Most Websites Don't Need to Vibrate: A Cost–Benefit Approach to Improving Browser Security

***Abstract*—New features are frequently added to web browsers. These features bring the benefit of enabling new types of web applications, but at the cost of potentially harming the security and privacy guarantees of the browser. Security sensitive software must strike a balance between the two, as Mozilla recently did when they removed the *Battery API* from Firefox. The API was found being used to track anonymous users online, and Mozilla decided that the API's cost outweighed its benefit for users.**

This work applies a comprehensive cost–benefit analysis to all 74 Web API standards in Firefox. We model a Web API standard's benefit as the number of websites that use it for core functionality, and it's cost as a combination of the number of CVEs, lines of code, and academic attacks related to the standard. We find that many browser standards provide little benefit to users, but pose significant risk to users' security and privacy.

We introduce a configurable browser extension that blocks these low-benefit, high-cost features in the browser, allowing users to select a better balance between browser functionality and security. We evaluated our extension with two hardened browser configurations, and found that blocking 15 of the 74 standards avoids 52.0% of code paths related to previous CVEs, and 50.0% of implementation code identified by our metric, without affecting the functionality of 94.7% of measured websites.

I. Introduction

Modern web browsers add new features at a terrific pace. These new capabilities go far beyond the hypertext documents the browser was originally designed for. To name just a few examples, web browsers currently allow websites to detect changes in ambient light levels [59], do complex audio synthesis [10], enforce digital rights management systems [21], cause vibrations in enabled devices [33], and create peer to peer networks [7], all without relying on third party plugins or external software.

However, as fast as the browser gains new capabilities, new attacks are developed that leverage these capabilities. For instance, *WebGL* and *Canvas* allowed Mowery and Shacham to create resilient, high entropy browser fingerprints without using local storage [46]. Even more dangerous is the recent HEIST attack; Vanhoef and Van Goethem were able to conduct CRIME-style SSL/TLS attacks within a victim's browser [62] using the *Fetch* and *Resource Timing* APIs. We present a more extensive overview of academic attacks and the JavaScript APIs that enable them in Section V-B1.

Concern about the costs some features were posing to user security lead Mozilla to remove the *Battery Status API* from Firefox.[1] In discussion,[2] Mozilla developers judged it to be low benefit (i.e. few websites using it as intended) and high cost (the standard was being used to fingerprint users [22]). Google's BoringSSL [25] takes a similar approach, removing features from OpenSSL that are of low benefit and high cost to users.

New features enable new potential use cases in the browser, but in practice such use cases are rare. A large portion of the web still provides its primary value through information dissemination rather than by providing access to web applications. These websites can still deliver their benefits to users through access to only a limited number of these JavaScript APIs.

An understanding of the benefits and risks of each JavaScript feature is necessary to make sound decisions about which features are actually needed to use the modern web experience. With this knowledge, a set of highly beneficial features can be exposed by default to all websites, while only trusted sites that need additional features are given the ability to access the full set of capabilities in the browser, thus enforcing the principle of least privilege on the Web API.

This work applies a systematic cost-benefit analysis to the entire standardized Web API as implemented in the Firefox web browser. We present a method to quantitatively evaluate both the **cost** of a feature (the added security risk of making a feature available) and the **benefit** of a feature (the number of websites that require the feature to function properly).

Using these cost-benefit measurements, we identify two hardened browser configurations by identifying high-cost standards that could be blocked in the browser without affecting the browsing experience on most websites. We present a Firefox browser extension that enforces these hardened browser configurations, and compare the usability of these hardened browser configurations against NoScript and the Tor Browser Bundle (TBB), other popular browser-security tools. We find that our hardened browser configurations offer substantial security benefits for users, while breaking fewer websites than either NoScript or the default configuration of the TBB during our evaluation on both the 200 most popular sites in the Alexa 10k, and a random sampling of the rest of the Alexa 10k.

Finally, we build our browser-hardening extension to be highly configurable, so that standards can be blocked or allowed

[1] https://www.fxsitecompat.com/en-CA/docs/2016/battery-status-api-has-been-removed/

[2] https://groups.google.com/forum/#!msg/mozilla.dev.platform/5U8NHoUY-1k/9ybyzQIYCAAJ

on a per-site basis, and the set of standards blocked can be updated to reflect changes in the popularity or security costs of each standard.

This work presents the following technical contributions:

- **ES6 Proxy based feature firewall. (Section III)** We leverage the ES6 proxy object to build a feature firewall which dynamically disables JavaScript API features *without* breaking most code that expects those features to exist.
- **Code complexity as cost. (Section IV-D2)** We perform a static analysis of the Firefox codebase to identify and count lines of code exclusively used to enable each web standard. We find a moderate, statistically significant relationship between this code complexity metric and previously discovered vulnerabilities attributed to these standards.
- **Contextual protection extension. (Section VII)** We package the feature firewall in an open source Firefox extension that allows the deployment of pre-defined conservative and aggressive feature blocking policies. The extension is highly customizable, with a user experience similar to popular ad blocking software, including blocked API notifications, streamlined reload and retry, and customizable site whitelisting. Furthermore, Section VI outlines our process of developing and evaluating the predefined conservative and aggressive policies. We find that our method can block access to large portions of the Web API while impacting far fewer websites than popular security and privacy focused browser modifications like NoScript or the Tor Browser Bundle.

