Browser Feature Usage on the Modern Web

Peter Snyder Lara Ansari Cynthia Taylor Chris Kanich

{psnyde2,lansar2,cynthiat,ckanich}@uic.edu
Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607

ABSTRACT

Modern web browsers are incredibly complex, with millions of lines of code and over one thousand JavaScript functions and properties available to website authors. This work investigates how these browser features are used on the modern, open web. We find that JavaScript features differ wildly in popularity, with over 50% of provided features never used on the web's 10,000 most popular sites according to Alexa.

We also look at how popular ad and tracking blockers change the features used by sites, and identify a set of approximately 10% of features that are disproportionately blocked (prevented from executing by these extensions at least 90% of the time they are used). We additionally find that in the presence of these blockers, over 83% of available features are executed on less than 1% of the most popular 10,000 websites.

We further measure other aspects of browser feature usage on the web, including how many features websites use, how the length of time a browser feature has been in the browser relates to its usage on the web, and how many security vulnerabilities have been associated with related browser features.

1. INTRODUCTION

The web is the world's largest open application platform. While initially developed for simple document delivery, it has grown to become the most popular way of delivering applications to users. Along with this growth in popularity has been a growth in complexity, as the web has picked up more capabilities over time.

This growth in complexity has been guided by both browser vendors and web standards. Many of these new web capabilities are provided through new JavaScript APIs (referred to in this paper as **features**). These capabilities are organized into collections of related features which are published as standards documents (in this paper, we refer to these collections of APIs as **standards**).

To maximize compatibility between websites and web browsers, browser vendors rarely remove features from browsers. Browser vendors aim to provide website authors with new features without breaking sites that rely on older browser features. The result is an ever growing set of features in the browser.

Many web browser features have been controversial and even actively opposed by privacy and free software activists for imposing significant costs on users, in the form of information leakage or loss of control. The *WebRTC* [9] standard has been criticized for revealing users' IP addresses [46], and protestors have literally taken to the streets [47] to oppose the *Encrypted Media Extensions* [15] standard. This standard aims to give content owners much more control over how their content is experienced within the browser. Such features could be used to prevent users from exerting control over their browsing experience.

Similarly, while some aspects of web complexity are understood (such as the number of resources web sites request [11]), other aspects of complexity are not, such as how much of the available functionality in the browser gets used, by which sites, how often, and for what purposes. Other related questions include whether recently introduced features are as popular as older features, whether popular websites use different features than less popular sites, and how the use of popular extensions, like those that block advertisements and online tracking, impact which browser features are used.

This paper answers these questions by examining the use of browser features on the web. We measure which browser features are frequently used by site authors, and which browser features are rarely used, by examining the JavaScript feature usage of the ten thousand most popular sites on the web. We find, for example, that 50% of the JavaScript provided features in the web browser are never used by the ten thousand most popular websites.

We additionally measure the browser feature use in the presence of popular ad and tracking blocking extensions, to determine how they effect browser feature use. We find that installing advertising and tracking blocking extensions not only reduces the amount of JavaScript users execute when browsing the web, but changes the kinds of features browsers execute. We identify a set of browser features (approximately 10%) that are used by websites, but which ad and tracking blockers prevent from executing more than 90% of the time. Similarly, we find that over 83% of features available in the browser are executed on less than 1% of websites in the presence of these popular extensions.

We have published data described in this work. This data includes the JavaScript feature usage of the Alexa 10k in

IMC '16, November 14–16, Santa Monica, CA, USA.

ISBN 978-1-4503-4526-2/16/11…$15.00

DOI: http://dx.doi.org/10.1145/2987443.2987466

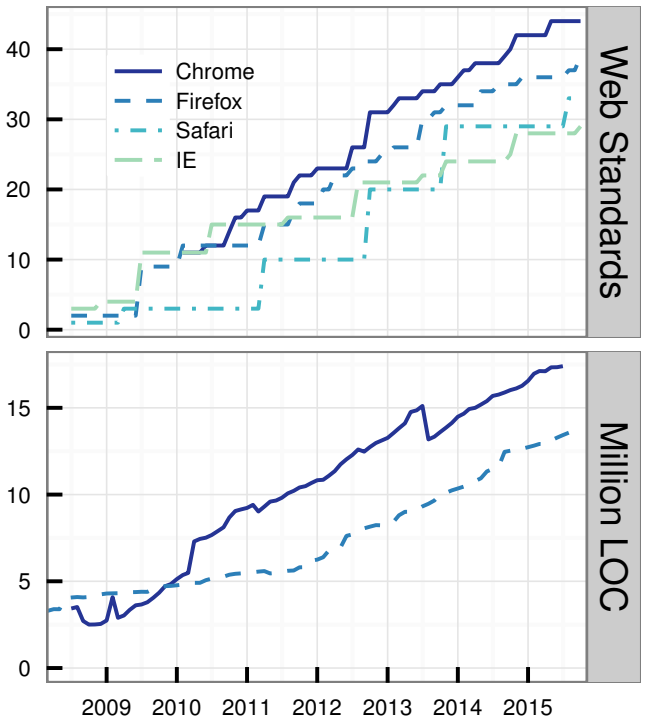

Figure 1: Feature families and lines of code in popular browsers over time.

both a default browser configuration and with ad and tracking blocking extensions in place, as well as the mappings of JavaScript features to standards documents. The database with these measurements, along with documentation describing the database's schema, is available online [49].

2. BACKGROUND

In this section, we discuss the complexity of the modern web browser, along with the use of ad and tracking blockers.

2.1 Modern Web Features

The functionality of modern web browsers has grown to encompass countless use cases. While the core functionality embodied by the combination of HTML, CSS, and JavaScript is largely stable, over the past few years many features have been added to enable for new use cases. Figure 1 shows the number of standards available in modern browsers, using data from W3C documents [61] and Can I Use [14]. Figure 1 also shows the total number of lines of code for Firefox and Chrome [10]. One relevant point of note in the figure is that in mid 2013, Google moved to the Blink rendering engine, which entailed removing at least 8.8 million lines of code from Chrome related to the formerly-used WebKit engine [34].

Vendors are very wary of removing features from the browser, even if they are used by a very small fraction of all websites [1, 2]. Because the web is evolving and competing with native applications, browser vendors are incentivized to continue adding new features to the web browser and not remove old features. This is exacerbated by browsers typically having a unified code base across different types of computers including mobile devices, browser-based computers such as Google Chromebooks, and traditional personal computers. Browser vendors then expose unique hardware capabilities like webcams, rotation sensors, vibration motors, or ambient light sensors [30, 31, 33, 55] directly through JavaScript, regardless of whether the executing device has such a capability. Furthermore, as new features are added, the current best practice is to roll them out directly to web developers as time limited experiments, and allow them to move directly from experimental features to standard features, available in all browsers that adhere to the HTML living standard. [48].

Individual websites are also quite complex. Butkiewicz et al. surveyed 2,000 random websites and found that loading the base page for a URL required fetching a median of 40 objects, and that 50% of websites fetched at least 6 JavaScript objects [11].

2.2 Ads and Tracking Blocking

Researchers have previously investigated how people use ad blockers. Pujol et al. measured AdBlock usage in the wild, discovering that while a significant fraction of web users use AdBlock, most users primarily use its ad blocking, and not its privacy preserving, features [44].

User tracking is a more insidious aspect of the modern web. Recent work by Radler found that users were much less aware of cross-website tracking than they were about collection of data by single sites such as Facebook and Google, and that users who were aware of it had greater concerns about unwanted access to private information than those who weren't aware [45]. Tracking users' web browsing activity across websites is largely unregulated, and a complex network of mechanisms and businesses have sprung up to provide services in this space [17]. Krishnamurthy and Willis found that aggregation of user-related data is both growing and becoming more concentrated, i.e. being conducted by a smaller number of companies [32].

Traditionally, tracking was done via client-side cookies, giving users a measure of control over how much they are tracked (i.e. they can always delete cookies). However, a wide variety of non-cookie tracking measures have been developed that take this control away from users, and these are what tracking blockers have been designed to prevent. These include browser fingerprinting [16], JavaScript fingerprinting [37, 40], Canvas fingerprinting [38], clock skew fingerprinting [29], history sniffing [27], cross origin timing attacks [56], evercookies [28], and Flash cookie respawning [7, 50]. A variety of these tracking behaviors have been observed in widespread use in the wild [3, 7, 36, 41, 42, 50, 51].

