and, if any of them appear to be allowed, repeat the tests in separate iframes. Moreover, running more simultaneous iframes might allow more domains to be tested in parallel, at the cost of browser setup time. And, of course, the attacker can test that a user has enabled the NoScript extension before proceeding with the fingerprinting by checking that a script from some other attacker-controlled domain isn't loaded and run.

4. CONCLUSIONS

We have described two new ways to fingerprint browsers. The first is based on the relative performance of core JavaScript operations, the second based on probing entries in the domain sitelist of the NoScript Firefox extension. Our new fingerprints can be used by large-scale service providers to harden user accounts against hijacking, whether through phishing or other password compromise. They can be deployed alongside most other forms of browser fingerprint, including cookies and other client-side datastores, the contents of the cache and the history file, and browser and plugin functionality differences. We have developed proof-of-concept implementations of our fingerprinting techniques, showing that our JavaScript performance fingerprinting can distinguish between major browser versions, operating systems and microarchitectures, and that our NoScript whitelist fingerprint can be used to efficiently query almost 70% of the Alexa Top 1000 domains.

Our implementations represent a lower bound on the effectiveness of our techniques. We have shown that it is possible to distinguish between browsers, along with the underlying system hardware and software, based solely on scripting benchmarks. We believe that a finer-grained approach to JavaScript performance fingerprinting can provide even more detailed information, such as hardware revisions within a processor family, clock speed, cache size, and the amount of RAM on the target system. Secondly, extending our technique to mobile devices should produce excellent results, given their unique and constant combination of mobile browser, operating system, and mobile hardware. Also, we believe that NoScript whitelist fingerprinting can be deployed relatively efficiently and against a larger fraction of top sites than we have currently shown.

In future work we hope to deploy a measurement study, modeled after that of Eckersley [9], to measure the effective entropy from our fingerprints. We believe that there will be sufficient entropy in users' browsers, hardware configurations, and NoScript whitelists to usefully augment current fingerprinting techniques.

Acknowledgments

We thank Steve Checkoway, Eric Rescorla, Tom Ristenpart, and Stefan Savage for helpful discussions and the W2SP reviewers for their comments.

This material is based upon work supported by the National Science Foundation under Grants No. CNS-0831532 and CNS-0964702, and by the MURI program under AFOSR Grant No. FA9550-08-1-0352.

5. REFERENCES

[1] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In I. Goldberg, editor, *Proceedings of USENIX Security 2010*, pages 79–93. USENIX, Aug. 2010.

[2] L. D. Baron. Preventing attacks on a user's history through CSS :visited selectors, Apr. 2010. Online: `http://dbaron.org/mozilla/visited-privacy`.

[3] M. Belshe. Updating javascript benchmarks for modern browsers. Online: `http://blog.chromium.org/2011/05/updating-javascript-benchmarks-for.html`, May 2011.

[4] A. Bortz, D. Boneh, and P. Nandy. Exposing private information by timing Web applications. In P. Patel-Schneider and P. Shenoy, editors, *Proceedings of WWW 2007*, pages 621–28. ACM Press, May 2007.

[5] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–16, aug 2005.

[6] Bugzilla@Mozilla. Bug 147777 – :visited support allows queries into global history, May 2002. Online: `https://bugzilla.mozilla.org/show_bug.cgi?id=147777`.

[7] A. Clover. Timing attacks on Web privacy. Online: `http://www.securiteam.com/securityreviews/5GP020A6LG.html`, Feb. 2002.

[8] R. Dingledine and N. Mathewson. Tor: The second-generation onion router. In M. Blaze, editor, *Proceedings of USENIX Security 2004*, pages 303–19. USENIX, Aug. 2004.

[9] P. Eckersley. How unique is your Web browser? In M. Atallah and N. Hopper, editors, *Proceedings of PETS 2010*, volume 6205 of *LNCS*, pages 1–18. Springer-Verlag, July 2010.

[10] E. W. Felten and M. A. Schneider. Timing attacks on Web privacy. In S. Jajodia, editor, *Proceedings of CCS 2000*, pages 25–32. ACM Press, Nov. 2000.

[11] G. Fleischer. Attacking Tor at the application layer. Presentation at DEFCON 2009, Aug. 2009. Online: `http://pseudo-flaw.net/content/defcon/`.

[12] C. Jackson, A. Bortz, D. Boneh, and J. Mitchell. Protecting browser state from Web privacy attacks. In C. Goble and M. Dahlin, editors, *Proceedings of WWW 2006*, pages 737–44. ACM Press, May 2006.

[13] M. Jakobsson and S. Stamm. Invasive browser sniffing and countermeasures. In C. Goble and M. Dahlin, editors, *Proceedings of WWW 2006*, pages 523–32. ACM Press, May 2006.

[14] A. Janc and L. Olejnik. Feasibility and real-world implications of Web browser history detection. In C. Jackson, editor, *Proceedings of W2SP 2010*. IEEE Computer Society, May 2010.

[15] A. Juels, M. Jakobsson, and T. N. Jagatic. Cache cookies for browser authentication (extended abstract). In V. Paxson and B. Pfitzmann, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 2006*, pages 301–05. IEEE Computer Society, May 2006.

[16] S. Kamkar. evercookie – never forget, Sept. 2010. Online: `http://samy.pl/evercookie/`.

[17] G. Maone. NoScript. Online: `http://noscript.net/`.

[18] J. R. Mayer. "Any person... a pamphleteer": Internet anonymity in the age of Web 2.0. Senior thesis, Princeton University, Apr. 2009.

[19] S. Murdoch. Hot or not: Revealing hidden services by their clock skew. In R. Wright and S. De Capitani di Vimercati, editors, *Proceedings of CCS 2006*, pages 27–36. ACM Press, Oct. 2006.

[20] M. Perry. Torbutton design documentation, June 2010. Online: `http://www.torproject.org/torbutton/en/design/index.html.en`.

[21] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In C. Jackson, editor, *Proceedings of W2SP 2010*, May 2010.

[22] T.-F. Yen, X. Huang, F. Monrose, and M. Reiter. Browser fingerprinting from coarse traffic summaries: Techniques and implications. In U. Flegel, editor, *Proceedings of DIMVA 2009*, volume 5587 of *LNCS*, pages 157–75. Springer-Verlag, July 2009.