Further, these tools enable an analysis of the Firefox source code with the intention of determining the costs and benefits of each Web API standard, yielding the following additional contributions.

Understanding feature benefit (Section V-A). We define the benefit of enabling a feature as the number of websites which require the feature to function correctly, as perceived by the user. To quantify this amount, we develop a methodology for determining the necessity of a given feature in a casual browsing scenario. We show that two humans using simple rules to independently gauge the functionality of a website under different levels of browser functionality can have high agreement (97%), and thus can be used to model the benefit of a given feature. We use this methodology to investigate the necessity of 74 different features in 1,684 different paired tests undertaken across 500 hours of human effort.

Understanding feature cost. (Section V-B) We define the cost of enabling a feature as the number of vulnerabilities in the newly exposed attack surface. Because this value is unknowable, we model cost in three ways: first, we model security cost as a function of the number of previously reported CVEs in a feature. We use previous vulnerabilities with the intuition that features which are difficult to code correctly are more likely to have further undiscovered vulnerabilities.

Second, we model security cost as the number of attacks introduced in academic papers which have been enabled by each Web API standard. While we do not use this as a quantified metric, it does shed light on which features are commonly leveraged to subvert users' security and privacy in novel ways.

Third, we model security cost as a function of code complexity. We attribute entry points in Firefox's C++ codebase to JavaScript exposed features, and then quantify complexity as the number of lines of code used solely to implement access to each feature. We find a moderate, statistically significant correlation between this exclusive-use lines of code metric and previously discovered vulnerabilities, which corroborates previous research in vulnerability prediction [54].

II. Related Work

In this section we discuss the current state of browser features, as well as existing user level security defenses.

A. Browser Feature Inclusion

Browsers compete on performance, security, and compatibility. This final point introduces two security related challenges: first, vendors are very wary of removing features from the browser, even if they are used by a very small fraction of all websites [2, 4]. Second, because the web is evolving and even competing with native applications (especially on mobile devices), browser vendors are incentivized to continue to add new features to the web browser and not remove old features. Browsers using the same code base across all devices, including mobile, browser OS devices (e.g., Google Chromebooks), and traditional PCs also increases the amount of code in the browser. The addition of support for this variety of devices means that JavaScript features that support hardware features (webcams, rotation sensors, vibration motors, or ambient light sensors, etc. [33, 34, 36, 59]) are included in the browser for all devices, regardless of whether they include such hardware. All of this has resulted in a massive growth of the amount of code in the browser, with Firefox currently containing over 13 million lines of code, and Chrome containing over 14 million [15].

B. Client Side Browser Defenses

There are variety of techniques which "harden" the browser against attacks via limiting what JavaScript is allowed to run within the browser. These defenses can be split into two categories: those configured by the user, and those configured by the website author. Our method is in the former category, allowing the user to make decisions about which features to enable when.

In the user configured category, both Adblock and NoScript prevent JavaScript from running based on the site serving it. While its primary function is to block ads for aesthetic purposes, Adblock [1] can also prevent infection by malware being served in those ads [16, 52]. Adblock blocks JavaScript features by preventing the loading of resources from certain domains, rather than disabling specific functionality. NoScript [39] prevents JavaScript on an all-or-nothing basis, decided based on its origin. Its default for unknown origins is to allow nothing,

rendering a large swath of the web unusable. It is worth noting that NoScript defaults to whitelisting a number of websites, which has resulted in a proof of concept exploit via purchasing expired whitelisted domains [17]. Beyond these popular tools, IceShield [29] dynamically detects suspicious JavaScript calls within the browser, and modifies the DOM to prevent attacks.

The Tor Browser [20] disables by default or prompts the user before using a number of features. Regarding JavaScript, they disable SharedWorkers [6], and prompt before using calls from HTML5 Canvas, the GamePad API, WebGL, the Battery API, and the Sensor API [53]. These particular features are disabled because they enable techniques which violate the Tor Browser's security and privacy goals.

On the website author side, Content Security Policy allows limiting of the functionality of a website, but rather than allowing browser users to decide what will be run, CSP allows web developers to constrain code on their own sites so that potential attack code cannot access functionality deemed unnecessary or dangerous [57]. Conscript is another client-side implementation which allows a hosting page to specify policies for any third-party scripts it includes [40]. There are also a number of technologies selected by the website author but enforced on the client side, including Google Caja [41] and GATEKEEPER [28].

There are existing models for enforcing policies to limit functionality outside of the web browser as well. Mobile applications use a richer permission model where permission to use certain features is asked of the user at either install or run-time [3, 13].

III. Graceful Feature Degradation in the Browser

Core to both our measurements and the browser hardening extension is the ability to disable specific features from the browser's JavaScript environment. Here we present a technique for removing access to features implemented in the browser while minimizing collateral damage due to code that expects those features to be available.