Especially relevant to our work is the use of JavaScript APIs for tracking. While some APIs, such as Beacon [20], are designed specifically for tracking, other APIs were designed to support various other functionality and co-opted into behaving as trackers [38, 60]. Balebako et al. evaluated tools which purport to prevent tracking and found that blocking add-ons were effective [8].

3. DATA SOURCES

This work draws on several existing sets of data. This section proceeds by detailing how we determined which websites are popular and how often they are visited, how we determined the current JavaScript-exposed feature set of web browsers, what web standards those features belong to and when they were introduced, how we determined the known vulnerabilities in the web browser (and which browser feature standard the vulnerability was associated with), and which

browser extensions we used as representative of common browser modifications.

3.1 Alexa Website Rankings

The Alexa rankings are a well known ordering of websites ranked by traffic. Typically, research which uses Alexa relies on their ranked list of the worldwide top one million sites. Alexa also provides other data about these sites. In addition to a global ranking of each of these sites, there are local rankings at country granularity, breakdowns of which subsites (by fully qualified domain name) are most popular, and a breakdown by page load and by unique visitor of how many monthly visitors each site gets.

We used the 10,000 top ranked sites from Alexa's list of the one-million most popular sites, and which collectively represent approximately one third of all web visits, as representative of the web in general.

3.2 Web API Features

We define a **feature** as a browser capability that is accessible through calling a JavaScript call or setting a property on a JavaScript object.

We determined the set of JavaScript-exposed features by reviewing the WebIDL definitions included in the Firefox version 46.0.1 source code. WebIDL is a language that defines the JavaScript features web browsers provided to web authors. In the case of Firefox, these WebIDL files are included in the browser source.

In the common case, Firefox's WebIDL files define a mapping between a JavaScript accessible method or property and the C++ code that implements the underlying functionality[1]. We examined each of the 757 WebIDL files in the Firefox and extracted 1,392 relevant methods and properties implemented in the browser.

3.3 Web API Standards

Web standards are documents defining functionality that web browser vendors should implement. They are generally written and formalized by organizations like the W3C, though occasionally standards organizations delegate responsibility for writing standards to third parties, such as the Khronos group who maintains the current *WebGL* standard.

Each standard contains one or more features, generally designed to be used together. For example, the *WebAudio API* [4] standard defines 52 JavaScript features that together allow page authors to do programmatic sound synthesis.

There are also web standards that cover non-JavaScript aspects of the browser (such as parsing rules, what tags and attributes can be used in HTML documents, etc.). This work focuses only on web standards that define JavaScript exposed functionality.

We identified 74 standards implemented in Firefox. We associated each of these to a standards document. We also found 65 API endpoints implemented in Firefox that are not found in any web standard document, which we associated with a catch-all *Non-Standard* categorization.

[1]In addition to mapping JavaScript to C++ methods and structures, WebIDL can also define JavaScript to JavaScript methods, as well as intermediate structures that are not exposed to the browser. In practice though, the primary role of WebIDL in Firefox is to define a mapping between JavaScript API endpoints and the underlying implementations, generally in C++.

In the case of extremely large standards, we identify substandards, which define a subset of related features intended to be used together. For example, we treat the subsections of the *HTML* standard that define the basic *Canvas API*, or the *WebSockets API*, as their own standards.

Because these sub-standards have their own coherent purpose, it is meaningful to discuss them independently of their parent standards. Furthermore, many have been implemented in browsers independent of the parent standard (i.e. browser vendors added support for "websockets" long before they implemented the full "HTML5" standard).

Some features appear in multiple web standards. For example, the `Node.prototype.insertBefore` feature appears in the *Document Object Model (DOM) Level 1 Specification* [6], *Document Object Model (DOM) Level 2 Core Specification* [23] and *Document Object Model (DOM) Level 3 Core Specification* [24] standards. In such cases, we attribute the feature to the earliest published standard.

3.4 Historical Firefox Builds

We determined when features were implemented in Firefox by examining the 186 versions of Firefox that have been released since 2004 and testing when each of the 1,392 features first appeared. We treat the release date of the earliest version of Firefox that a feature appears in as the feature's "implementation date".

Most standards do not have a single implementation date, since it could take months or years for all features in a standard to be implemented in Firefox. We therefore treat the introduction of a standard's currently most popular feature as the standard's implementation date. For ties (especially relevant when no feature in a standard is used), we default to the earliest feature available.

3.5 CVEs

We collected information about browser vulnerabilities by finding all Common Vulnerabilities and Exposures (CVEs) [54] (security-relevant bugs discovered in software) related to Firefox that have been documented in the last three years.

The CVE database lists 470 issues from the last three years that mention Firefox. On manual inspection we found 14 of these were not actually issues in Firefox, but issues in other web-related software where Firefox was used to demonstrate the vulnerability.

Of the remaining 456 CVEs, we were able to manually associate 111 CVEs with a specific web standard. For example, CVE-2013-0763 [52] describes a potential remote execution vulnerability introduced in Firefox's implementation of the *WebGL* [26] standard, and CVE-2014-1577 [53] documents a potential information-disclosing bug related to Firefox's implementation of the *Web Audio API* standard.

We note that this work only considers CVEs associated with JavaScript accessible features. It does not include CVEs reporting vulnerabilities in other parts of the browser. For example, if a CVE reported an vulnerability due to the implementation of a `SVG` manipulating JavaScript function, we included it in our analysis. If, however, the CVE dealt with some other issue in Firefox's SVG handling, such as parsing the text of a SVG document, we did not consider it in this work.

3.6 Blocking Extensions

Finally, this work pulls from commercial and crowd-sourced browser extensions, which are popularly used to modify the browser environment.

This work uses two such browser extensions, Ghostery and AdBlock Plus. Ghostery is a browser extension that allows users to increase their privacy online by modifying the browser to not load resources or set cookies associated with cross-domain passive tracking, as determined by the extension's maintainer, Ghostery, Inc..

This work also uses the AdBlock Plus browser extension, which modifies the browser to not load resources the extension associates with advertising, and to hide elements in the page that are advertising related. AdBlock Plus draws from a crowdsourced list of rules and URLs to determine if a resource is advertising-related.

This work uses the default configuration for each browser extension, including the default rule sets for which elements and resources to block. No changes were made to the configuration or implementation of either extension.

4. METHODOLOGY

To understand browser feature use on the open web, we conducted a survey of the Alexa 10k, visiting each site ten times and recording which browser features were used. We visited each site five times with an unmodified browsing environment, and five times with popular tracking-blocking and advertising-blocking extensions installed. This section describes the goals of this survey, followed by how we instrumented the browser to determine which features are used on a given site, and then concludes with how we used our instrumented browser to measure feature use on the web in general.

4.1 Goals

The goal of our automated survey is to determine which browser features are used on the web as it is commonly experienced by users. This requires us to take a broad-yet-representative sample of the web, and to exhaustively determine the features used by those sites.

To do so, we built a browser extension to measure which features are used when a user interacts with a website. We then chose a representative sample of the web to visit. Finally, we developed a method for interacting with these sites in an automated fashion to elicit the same functionality that a human web user would experience. Each of these steps is described in detail in the proceeding subsections.

This automated approach only attempts to measure the "open web", or the subset of webpage functionality that a user encounters *without* logging into a website. Users may encounter different types of functionality when interacting with websites they have created accounts for and established relationships with, but such measurements are beyond the scope of this paper. We note this restriction, only measuring functionality used by non-authenticated portions of websites, as a limitation of this paper and a possible area for future work.

4.2 Measuring Extension

We instrumented a recent version of the Firefox web browser (version 46.0.1) with a custom browser extension to records each time a JavaScript feature has been used on a visited page. Our extension injects JavaScript into each page after the browser has created the DOM for that page, but before the page's content has been loaded. By injecting our instrumenting JavaScript into the browser before the page's content has been fetched and rendered, we can modify the methods and properties in the DOM before it becomes available to the requested page.

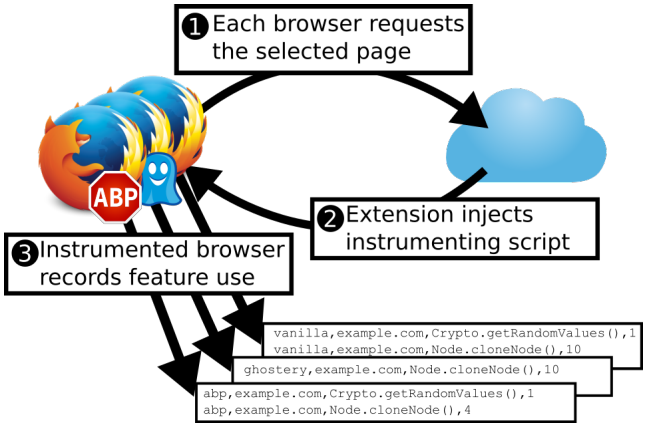

Figure 2: One iteration of the feature invocation measurement process.