A. Web API / W3C standards

When visiting and displaying websites, browsers build a tree-based model of the document. This tree, along with the methods and properties the browser provides to allow site authors to interact with the browser and the tree, are collectively known as the DOM (document object model), or the Web API.

The browser makes much of its functionality available to websites through a single, global object, called `window`. Almost all JavaScript accessible browser functionality is implemented as a property or method on this global object. The set of properties, functions, and methods available in the DOM is standardized using Interface Description Language documents. Browser vendors implement these standards in their browsers.

For the purposes of this paper, we define a **feature** as an individual JavaScript method or property available in the browser, and a **Web API standard** (or just **standard**) as a collection of features collected into a single document and published together. Each standard generally contains features that are intended to be used together to enable a common functionality (such as WebGL graphics manipulation, geolocation services, or cryptographic services).

B. Removing Features from the DOM

Each webpage and iframe gets its own global window object. Changes made to the global object are shared across all scripts on the same page, but not between pages. Furthermore, changes made to this global object are seen immediately by all other script running in the page. If one script deletes or overwrites the `window.alert` function, for example, no other scripts on the page will be able to use the `alert` function, and there is no way they can recover it.

As a result, code executed earlier can arbitrarily modify the browser environment seen by code executed later. Since code run by browser extensions can run before any scripts included by the page, extensions can modify the browser environment for all code executed in any page. The challenge in removing a feature from the browser environment is not to *just* prevent pages from reaching the feature, but to do so *in way that still allows the rest of the code on the page to execute without introducing errors*.

For example, to disable the `getElementsByTagName` feature, one could simply remove the `getElementsByTagName` method from the `window.document` object. However, this will result in fatal errors if future code attempts to call that now-removed method.

Consider the code in Figure 1: removing the `window.document.getElementsByTagName` method will cause an error on line one, as the site would be trying to call the now-missing property as if were a function. Replacing `getElementsByTagName` with a new, empty function would solve the problem on line one, but would cause an error on line two unless the function returned an array of at least length five. Even after accounting for that result, one would need to expect that the `setAttribute` method was defined on the fourth element in that array. One could further imagine that other code on the page may be predicated on other properties of that return value, and fail when those are not true.

```
1 var ps = document.getElementsByTagName("p");
2 var fifthP = ps[4];
3 fifthP.setAttribute("style", "color: red");
4 alert("Success!");
```

Fig. 1: Trivial JavaScript code example, changing the color of the text in a paragraph.

C. ES6 Proxy Configuration

Our technique solves this problem through a specially constructed version of the `Proxy` object. The `Proxy` object

is a recent addition to the JavaScript language[3] that allows one object to either intercept operations or pass them on to another object. Relevant to this work, proxy objects also allow code to trap on general language-operations. Proxies can register generic handlers that fire when the proxy is called like a function, indexed into like an array, has its properties accessed like an object, and operated on in other ways.

We take advantage of the `Proxy` object's versatility by creating a proxy object, registering callback functions for *all* possible JavaScript operations, and having those callback functions return a reference to the same proxy object. We also handle cases where Web API properties and functions return scalar values (instead of functions, arrays or higher order objects), by programming the proxy to evaluate to `0`, empty string, or `undefined`, depending on the context. Thus configured, the proxy object can validly take on the semantics of any variable in any JavaScript program.

D. Proxy-Based Approach

Using our configured proxy, we can now solve the problems described in III-B. By replacing `getElementsByTagName` with our proxy, the code in Figure 1 will execute cleanly and the alert dialog on line four will successfully appear. On line one, the proxy object's function handler will execute, resulting in the proxy being stored in the `ps` variable. On line two, the proxy's `get` handler will execute, which also returns the proxy, resulting in the proxy again being stored in `fifthP`. Calling the `setAttribute` method on line three causes the proxy object to be called twice, first because of looking up the `setAttribute`, and then because of the result of that look up being called as a function. The end result is that the code executes correctly, but without accessing any browser functionality beyond the core JavaScript language.

The complete proxy-based approach to graceful degradation can be found in the source code of our browser extension[4].

Most state changing features in the browser are implemented through methods, which we block or record using the above described method. A small number of other features are implemented through property sets on singleton objects in the browser (e.g. assigning a string to `document.location` redirects the browser to the URL represented by the string, or writing to `document.title` changes the name of the page displayed in the browser). In these cases, we interpose on the property by assigning a new "set" function for the property on the singleton using "Object.defineProperty".

For property sets on non-singleton objects, the above approach would not work because objects created during a page's execution do not yet exist when the extension is modifying the DOM. We instead interpose on all methods that return references to these non-singleton objects. Interposing at this level allows us to prevent access to those properties by returning specially configured `Proxy` objects rather than the original non-singleton object.

[3]The Proxy object is currently available in the Edge, Firefox, Chrome, and Android browsers.

[4]URL Redacted for review.

IV. Methodology

In this section we describe how we determine the costs and benefits of Web API standards. We measure the benefit of each standard using the described feature degradation technique for each standard of features, browsing sites that use those feature, and observing the result. We measure the cost of enabling each standard in three ways: as a function of the prior research identifying security or privacy issues with the standard, the number and severity of associated historical CVEs, and as a metric based on the LoC needed solely to implement that standard.

A. Measuring by Standard

We extracted the 1,392 Web API features implemented in Firefox 43.0.4, which we then categorized into 74 Web API standards, using the same technique as in [55]. Using the features listed in the W3C's (and related standards organizations) publications, we categorized `Console.prototype.log` and `Console.prototype.timeline` with the *Console API*, `SVGFilterElement.apply` and `SVGNumberList.prototype.getItem` with the *SVG* standard, and so forth, for each of the 1,392 features. We use these 74 standards as our unit of Web API measurement.

We use standards as our unit of measurement for two reasons. First, focusing on 74 standards leads to less of a combinatorial explosion when testing different subsets of Web API functionality. Secondly, as standards are organized around high level features of the browser that often have one cohesive purpose, for instance the *Web Crypto* standard or the *Web Audio API*, being able to reason about what features a website might need is useful for communicating with users who might be interested in blocking (or allowing) such features to run as part of a given website.

B. Determining When A Website "Needs" A Feature

Core to our benefit metric is determining whether a given website needs a browser feature to function. When a site does not need a feature, enabling the feature on the site provides little benefit to browser users.

Determining whether a website actually needs a feature to function is surprisingly difficult. On one end of the spectrum, when a website never uses a feature, the site trivially does not need to feature to run correctly. Previous work [55] shows that most features in the browser fall in this category, and are rarely used on the open web.

However, a website may use a feature, but not need it to carry out the site's core functionality. With the feature removed, the website will still function correctly and be fully usable. For example, a blog may wish to use the *Canvas* standard to invisibly fingerprint the visitor. But if a visitor's browser does not support the *Canvas* standard, the visitor will still be able to interact with the blog as if the standard was enabled (though the invisible fingerprinting attempt will fail).

This measure of feature "need" is intentionally focused on the *the perspective of the browser user*. The usefulness of a feature

to a website author is not considered, except for indirectly through the ability of the site author to deliver user-experience to the browser user. If a site's functionality is altered (e.g. tracking code is broken, or the ability to A/B test is hampered) in a way the user cannot perceive, then we consider this feature as not being needed from the perspective of the browser user, and thus not needed for the site.

With this insight in mind, we developed a methodology for evaluating the functionality of a given website. We instructed two undergraduate workers to each visit the same website twice in a row. The first visit is used as a control, and was conducted in an unmodified Firefox browser. The worker was instructed to perform as many different actions on the page as possible within one minute. (This is in keeping with the average dwell time a user spends on a website, which is slightly under a minute [38].) On a news site this would mean skimming articles or watching videos, on e-commerce sites searching for products, adding them to the cart and beginning the checkout process, on sites advertising products reading or watching informational material and trying any live demos available, etc.

The second visit is used to measure the effect of a specific treatment on the browsing experience. The worker visits the same page a second time, with all of the features in a Web API standard disabled. For another minute, the worker attempts to perform the same actions they did during the first visit. They then score the functionality of the site into one of three categories. The first category represents an unchanged experience with no perceptible difference between the control and treatment conditions. The second category corresponds to a different browsing experience, but one in which the worker was still able to complete the same tasks as during the first visit. Finally, the third category corresponds to a failure to complete the tasks that were completed during the control visit. We refer to these as categories 1, 2, and 3. The worker records the category, and for categories 2 and 3, also includes a quick note as to why the website did not work as expected. We define a site as broken when the user cannot accomplish their intended task (i.e., the visit was coded as a 3).

This approach is inherently subjective. To account for this, we had both student workers browse the same site independently, and record their score without knowledge of the other's experience. Our workers averaged a 96.74% agreement ratio. This high agreement supports the hypothesis that the workers were able to successfully gauge whether particular functionality was necessary to the goals of a user performing casual web browsing.

C. Determining Per-Standard Benefit

We determined the benefit of each of the 74 measured standards in four steps.

First, we select a set of websites to represent the internet as a whole. This work considers the top 10,000 most popular websites on the Alexa rankings as representative of the web in general, as of July 1, 2015, when this work began.

Second, for each standard, we randomly sample 40 sites from the Alexa 10k that use the standard, as identified by [55]. Where there were less than 40 sites in the Alexa 10k that used the standard, we selected all of the sites known to use that standard. We treat these randomly sampled 40 as representative of all sites using the standard.

Third, we used the technique described in Section III to create multiple browser configurations, each with one standard disabled. This yielded 75 different browser configurations (one configuration with each standard disabled, and one "control" case with all standards enabled).

Fourth, we performed the manual testing described in Section IV-B. We carried out the above process twice for each of the 1679 sites tested for this purpose. By carrying out the above process for all 74 standards, we were able to measure the **site break rate** for each Web API standard, defined as the percentage of times we observed a site break during our paired tests with the featured disabled, multiplied by how frequently the standard is used in the Alexa 10k. We then define the benefit of a standard as a function of its site break rate; the more sites break when a standard is disabled, the more useful the standard is to a browser user. The results of this measurement are discussed in Section V.