The JavaScript that the extension injects into each requested page modifies the DOM to count when an instrumented method is called or that an instrumented property is written to. How the extension measures these method calls and property writes is detailed in the following two subsections. Figure 2 presents a representative diagram of the crawling process.

4.2.1 *Measuring Method Calls*

The browser extension counts when a method has been invoked by overwriting the method on the containing object's prototype. This approach allows us to shim in our own logging functionality for each method call, and then call the original method to preserve the original functionality. We replace each reference to each instrumented method in the DOM with an extension managed, instrumented method.

We take advantage of closures in JavaScript to ensure that web pages are not able to bypass the instrumented methods by looking up–or otherwise directly accessing–the original versions of each method.

4.2.2 *Measuring Property Writes*

Properties were more difficult to instrument. JavaScript provides no way to intercept when a property has been set or read on a client script-created object, or on an object created after the instrumenting code has finished executing. However, through the use of the non-standard `Object.watch()`[39] method in Firefox, we were able to capture when pages set properties on the singleton objects in the browser (e.g. `window`, `window.document`, `window.navigator`). Using this `Object.watch()` method allowed the extension to capture and count all writes to properties on singleton objects in the DOM.

There are a small number of features in the `DOM` where we were not able to intercept property writes. We were therefore unable to count how frequently these features were used. These features, found primarily in older standards, are properties where writes trigger side effects on the page.

The most significant examples of such properties are `document.location` (where writing to the property can trigger page redirection) and `Element.innerHTML` (where writing to the property causes the subtree in the document to be replaced). The implementation of these features in Firefox make them unmeasurable using our technique. We note them here as a small but significant limitation of our measurement technique.

4.2.3 Other Browser Features

Web standards define other features in the browser too, such as browser events and CSS layout rules, selectors, and instructions. Our extension-based approach did not allow us to measure the use of these features, and so counts of their use are not included in this work.

In the case of standard defined browser events (e.g. `onload`, `onmouseover`, `onhover`) the extension could have captured some event registrations through a combination of watching for event registrations with `addEventListener` method calls and watching for property-sets to singleton objects. However, we would not have been able to capture event registrations using the legacy `DOM0` method of event registration (e.g. assigning a function to an object's `onclick` property to handle click events) on non-singleton objects. Since we would only have been able to see a subset of event registrations, we decided to omit events completely from this work.

Similarly, this work does not consider non-JavaScript exposed functionality defined in the browser, such as CSS selectors and rules. While interesting, this work focuses solely on functionality that the browser allows websites to access though JavaScript.

4.3 Eliciting Site Functionality

Using our feature-detecting browser extension, we were able to measure which browser features are used on the 10k most popular websites. The following subsections describe how we simulated human interaction with web pages to measure feature use, first with the browser in its default state, and again with the browser modified with popular advertising and tracking blocking extensions.

4.3.1 Default Case

To understand which features are used in a site's execution, we installed the instrumenting extension described in Section 4.2 and visited sites from the Alexa 10k, with the goal of exercising as much of the functionality used on the page as possible. While some JavaScript features of a site are automatically activated on the home page (e.g. advertisements and analytics), many features will only be used as a result of user interaction, either within the page or by navigating to different areas of the site. Here we explain our strategy for crawling and interacting with sites.

In order to trigger as many browser features as possible on a website, we used a common site testing methodology called "monkey testing". Monkey testing refers to the strategy of instrumenting a page to click, touch, scroll, and enter text on random elements or locations on the page. To accomplish this, we use a modified version of gremlins.js [62], a library built for monkey testing front-end website interfaces. We modified the gremlins.js library to allow us to distinguish between when the gremlins.js script uses a feature, and when the site being visited uses a feature. The former feature usage is omitted from the results described in this paper.

We started our measurement by visiting the home page of site and allowing the monkey testing to run for 30 seconds. Because the randomness of monkey testing could cause navigation to other domains, we intercepted and prevented any interactions which might navigate to a different page. For navigations that would have been to the local domain, we noted which URLs the browser would have visited in the absence of the interception.

We then proceeded in a breadth first search of the site's hierarchy using the URLs that would have been visited by the actions of the monkey testing. We selected 3 of these URLs that were on the same domain (or related domain, as determined by the Alexa data), and visited each, repeating the same 30 second monkey testing procedure and recording all used features. From each of these 3 sites, we then visited three more pages for 30 seconds, which resulted in a total of 13 pages interacted with for a total of 390 seconds per site.

If more than three links were clicked during any stage of the monkey testing process, we selected which URLs to visit by giving preference to URLs where the path structure of the URL had not been previously seen. In contrast to traditional interface fuzzing techniques, which have as a goal finding unintended or malicious functionality [5, 35], we were interested in finding all functionalities that users will commonly interact with. By selecting URLs with different path-segments, we tried to visit as many types of pages on the site as possible, with the goal of capturing all of the functionality on the site that a user would encounter. The robustness and validity our strategy are evaluated in Section 6.

4.3.2 Blocking Case

In addition to the default case measurements described in Section 4.3.1, we also re-ran the same measurements against the Alexa 10k with an ad blocker (AdBlock Plus) and a tracking-blocker (Ghostery) to generate a second, 'blocking', set of measurements. We treat these blocking extensions as representative of the types of modifications users make to customize their browsing experience. While a so-modified version of a site no longer represents its author's intended representation (and may in fact break the site), the popularity of these content blocking extensions shows that this blocking case is a common valid alternative experience of a website.

4.3.3 Automated Crawl

Domains measured	9,733
Total website interaction time	480 days
Web pages visited	2,240,484
Feature invocations recorded	21,511,926,733

Table 1: Amount of data gathered regarding JavaScript feature usage on the Alexa 10k. "Total website interaction time" is an estimate based on the number of pages visited and 30 seconds of page interaction per visit.

For each site in the Alexa 10k, we repeated the above procedure ten times to ensure we measured all features used on the page, first five times in the default case, and then again five times in the blocking case. By parallelizing this crawl with 64 Firefox installs operating over 4 machines, we were able to complete the crawl in two days.

We present findings for why five times is sufficient to induce all types of site functionality in each test case in Section 6. Table 1 presents some high level figures of this automated crawl. For 267 domains, we were unable to measure feature usage for a variety of reasons, including non-responsive domains and sites that contained syntax errors in their JavaScript code that prevented execution.

5. RESULTS

In this section we discuss our findings, including the popularity distribution of JavaScript features used on the web with and without blocking, a feature's popularity in relation to its age, which features are disproportionately blocked, and which features are associated with security vulnerabilities.

5.1 Definitions

This work uses the term **feature popularity** to denote the percentage of sites that use a given feature at least once during automated interaction with the site. A feature that is used on every site has a popularity of 1, and a feature that is never seen has a popularity of 0.

Similarly, we use the term **standard popularity** to denote the percentage of sites that use at least one feature from the standard at least once during the site's execution.

Finally, we use the term **block rate** to denote how frequently a feature would have been used if not for the presence of an advertisement- or tracking-blocking extension. Browser features that are used much less frequently on the web when a user has AdBlock Plus or Ghostery installed have high block rates, while features that are used on roughly the same number of websites in the presence of blocking extensions have low block rate.

5.2 Standard Popularity

In this subsection, we present measurements of the popularity of the standards in the browser, first in general, then followed by comparisons to the individual features in each standard, the popularity of sites using each standard, and when the standard was implemented in Firefox.

5.2.1 Overall

Figure 3 displays the cumulative distribution of standard popularity. Some standards are extremely popular, and others are extremely unpopular: six standards are used on over 90% of all websites measured, and a full 28 of the 75 standards measured were used on 1% or fewer sites, with eleven not used at all. Standard popularity is not feast or famine however, as standards see several different popularity levels between those two extremes.

5.2.2 Standard Popularity By Feature

We find that browser features are not equally used on the web. Some features are extremely popular, such as the `Document.prototype.createElement` method, which allows sites to create new page-elements. The feature is used on 9,079–or over 90%–of pages in the Alexa 10k.