D. Determining Per-Standard Cost

We measure the security cost of enabling a Web API standard in three ways.

First, we measure the cost of enabling a Web API standard in Firefox as a function of CVEs that have been reported against the standard in the past. We take past CVEs as an indicator of present risk for three reasons. First, areas of code that have multiple past CVEs suggest that there is something about the problem domain addressed by this code that is difficult to code securely, suggesting that these code areas deserve heightened scrutiny (and carry additional risk). Second, prior research [51, 66] suggest that bugs fixes often introduce nearly as many bugs as they address, suggesting that code that has been previously patched for CVEs carries heightened risk for future CVEs. Third, recent notable industry practices suggest that project maintainers sometimes believe that code that has had multiple security vulnerabilities should be treated greater caution (and that shedding the risky code is safer than continually patching it) [25].

Second, we measure the cost of including a Web API standard by the amount of related academic work documenting security and privacy issues in a standard. We searched for attacks leveraging each Web API standard in security conferences and journals over the last five years.

Third, we measure the cost of including a Web API standard by the number of lines of code needed solely to implement the standard in Firefox, as code complexity (measured through number of lines of code in function definitions) has been shown to have moderate predictive power for discovering where vulnerabilities will happen within the Firefox codebase [54].

1) CVEs: We determined the number of CVEs previously associated with each Web API standard through the following steps:

First, we searched the MITRE CVE database for all references to Firefox in CVEs issued in 2010 or later, resulting in 1,554 CVE records.

We then reviewed each CVE and discarded 41 CVEs that were predominantly about other pieces of software being used in or with Firefox (such as the Adobe Flash Player plugin [43], or vulnerabilities in web sites that are exploitable through Firefox [44]).

Next, we examined each of the remaining CVEs to determine if they documented vulnerabilities in the implementation of one of the 74 considered Web API standards, or in some other part of the browser, such as the layout engine, the JavaScript runtime, or networking libraries. We identified 175 CVEs describing vulnerabilities in Firefox's implementation of 39 standards. 13 CVEs documented vulnerabilities affecting multiple standards.

We identified which Web API standard a CVE related to by reading the text description of each CVE. We were able to attribute CVEs to individual standards in the following ways:

- 117 (66.9%) CVEs explicitly named a Web API standard.
- 32 (18.3%) CVEs named a JavaScript method, structure or interface) that we tied to a larger standard.
- 21 (12%) CVEs named a C++ class or method that we tie to the implementation of Web API standard, using the methodology described in IV-D2.
- 5 (2.8%) CVEs named browser functionality defined by a Web API standard (e.x. several CVEs described vulnerabilities in Firefox's handling of drag-and-drop events, which are covered by the HTML standard [63]).

When associating CVEs with Web API standards, we were careful to distinguish between CVEs associated with DOM-level functionality and those associated with more core functionality. This was done to narrowly measure the cost of *only* the DOM implementation of the standard. For example, the SVG Web API standard [18] allows site authors to use JavaScript to dynamically manipulate SVG documents embedded in websites. We counted CVEs like `CVE-2011-2363` [42], a "Use-after-free vulnerability" in Firefox's implementation of JavaScript DOM API for manipulating SVG documents, as part of the cost of including the SVG Web API standard in Firefox. We did not consider CVEs relating to other aspects of SVGs handing in our Web API standard costs. `CVE-2015-0818` [45], a privilege escalation bug in Firefox's SVG handling, is an example of a CVE we did not associate with the SVG Web API standard, as it was not part of the DOM.

2) Implementation Complexity: We use the Firefox source to generate lower-bound approximations for how complex the implementation is for each of the 74 measured standards, with complexity being measured as significant lines of C/C++ code. We consider standards with more complex implementations as having a greater cost to the security of the browser than those with simpler implementations.

We model complexity as the number of lines of C/C++ code used *only* to support JavaScript based access to that specific feature. We henceforth refer to this metric as Exclusive Lines of Code, or **ELoC**. We compute the ELoC for each Web API standard in three steps.

We generate a call graph for the Firefox codebase using a modified version of Mozilla's DXR tool [47]. DXR uses a clang compiler plugin to produce an annotated version of the source code through a web app.[5] Using the database underlying this web app, we constructed a static call graph of the core Firefox browser, allowing us to determine which functions call which other functions, where functions are referenced, etc. We also modified DXR to record the number of lines of code for each function.

Next, we associated each Web API standard with its unique entry points in the call graph. Each property, method or interface defined by a Web API standard has two categories of underlying C++ code in Firefox code. There is **implementation code** (hand written code that provides Web API standard's functionality), and **binding code** (programmatically generated C++ code only called by the JavaScript runtime). Binding code is generated at build time from WebIDL documents, an interface description language that defines each Web API standard's JavaScript API endpoints. By mapping each feature in each Web IDL document to a Web API standard, we are able to associate each binding code function with a Web API standard.