Other browser features are never used. 689 features, or almost 50% of the 1,392 implemented in the browser, are never used in the 10k most popular sites. A further 416 features are used on less than 1% of the 10k most popular websites. Put together, this means that over 79% of the

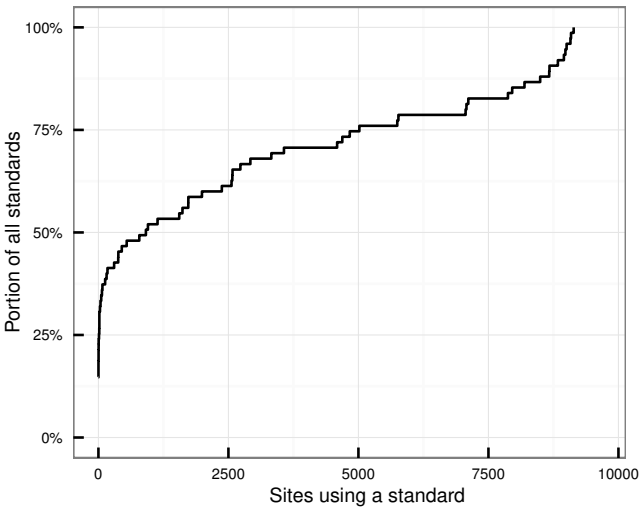

Figure 3: Cumulative distribution of standard popularity within the Alexa 10k.

features available in the browser are used by less than 1% of the web.

We also find that browser features do not have equal block rates; some features are blocked by advertisement and tracking blocking extensions far more often than others. Ten percent of browser features are prevented from executing over 90% of the time when browsing with common blocking extensions. We also find that 1,159 features, or over 83% of features available in the browser, are executed on less than 1% of websites in the presence of popular advertising and tracking blocking extensions.

5.3 Standard Popularity vs. Site Popularity

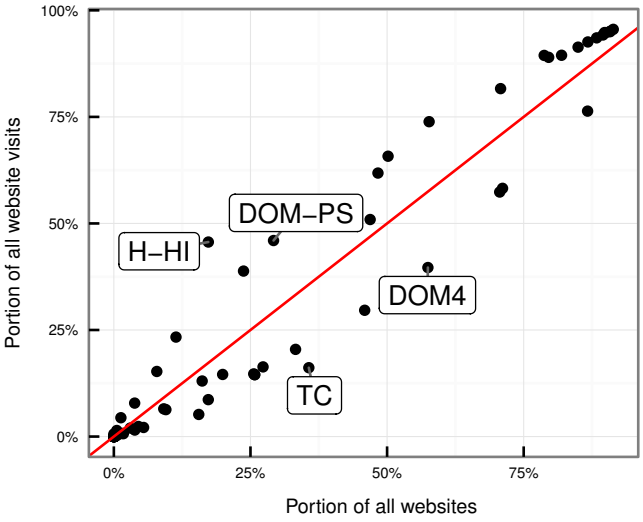

Figure 4: Comparison of percentage of sites using a standard versus percentage of web traffic using a standard.

The results described in this paper give equal weight to all sites in the Alexa 10k. If the most popular and least popular sites use the same standard, both uses of that standard are given equal consideration. In this section we examine the accuracy of this assumption by measuring the differ-

ence between the number of sites using a standard, and the percentage of site visits using a standard.

Figure 4 shows the results of this comparison. The x-axis shows the percentage of sites that use at least one feature from a standard, and the y-axis shows the estimated percentage of site views on the web that use this standard. Standards above the `x=y` line are more popular on frequently visited sites, meaning that the percentage of page views using the standard is greater than the percentage of sites using the standard. A site on the `x=y` line indicates that the feature is used exactly as frequently on popular sites as on less popular sites.

Generally, the graph shows that standard usage is not equally distributed, and that some standards are more popular with frequently visited sites. However, the general trend appears to be for standards to cluster around the `x=y` line, indicating that while there are some differences in standard usage between popular and less popular sites, they do not affect our general analysis of standard usage on the web.

Therefore, for the sake of brevity and simplicity, all other measures in this paper treat standard use on all domains as equal, and do not consider a site's popularity.

In addition to the datasets used in this paper, we have also collected data from even-less popular sites from the Alexa one-million, sites with rank less than 10k, to determine whether feature usage in less popular portions of the web differs significantly from feature usage patterns in the Alexa 10k. That measurement found no significant difference in feature usage. We therefore, as a simplifying assumption, treat the Alexa 10k as representative of the web in general.

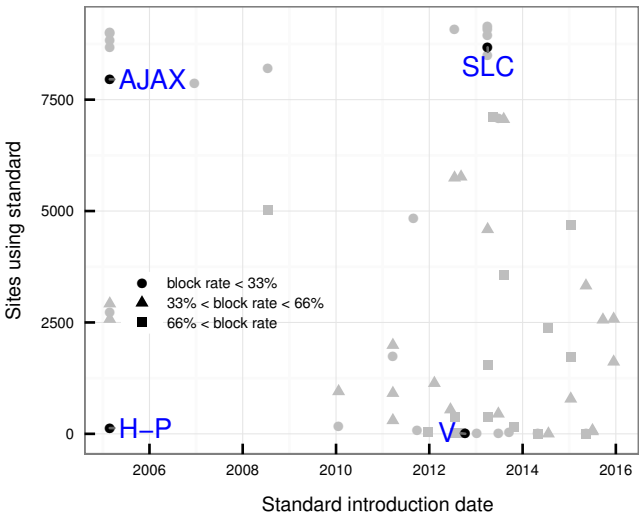

Figure 5: Comparison of a standard's availability date, and its popularity.

5.4 Standard Popularity By Introduction Date

We also measured the relationship between when a standard became available in the browser, its popularity, and how frequently its execution is prevented by popular blocking extensions.

As the graph shows, there is no simple relationship between when a standard was added to the browser, how frequently the standard is used on the web, and how frequently the standard is blocked by common blocking extensions. However, as Figure 5 indicates, some standards have become extremely popular over time, while others, both recent and old, have languished in disuse. Furthermore, it appears that some standards have been introduced extremely recently but have nevertheless been readily adopted by web authors.

Old, Popular Standards. For example, point **AJAX** depicts the *XMLHttpRequest* [58], or *AJAX* standard, used to send information to a server without fetching the entire document again. This standard has been available in the browser for almost as long as Firefox has been released (since 2004), and is extremely popular; the standard's most popular feature, `XMLHttpRequest.prototype.open`, is used by 7,955 sites in the Alexa 10k. Standards in this portion of the graph have been in the browser for a long time, and appear on a large fraction of sites. This cluster of standards have block rates of less than 50%, considered low in this study.

Old, Unpopular Standards. Other standards, despite existing in the browser nearly since Firefox's inception, are much less popular on the web. Point **H-P** shows the *HTML: Plugins* [22] standard, a subsection of the larger *HTML* standard that allows document authors to detect the names and capabilities of plugins installed in the browser (such as Flash, Shockwave, Silverlight, etc.). The most popular features of this standard have been available in Firefox since 2005. However, the standard's most popular feature, `PluginArray.prototype.refresh`, which checks for changes in browser plugins, is used on less than 1% of current websites (90 sites).

New, Popular Standards. Point **SEL** depicts the *Selectors API Level 1* [59] standard, which provides site authors with a simplified interface for selecting elements in a document. Despite being a relatively recent addition to the browser (the standard was added in 2013), the most popular feature in the standard–`Document.prototype.querySelectorAll`–is used on over 80% of websites. This standard, and other standards in this area of the graph, have low block rates.

New, Unpopular Standards. Point **V** shows the *Vibration* [30] standard, which allows site authors to trigger a vibration in the user's device on platforms that support it. Despite this standard having been available in Firefox longer than the previously mentioned *Selectors API Level 1* standard, the *Vibration* standard is significantly less popular on the web. The sole method in the standard, `Navigator.prototype.vibrate`, is used only once in the Alexa 10k.

5.5 Standard Blocking

Many users alter their browsing environment when visiting websites. They do so for a variety of reasons, including wishing to limit advertising displayed on the pages they read, reducing their exposure to malware distributed through advertising networks, and increasing their privacy by reducing the amount of tracking they experience online. These browser modifications are typically made by installing browser extensions.

We measured the effect of installing two common browser extensions, AdBlock Plus and Ghostery, on the type and number of features that are executed when visiting websites.

5.5.1 Popularity vs. Blocking

Ad and tracking blocking extensions do not block the use of all standards equally; some standards are blocked far more of often that others. Figure 6 depicts the relationship between a standard's popularity (represented by the number of sites the standard was used on, log scale) and its block rate. Since a standard's popularity is the number of sites where a feature in a standard is used at least once, the popularity of the standard is equal to at least the popularity of the most popular feature in the standard.

Each quadrant of the graph tells a different story about the popularity and the block rate of a standard on the web.

Popular, Unblocked Standards. The upper-left quadrant contains the standards that occur very frequently on the web, and are rarely blocked by advertising and tracking blocking extensions.

One example, point **CSS-OM**, depicts the *CSS Object Model* [43] standard, which allows JavaScript code to introspect, modify and add to the styling rules in the document. It is positioned near the top of the graph because 8,193 sites used a feature from the standard at least once during measurement. The standard is positioned to the left of the graph because the standard has a low block rate (12.6%), meaning that the addition of blocking extensions had relatively little effect on how frequently sites used any feature from the standard.

Popular, Blocked Standards. The upper-right quadrant of the graph shows standards that are used by a large percentage of sites on the web, but which blocking extensions frequently prevent from executing.

A representative example of such a standard is the *HTML: Channel Messaging* [21] standard, represented by point **H-CM**. This standard contains JavaScript methods allowing embedded documents (`iframes`) and windows to communicate with their parent document. This functionality is often used by embedded-content and pop-up windows to communicate with the hosting page, often in the context of advertising. This standard is used on over half of all sites by default, but is prevented from being executed over 77% of the time in the presence of blocking extensions.

Unpopular, Blocked Standards. The lower-right quadrant of the graph shows standards that are rarely used by websites, and that are almost always prevented from executing by blocking extensions.

Point **ALS** shows the *Ambient Light Events* standard [55], which defines events and methods allowing a website to react to changes to the level of light the computer, laptop or mobile phone is exposed to. The standard is rarely used on the web (14 out of 10k sites), but is prevented from being executed 100% of the time by blocking extensions.

Unpopular, Unblocked Standards. The lower-left quadrant of the graph shows standards that were rarely seen in our study, and were rarely prevented from executing. Point **E** shows the *Encodings* [57] standard. This standard allows JavaScript code to read and convert text between different text encodings, such as reading text from a document encoded in *GBK* and inserting it into a website encoded in *UTF-8*.

The *Encodings* [57] standard is rarely used on the web, with only 1 of the Alexa 10k sites attempting to use it. However, the addition of an advertising or tracking blocking extension had no affect on the number of times the standard was used; this sole site still used the *Encodings* standard when AdBlock Plus and Ghostery were installed.

5.5.2 Blocking Frequency

As discussed in 5.5.1, blocking extensions do not block all browser standard usage equally. As Figure 6 shows, some standards are greatly impacted by installing these advertising and tracking blocking extensions, while others are not impacted at all.

For example, the *Beacon* [20] standard, which allows websites to trigger functionality when a user leaves a page, has a 83.6% reduction in usage when browsing with blocking extensions. Similarly, the *SVG* standard, which includes functionality that allows for fingerprinting users through font enumeration[2], sees a similar 86.8% reduction in site usage when browsing with blocking extensions.

Other browser standards, such as the core *DOM* standards, see little reduction in use in the presence of blocking extensions.

5.5.3 Blocking Purpose

In addition to measuring which standards were blocked by extensions, we also distinguished which extension did the blocking. Figure 7 plots standards' block rates in the presence of an advertising blocking extension (x-axis), versus standards' block rates when a tracking-blocking extension is installed (y-axis).

Points on the `x=y` line in the graph are standards that were blocked equally in the two cases, with points closer to the upper-right corner being blocked more often (in general), and points closer to the lower-left corner being blocked less often (in general).

Points in the upper-left depict standards that were blocked more frequently by the tracking-blocking extension than the advertising-blocking extension, while points in the lower-right show standards that were blocked more frequently by the advertising-blocking extension.

As the graph shows, some standards, such as *WebRTC* [9] (which is associated with attacks revealing the user's IP address), *WebCrypto API* [25] (which is used by some analytics libraries to generate identifying nonces), and *Performance Timeline Level 2* [18] (which is used to generate high resolution time stamps) are blocked by tracking-blocking extensions more often than they are blocked by advertisement blocking extensions.

The opposite is true, to a lesser extent, for the *UI Events Specification* [19] standard, which specifies new ways that sites can respond to user interactions.

5.6 Vulnerabilities

Just as all browser standards are not equally popular on the web, neither are all standards equally associated with known vulnerabilities in Firefox. The implementations of some standards have been associated with a large number of vulnerabilities, while others have not been associated with any publicly known issues. Here we investigate the link between individual browser standards and known security vulnerabilities (in the form of filed CVEs), as well as compare these metrics with the relative popularity and block rates of those standards.

[2]The `SVGTextContentElement.prototype.getComputedTextLength` method

Standard Name	Abbreviation	# Features	# Sites	Block Rate	# CVEs
HTML: Canvas	H-C	54	7,061	33.1%	15
Scalable Vector Graphics 1.1 (2nd Edition)	SVG	138	1,554	86.8%	14
WebGL	WEBGL	136	913	60.7%	13
HTML: Web Workers	H-WW	2	952	59.9%	11
HTML 5	HTML5	69	7,077	26.2%	10
Web Audio API	WEBA	52	157	81.1%	10
WebRTC 1.0	WRTC	28	30	29.2%	8
XMLHttpRequest	AJAX	13	7,957	13.9%	8
DOM	DOM	36	9,088	2.0%	4
Indexed Database API	IDB	48	302	56.3%	3
Beacon	BE	1	2,373	83.6%	2
Media Capture and Streams	MCS	4	54	49.0%	2
Web Cryptography API	WCR	14	7,113	67.8%	2
CSSOM View Module	CSS-VM	28	4,833	19.0%	1
Fetch	F	21	77	33.3%	1
Gamepad	GP	1	3	0.0%	1
High Resolution Time, Level 2	HRT	1	5,769	50.2%	1
HTML: Web Sockets	H-WS	2	544	64.6%	1
HTML: Plugins	H-P	10	129	29.3%	1
Web Notifications	WN	5	16	0.0%	1
Resource Timing	RT	3	786	57.5%	1
Vibration API	V	1	1	0.0%	1
Battery Status API	BA	2	2,579	37.3%	0
CSS Conditional Rules Module, Level 3	CSS-CR	1	449	36.5%	0
CSS Font Loading Module, Level 3	CSS-FO	12	2,560	33.5%	0
CSS Object Model (CSSOM)	CSS-OM	15	8,193	12.6%	0
DOM, Level 1 - Specification	DOM1	47	9,139	1.8%	0
DOM, Level 2 - Core Specification	DOM2-C	31	8,951	3.0%	0
DOM, Level 2 - Events Specification	DOM2-E	7	9,077	2.7%	0
DOM, Level 2 - HTML Specification	DOM2-H	11	9,003	4.5%	0
DOM, Level 2 - Style Specification	DOM2-S	19	8,835	4.3%	0
DOM, Level 2 - Traversal and Range Specification	DOM2-T	36	4,590	33.4%	0
DOM, Level 3 - Core Specification	DOM3-C	10	8,495	3.9%	0
DOM, Level 3 - XPath Specification	DOM3-X	9	381	79.1%	0
DOM Parsing and Serialization	DOM-PS	3	2,922	60.7%	0
execCommand	EC	12	2,730	24.0%	0
File API	FA	9	1,991	58.0%	0
Fullscreen API	FULL	9	383	79.9%	0
Geolocation API	GEO	4	174	13.1%	0
HTML: Channel Messaging	H-CM	4	5,018	77.4%	0
HTML: Web Storage	H-WS	8	7,875	29.2%	0
HTML	HTML	195	8,980	4.3%	0
HTML: History Interface	H-HI	6	1,729	18.7%	0
Media Source Extensions	MSE	8	1,616	37.5%	0
Performance Timeline	PT	2	4,690	75.8%	0
Performance Timeline, Level 2	PT2	1	1,728	93.7%	0
Selection API	SEL	14	2,575	36.6%	0
Selectors API, Level 1	SLC	6	8,674	7.7%	0
Timing control for script-based animations	TC	1	3,568	76.9%	0
UI Events Specification	UIE	8	1,137	56.8%	0
User Timing, Level 2	UTL	4	3,325	33.7%	0
DOM4	DOM4	3	5,747	37.6%	0
Non-Standard	NS	65	8,669	24.5%	0

Table 2: Popularity and blockrate for the web standards that are used on at least 1% of the Alexa 10k or have at least one associated CVE advisory in the last three years.
Columns one and two list the name and abbreviation of the standard.
Column three gives the number of features (methods and properties) from that standard that we were able to instrument.
Column four includes the number of pages that used at least one feature from the standard, out of the entire Alexa 10k.
Column five shows the number of sites on which *no features* in the standard executed in the presence of advertising and tracking blocking extensions (given that the website executed *at least one feature* from the standard in the default case), divided by the number of pages where at least one feature from the standard was executed. In other words, how often the blocking extensions prevented all features in a standard from executing, given at least one feature would have been used.
Column six shows the number of CVEs associated with this standard's implementation in Firefox within the last three years.