Once we know which Web API standard each binding function is associated with, we use a recursive graph algorithm to identify implementation code associated with each standard. We illustrate an example of applying this approach in Figure 2. In step 1, we programmatically extract the standard's definitions for its binding functions, as we do here using a simplified version of the *Battery API*. In step 2, we locate these generated binding functions in the Firefox call graph (denoted by blue nodes). By following the call graph, we identify implementation functions that are called by the *Battery API's* binding functions, denoted by pink nodes. (step 3). If these pink nodes have no incoming edges other than binding functions, we know they are solely in the code base because of the Web API standard associated with those binding functions.

The first iteration of the algorithm identifies two functions, `Charging` and `DischargingTime`, as being solely related to the *Battery API* standard, since no other code within the Firefox codebase contains a reference or call to those functions. The second iteration of the pruning process identifies the `ChargingTime` function as also guaranteed to be solely related to the *Battery API* standard's implementation, since it is only called by functions we know to be solely part of the *Battery API*'s implementation. Thus, the lines implementing all three of these pink implementing functions are used to compute the ELoC metric for the *Battery API*.

3) Third Party Libraries: This technique gives a highly accurate, lower bound measurement of lines of code *in the Firefox source* included only to implement a single Web API standard. It does not include code from third-party libraries,

[5] An example of the DXR interface is available at https://dxr.mozilla.org/mozilla-central/source/.

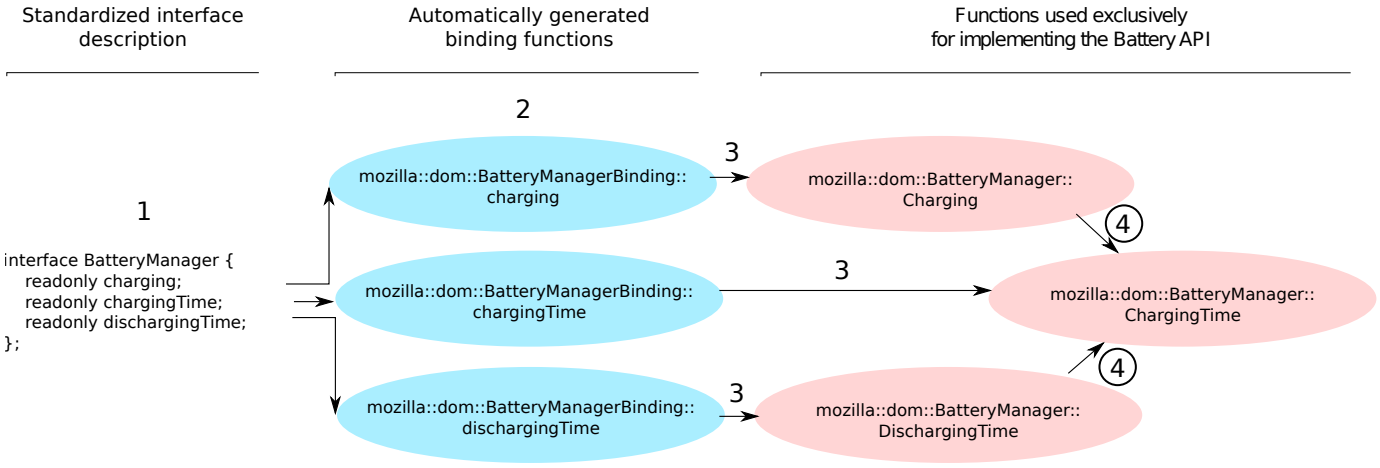

Fig. 2: An example of applying the graph pruning algorithm to a simplified version of the *Battery API*.

which are compiled as a separate step in the Firefox build process, and thus excluded from DXR's call-graph.

To better understand their use, we investigated how third party libraries are used in the Firefox build process. In nearly all cases, the referenced third party libraries are used in multiples places in the Firefox codebase and cannot be uniquely attributed to any single standard. These third-party libraries include code to handle multimedia (e.x. *libav* and *libpng*), advanced graphics operations (e.x. *skia* and *cairo*), or handling database operations (*SQLite3*). Such libraries are used across multiple standards, as well as in the non-Web API portions of Firefox, and thus are not relevant to our per-standard ELoC counts.

One sole exception is the *WebRTC* standard, which makes use of a large amount of third-party code that is not used elsewhere in the codebase. The *WebRTC* standard uses the *libjingle* and *Chromium WebRTC* libraries, which make up 500k significant lines of code. The result is that our ELoC metric dramatically undercounts the complexity *WebRTC* adds to Firefox.

While this undercount is large, it is ultimately not significant to our goal of identifying high-cost, low-benefit standards, as the high number of vulnerabilities in the standard (as found in CVEs) and comparatively high ELoC metric already flag the standard as being high-cost. All other standards either do not rely on a third partly library, or use a third party library that is shared with other parts of the Firefox base.

V. Measured Cost and Benefit

This section presents the results of the methodology discussed in Section IV. The section proceeds by first describing the benefit of each Web API standard, follows with the cost of each standard, and ends by measuring the correlation between our different cost metrics.

A. Per-Standard Benefit

As explained in Section IV-C, our workers conducted up to 40 measurements of websites in the Alexa 10k known to use each specific Web API standard. If a standard was observed being used fewer than 40 times within the Alexa 10k, all sites using that standard were measured. In total, we did two measurements of 1,684 (website, disabled feature) tuples, one by each worker.

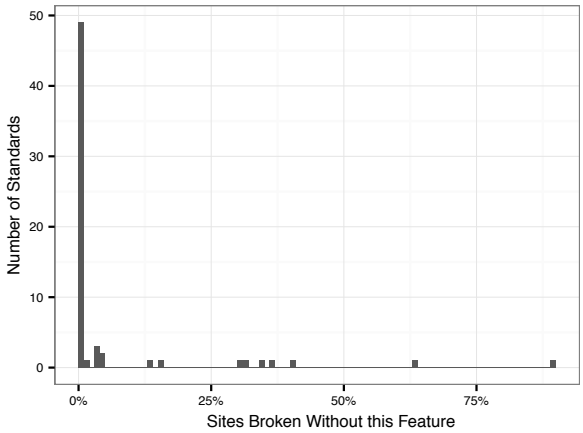

Fig. 3: A histogram giving the number of standards binned by the percentage of sites that broke when removing the standard.

Figure 3 gives a histogram of the break rates for each of the 74 standards measured in this work. As the graph shows, removing over 60% of the measured standards resulted in no noticeable effect on the user's experience on the page.

In some cases, such as the *WebVTT* standard (which allows document authors to synchronize text changes with media playing on the page), this 0% break rate is because the standard was never used in our casual browsing scenario. Other standards, such as the *Beacon* standard (which allows content authors to trigger code execution when a user browses away from a website), had a 0% break rate because their functionality is "behind the scenes", and not intended to be noticeable to the user in the first place.

Other standards caused a large number of sites to break when removed from the browser. Disabling access to the *DOM*

1 standard (which provides basic functionality for modifying the text and appearance of a document) broke an estimated 69.05% of the web.

A listing of the site break rate for all 74 standards is provided in the appendix in Table IV.

B. Per-Standard Cost

As described in Section IV-D, we measure the cost of a Web API standard being available in the browser in three ways: first by related research documenting security and privacy attacks that leverage the standard (Section V-B1), second by the number of historical CVEs reported against the standard since 2010 (Section IV-D1), and third with a lower bound estimate of the number of ELoC needed to implement the standard in the browser (Section IV-D2).

1) Security Costs - Attacks from Related Research: We searched through the last five years of work published at major research conferences and journals for research documenting browser weaknesses related to Web API standards. These papers either explicitly identify either Web API standards, or features or functionality that belong to a Web API standard. In each case the standard was either necessary for the attack to succeed, or was used to make the attack faster or more reliable.

The most frequently cited standard was the *High Resolution Time Level 2* [5] standard, which provides highly accurate, millisecond-resolution timers. Seven papers published since 2013 leverage the standard to break the isolation protections provided by the browser, such as learning information about the environment the browser is running in [27, 30, 50], learning information about other open browser windows [12, 27, 35], and gaining identifying information from other domains [60].

Other implicated standards include the *Canvas* standard, which was identified by researchers as allowing attackers to persistently track users across websites [8], learn about the browser's execution environment [30] or obtain information from other browsing windows [35], and the *Media Capture and Streams* standard, which was used by researchers to perform "cross-site request forgery, history sniffing, and information stealing" attacks [58].

In total we identified 20 papers leveraging 23 standards to attack the privacy and security protections of the web browser. Citations for these papers are included in Table IV.

2) Security Costs - CVEs: Vulnerability reports are not evenly distributed across browser standards. Figures 4 and 5 present this comparison of standard benefit (measured by the number of sites that require the standard to function) on the y-axis, and the number of CVEs historically associated with the standard on the x-axis.

Points in the upper-left of the graph depict standards that are high benefit, low cost, i.e. standards that are frequently required on the web but have rarely (or never) been implicated in CVEs. For example, consider the *Document Object Model (DOM) Level 2 Events Specification* standard, denoted by **DOM2-E** in Figure 5. This standard defines how website authors can associate functionality with page events such as button clicks and mouse movement. This standard is highly beneficial to

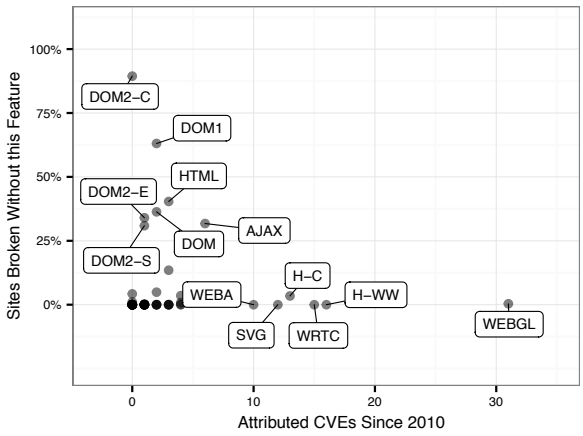

Fig. 4: A scatter plot showing the number of CVEs filed against each standard since 2010, by how many sites in the Alexa 10k break when the standard is removed.