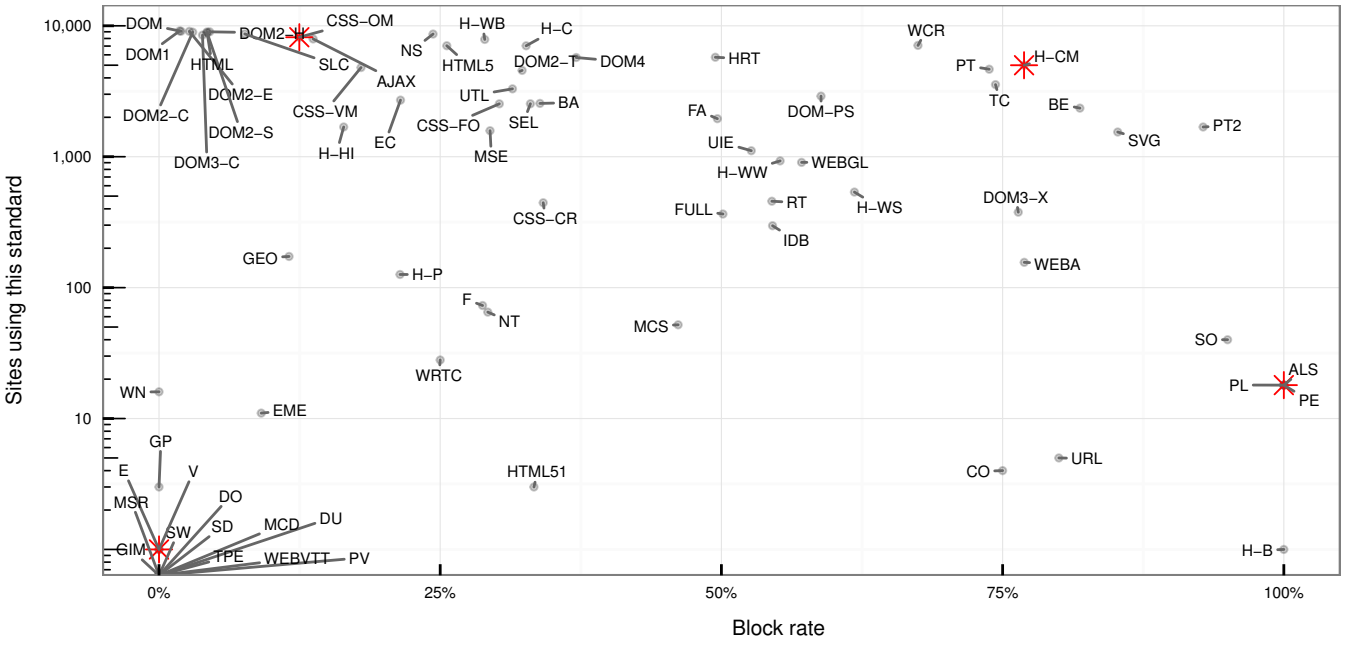

Figure 6: Popularity of standards versus their block rate, on a log scale.

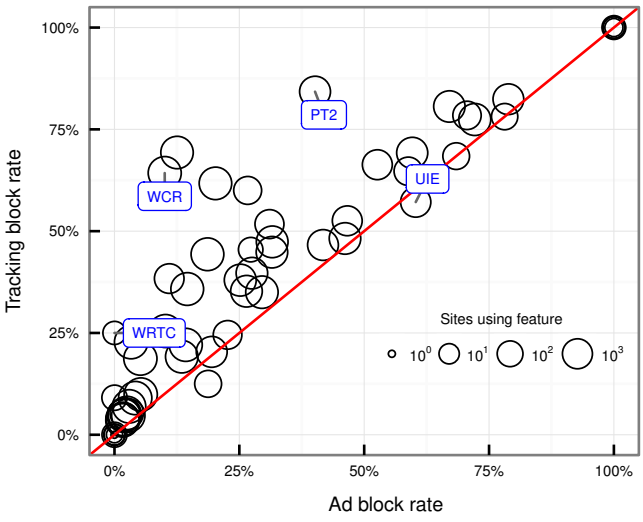

Figure 7: Comparison of block rates of standards using advertising vs. tracking blocking extensions.

Column five of table 2 shows the number of CVEs associated with the standard's implementation in Firefox within the last three years. As the table shows, some implementations of web standards have been associated with a large number of security bugs even though those standards are not popular on the web. Other standards are associated with a large number of security vulnerabilities despite being blocked by advertising and tracking blocking extensions.

For example, the *Web Audio API* [4] standard is unpopular with website authors, and implementing it the browser though has exposed users to a substantial number of security vulnerabilities. We observed the *Web Audio API* standard in use on fewer that 2% of sites in our collection, but its implementation in Firefox is associated with at least 10 CVEs in the last 3 years. Similarly, *WebRTC* [9] is used on less than 1% of sites in the Alexa 10k, but is associated with 8 CVEs in the last 3 years.

The *Scalable Vector Graphics* [13] standard is an example of a frequently blocked standard that has been associated with a significant number of vulnerabilities. The standard is very frequently blocked by advertising and tracking blocking extensions; the standard is used on 1,554 sites in the Alexa 10k, but is prevented from executing in 87% of cases. At least 14 CVE's have been reported against Firefox's implementation of the standard in the last 3 years.

5.7 Site Complexity

We also evaluated sites based on their complexity. We define complexity as the number of standards used on a given website. As Figure 8 shows, most sites use many standards: between 14 and 32 of the 74 available in the browser. No site used more than 41 standards, and a second mode exists around the zero mark, showing that a small but measurable number of sites use little to no JavaScript at all.

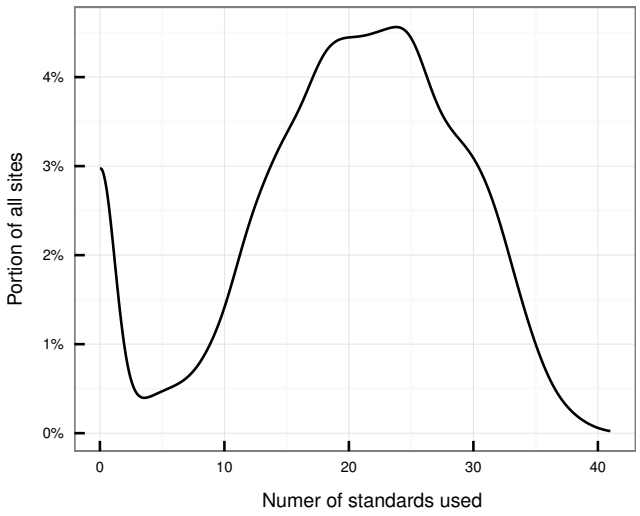

Figure 8: Probability density function of number of standards used by sites in the Alexa 10k.

6. VALIDATION

This study measures the features executed over repeated, automated interactions with a website. We treat these automated measurements as representative of the features that would be executed when a human visits the website.

Thus, our work relies on our automated measurement technique triggering (at least) the browser functionality a human user's browser will execute when interacting with the same website. This section explains how we verified this assumption to be reasonable.

6.1 Internal Validation

Round #	Avg. New Standards
2	1.56
3	0.40
4	0.29
5	0.00

Table 3: Average number of new standards encountered on each subsequent automated crawl of a domain.

As discussed in Section 4.3.1, we applied our automated measurement technique to each site in the Alexa 10k ten times, five times in an unmodified browser, and five times with blocking extensions in place. We measured five times in each condition with the goal of capturing the full set of functionality used on the site, since the measurement's random walk technique means that each subsequent measurement may encounter different, new parts of the site.

A natural question then is whether five measurements are sufficient to capture all potentially encountered features per site, or whether additional measurements are necessary. To ensure that five measurements were sufficient, we examined how many new standards were encountered on each round of measurement. If new standards were still being encountered in the final round of measurement, it would indicate we had not measured enough, and that our data painted an incomplete picture of the types of features used by each site.

Table 3 shows the results of this verification. The first column lists each round of measurement, and the second column lists the number of new standards encountered that had not yet been observed in the previous rounds (averaged across the entire Alexa 10k). As the table shows, the average number of new standards observed on each site decreased with each measurement, until the 5th measurement for each site, at which point we did not observe any new features being executed on any site.

From this we concluded that five rounds was sufficient for each domain, and that further automated measurements of these sites were unlikely to observe new feature usage.

6.2 External Validation

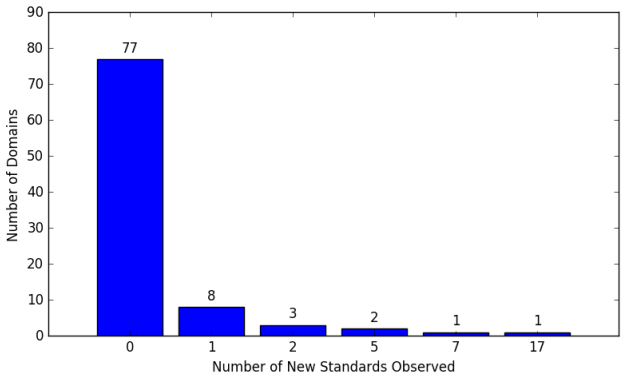

Figure 9: Histogram of the number of standards encountered on a domain under manual interaction that were not encountered under automated interaction.

We also tested whether our automated technique observed the same feature use as human web users encounter. We randomly chose 100 sites to visit from the Alexa 10k and interacted with each for 90 seconds in a casual web browsing fashion. This included reading articles and blog posts, scrolling through websites, browsing site navigation listings, etc.

We interacted with the home page of the site (the page directed to from the raw domain) for 30 seconds, then clicked on a prominent link we thought a typical human browser would choose (such as the headline of a featured article) and interacted with this second page for 30 more seconds. We then repeated the process a third time, loading a third page that was interacted with for another 30 seconds.

After omitting pornographic and non-English sites, we completed this process for 92 different websites. We then compared the features used during manual interaction with our automated measurements of the same sites. Figure 9 provides a histogram of this comparison, with the x-axis showing the number of new standards observed during manual interaction that *were not* observed during the automated interaction. As the graph shows, in the majority of cases (83.7%), no features were observed during manual interaction that the automated measurements did not catch.

The graph also shows a few outliers, including a very significant one, where manual interaction triggered standards that our automated technique did not. On closer inspection, this outlier was due to the site updating its content between when we performed the automated measurement and the

manual measurement. The outlier site, `buzzfeed.com`, is a website that changes its front page content hour to hour. The site further features subsections that are unlike the rest of the site, and can have widely differing functionality, resulting in very different standard usage over time. We checked to see if standards were used under manual evaluation of the outlier that were not observed during automated testing on the rest of the Alexa 10k, and did not find any.

From this we conclude that our automated measurement technique did a generally accurate job of elicit the feature use a human user would encounter on the web, even if the technique did not perfectly emulate human feature use in all cases.

7. DISCUSSION

In this section, we discuss the potential ramifications of these findings, including what our results mean for browser complexity.

7.1 Popular and Unpopular Browser Features

There are a small number of standards in the browser that are extremely popular with website authors, providing features that are necessary for for modern web pages to function. These standards provide functionality like querying the document for elements, inspecting and validating forms, and making client-side page modifications.[3]

A much larger portion of the browser's functionality, however, is unused by most site authors. Eleven JavaScript-exposed standards in Firefox are completely unused in the ten-thousand most popular websites, and 28 (nearly 37% of standards available in the browser) are used by less than 1% of sites.

While many unpopular standards are relatively new to the browser, youth alone does not explain the extreme unpopularity of most features in the browser on the open web. Lesser used features may be of interest only to those creating applications which require authentication, or to only small niches of developers and site visitors.

7.2 Blocked Browser Features

When users employ common advertising and tracking blocking extensions, they further reduce the frequency and number of standards that are executed. This suggests that some standards are primarily used to support the advertising and tracking infrastructure built into the modern web. When users browse with these common extensions installed, four additional standards go unused on the web (a total of 15 standards, or 20% of those available in the browser). An additional 20 standards become used on less than 1% of websites (for a total of 31 standards, or 41% of standards in the browser). 16 standards are blocked over 75% of the time by blocking extensions.

Furthermore, while content blocker rules do not target JavaScript APIs directly, that a standard like *SVG* [13], used on 16% of the Alexa 10k, would be prevented from running 87% of the time is circumstantial evidence that whatever website functionality this enables is not necessary to the millions of people who use content blocking extensions. This phenomenon lends credence to what has been called "the Website Obesity Crisis" - the conjecture that websites include far more functionality than is actually necessary to serve users' goals [12].

The presence of a large amount of unused functionality in the browser seems to contradict the common security principal of least privilege, or of giving applications only the capabilities they need to accomplish their intended task. This principal exists to limit attack surface and limit the unforeseen security risks that can come from the unexpected, and unintended, composition of features. As the list of CVEs in Figure 2 shows, unpopular and heavily blocked features have imposed substantial security costs to the browser.

7.3 Future Work

This study develops and validates the use of monkey testing to elicit browser feature use on the open web. The closed web (i.e. web content and functionality that is only available after logging in to a website) may use a broader set of features. With the correct credentials, the monkey testing approach could be used to evaluate "closed" websites, although it may need to be improved with a rudimentary understanding of site semantics.

Finally, a more complete treatment of the security implications of these broad APIs would be valuable. In recent years, plugins like Java and Flash have become less popular, and the native capabilities of browsers have become more impressive. The modern browser is a monolithic intermediary between web applications and user hardware, like an operating system. For privacy conscious users or those with special needs (like on public kiosks, or electronic medical record readers), understanding the privacy and security implications of this broad attack surface is important.

8. CONCLUSION

The Web API offers a standardized API for programming across operating systems and web browsers. This platform has been tremendously useful in the success of the web as a platform for content dissemination and application distribution. Feature growth has enabled the modern web, built on JavaScript and offering functionality like video, games, and productivity applications. Applications that were once only possible as native apps or external plugins are now implemented in JavaScript in the browser.

Over time, more features have been standardized and implemented in the browser. Some of these features have been readily adopted by websites to implement new types of applications; other features are infrequently or never used.

Beyond this popularity divide, however, are features which are blocked by content blockers in the vast majority of attempted uses. That these features are simultaneously popular with site authors but overwhelmingly blocked by site users signals that these features may exist in the browser to serve the needs of the site author rather than the site visitor.

This work documents that much of the JavaScript-accessible functionality in the browser is unused by websites, and even more of it goes unused when popular ad and tracking blocking extensions are installed. Some of this unpopular functionality has been implicated in past security vulnerabilities. This work's findings may guide browser vendors, standards authors, and web users in deciding what features are necessary for a secure, vibrant, useful web.

[3]All of which are covered by the *Document Object Model (DOM) Level 1 Specification* standard, dating back to 1998.

9. ACKNOWLEDGEMENTS

We would like to thank our anonymous reviewers and our shepherd, Christo Wilson, for their feedback and assistance in improving this paper. We would also like to thank Oliver Hui and Daniel Moreno for their assistance performing manual website analysis. This work was supported in part by National Science Foundation grants CNS-1351058, CNS-1409868, and CNS-1405886.

References

[1] Chromium blink mailing list discussion. https://groups.google.com/a/chromium.org/forum/#!topic/blink-dev/1wWhVoKWztY, 2014. [Online; accessed 15-February-2016].

[2] Chromium blink web features guidelines. https://dev.chromium.org/blink#new-features, 2016. [Online; accessed 15-February-2016].

[3] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 674–689. ACM, 2014.

[4] P. Adenot, C. Wilson, and C. Rogers. Web audio api. http://www.w3.org/TR/webaudio/, 2013.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[6] V. Apparao, S. Byrne, M. Champion, S. Isaacs, A. L. Hors, G. Nicol, J. Robie, P. Sharpe, B. Smith, J. Sorensen, R. Sutor, R. Whitmer, and C. Wilson. Document object model (dom) level 1 specification. https://www.w3.org/TR/REC-DOM-Level-1/, 1998. [Online; accessed 10-May-2016].

[7] M. Ayenson, D. J. Wambach, A. Soltani, N. Good, and C. J. Hoofnagle. Flash cookies and privacy ii: Now with html5 and etag respawning. *Available at SSRN 1898390*, 2011.

[8] R. Balebako, P. Leon, R. Shay, B. Ur, Y. Wang, and L. Cranor. Measuring the effectiveness of privacy tools for limiting behavioral advertising. In *Web 2.0 Security and Privacy Workshop*, 2012.

[9] A. Bergkvist, D. C. Burnett, C. Jennings, A. Narayanan, and B. Aboba. Webrtc 1.0: Real-time communication between browser. https://www.w3.org/TR/webrtc/, 2016. [Online; accessed 10-May-2016].

[10] Black Duck Software Inc. The chromium (google chrome) open source project on open hub. https://www.openhub.net/p/chrome/analyses/latest/code_history, 2015. [Online; accessed 16-October-2015].

[11] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 313–328. ACM, 2011.

[12] M. Cegłowski. The website obesity crisis. http://idlewords.com/talks/website_obesity.htm, 2015.

[13] E. Dahlström, P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers, and J. Watt. Scalable vector graphics (svg) 1.1 (second edition). http://www.w3.org/TR/SVG11/, 2011.

[14] A. Deveria. Can i use. http://caniuse.com/. [Online; accessed 16-October-2015].

[15] D. Dorwin, J. Smith, M. Watson, and A. Bateman. Encrypted media extensions. http://www.w3.org/TR/encrypted-media/, 2015.

[16] P. Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.

[17] M. Falahrastegar, H. Haddadi, S. Uhlig, and R. Mortier. Anatomy of the third-party web tracking ecosystem. *arXiv preprint arXiv:1409.1066*, 2014.

[18] I. Grigorik, J. Mann, and Z. Wang. Performance timeline level 2. https://w3c.github.io/performance-timeline/, 2016. [Online; accessed 11-May-2016].

[19] I. Grigorik, J. Mann, and Z. Wang. Ui events. https://w3c.github.io/uievents/, 2016. [Online; accessed 11-May-2016].

[20] I. Grigorik, A. Reitbauer, A. Jain, and J. Mann. Beacon w3c working draft. http://www.w3.org/TR/beacon/, 2015.

[21] I. Hickson, S. Pieters, A. van Kesteren, P. Jägenstedt, and D. Denicola. Html: Channel messaging. https://html.spec.whatwg.org/multipage/comms.html#channel-messaging, 2016. [Online; accessed 10-May-2016].

[22] I. Hickson, S. Pieters, A. van Kesteren, P. Jägenstedt, and D. Denicola. Html: Plugins. https://html.spec.whatwg.org/multipage/webappapis.html#plugins-2, 2016. [Online; accessed 10-May-2016].

[23] A. L. Hors, P. L. Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document object model (dom) level 2 core specification. https://www.w3.org/TR/DOM-Level-2-Core/, 2000. [Online; accessed 10-May-2016].

[24] A. L. Hors, P. L. Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document object model (dom) level 3 core specification. https://www.w3.org/TR/DOM-Level-3-Core/, 2004. [Online; accessed 10-May-2016].

[25] A. L. Hors, P. L. Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Web cryptography api. https://www.w3.org/TR/WebCryptoAPI/, 2014. [Online; accessed 11-May-2016].

[26] D. Jackson. Webgl specification. https://www.khronos.org/registry/webgl/specs/1.0/, 2014.

[27] 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.

[28] S. Kamkar. Evercookie - virtually irrevocable persistent cookies. http://samy.pl/evercookie/,, 2015. [Online; accessed 15-October-2015].

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

[30] A. Kostiainen. Vibration. http://www.w3.org/TR/vibration/, 2105.

[31] A. Kostiainen, I. Oksanen, and D. Hazaël-Massieux. Html media capture. http://www.w3.org/TR/html-media-capture/, 2104.

[32] B. Krishnamurthy and C. Wills. Privacy diffusion on the web: a longitudinal perspective. In *Proceedings of the 18th international conference on World wide web*, pages 541–550. ACM, 2009.

[33] M. Lamouri and M. Cáceres. Screen orientation. http://www.w3.org/TR/screen-orientation/, 2105.

[34] F. Lardinois. Google has already removed 8.8m lines of webkit code from blink. http://techcrunch.com/2013/05/16/google-has-already-removed-8-8m-lines-of-webkit-code-from-blink/, 2013. [Online; accessed 12-May-2016].

[35] B. Liu, S. Nath, R. Govindan, and J. Liu. Decaf: detecting and characterizing ad fraud in mobile apps. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 57–70, 2014.

[36] A. M. McDonald and L. F. Cranor. Survey of the use of adobe flash local shared objects to respawn http cookies, a. *ISJLP*, 7:639, 2011.

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

[38] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in html5. *Proceedings of W2SP*, 2012.

[39] Mozilla Developer Network. Object.prototype.watch() - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/watch. [Online; accessed 16-October-2015].

[40] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, volume 5, 2013.

[41] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Symposium on Security and Privacy*, 2013.

[42] L. Olejnik, T. Minh-Dung, C. Castelluccia, et al. Selling off privacy at auction. In *Annual Network and Distributed System Security Symposium (NDSS). IEEE*, 2014.

[43] S. Pieters and D. Glazman. Css object model (css-om). https://www.w3.org/TR/cssom-1/, 2016. [Online; accessed 10-May-2016].

[44] E. Pujol, O. Hohlfeld, and A. Feldmann. Annoyed users: Ads and ad-block usage in the wild. In *IMC*, 2015.

[45] E. Rader. Awareness of behavioral tracking and information privacy concern in facebook and google. In *Proc. of Symposium on Usable Privacy and Security (SOUPS), Menlo Park, CA, USA*, 2014.

[46] M. Reavy. Webrtc privacy. https://mozillamediagoddess.org/2015/09/10/webrtc-privacy/, 2015. [Online; accessed 11-May-2016].

[47] Z. Rogoff. We've got momentum, but we need more protest selfies to stop drm in web standards. https://www.defectivebydesign.org/weve-got-momentum-but-we-need-more-protest-selfies, 2016. [Online; accessed 11-May-2016].

[48] A. Russell. Doing science on the web. https://infrequently.org/2015/08/doing-science-on-the-web/, 2015.

[49] P. Snyder, L. Ansari, C. Taylor, and C. Kanich. Web api usage in the alexa 10k. http://imdc.datcat.org/collection/1-0723-8=Web-API-usage-in-the-Alexa-10k, 2016.

[50] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle. Flash cookies and privacy. In *AAAI Spring Symposium: Intelligent Information Privacy Management*, volume 2010, pages 158–163, 2010.

[51] O. Sorensen. Zombie-cookies: Case studies and mitigation. In *Internet Technology and Secured Transactions (ICITST), 2013 8th International Conference for*, pages 321–326. IEEE, 2013.

[52] The MITRE Corporation. CVE-2013-0763. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0763, 2013. [Online; accessed 13-November-2015].

[53] The MITRE Corporation. CVE-2014-1577. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1577, 2014. [Online; accessed 13-November-2015].

[54] The MITRE Corporation. Common vulnerabilities and exposures. https://cve.mitre.org/index.html, 2015. [Online; accessed 13-November-2015].

[55] D. Turner and A. Kostiainen. Ambient light events. http://www.w3.org/TR/ambient-light/, 2105.

[56] T. Van Goethem, W. Joosen, and N. Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1382–1393. ACM, 2015.

[57] A. van Kesteren. Encoding standard. https://encoding.spec.whatwg.org/, 2016. [Online; accessed 11-May-2016].

[58] A. van Kesteren. Xmlhttprequest. https://xhr.spec.whatwg.org/, 2016. [Online; accessed 10-May-2016].

[59] A. van Kesteren and L. Hunt. Selectors api level 1. https://www.w3.org/TR/selectors-api/, 2013. [Online; accessed 10-May-2016].

[60] V. Vasilyev. fingerprintjs2. https://github.com/Valve, 2015.

[61] World Wide Web Consortium (W3C). All standards and drafts. http://www.w3.org/TR/, 2015. [Online; accessed 16-October-2015].

[62] F. Zaninotto. Gremlins.js. https://github.com/marmelab/gremlins.js, 2016.