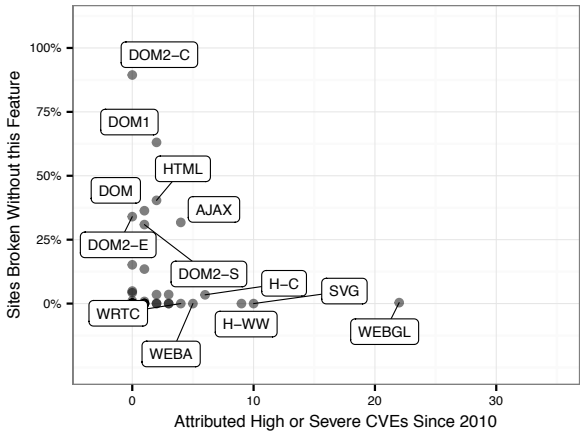

Fig. 5: A scatter plot showing the number of "high" or "severe" CVEs filed against each standard since 2010, by how many sites in the Alexa 10k break when the standard is removed.

browser users, being required by 34% of pages to function correctly. Enabling the standard comes with little risk to web users, being associated with zero CVEs since 2010.

Standards in the lower-right section of the graph, by contrast, are low benefit, high cost standards, when using historical CVE counts as an estimate of security cost. The *WebGL Specification* standard, denoted by **WEBGL** in Figure 5, is an example of such a low-benefit, high-cost standard. The standard allows websites to take advantage of graphics hardware on the browsing device for 3D graphics and other advanced image generation. The standard is needed for less than 1% of web sites in the Alexa 10k to function correctly, but is implicated in 22 high or severe CVEs since 2010. How infrequently this standard is needed on the web, compared with how often the standard has previously been the cause of security vulnerabilities, suggests that the standard poses a high security risk to users going forward, with little attenuating benefit.

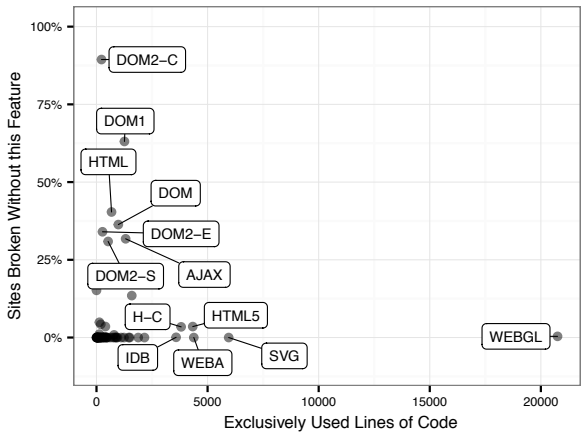

Fig. 6: A scatter plot showing the LOC measured to implement each standard, by how many sites in the Alexa 10k break when the standard is removed.

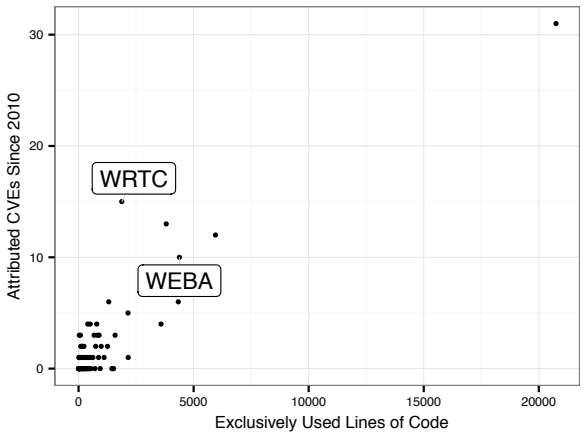

Fig. 7: The number of CVEs reported against each standard as a function of the found LoC metric for that standard.

As Figures 4 and 5 show, some standards have historically put users at much greater risk than others. Given that for many of these standards the risk has come with little benefit to users, these standards are good candidates for disabling when visiting untrusted websites.

3) Security Costs - Implementation Complexity: We further found that the cost of implementing standards in the browser are not equal, and that some standards have far more complex implementations than others (with complexity measured as the ELoC uniquely needed to implement a given standard). Figure 6 presents a comparison of standard benefit (again measured by the number of sites that require the standard to function) and the exclusive lines of code needed to implement the standard, using the method described in section IV-D2.

Points in the upper-left of Figure 6 depict standards that are frequently needed on the web for sites for function correctly, but which have relatively non-complex implementations. One example of such a standard is the *Document Object Model (DOM) Level 2 Core Specification* standard, denoted by **DOM2-C**. This standard provides extensions the browser's basic document modification methods, most popularly, the `Document.prototype.createDocumentFragment` method, which allows websites to quickly create and append sub-documents to the current website. This method is needed for 89% of websites to function correctly, suggesting it is highly beneficial to web users to have it enabled in their browser. The standard comes with a low security cost to users as well; our technique identifies only 225 exclusive lines of code that are in the codebase solely to enable this standard.

Points in the lower-right of the figure depict standards that provide infrequent benefit to browser users, but which are responsible for a great deal of complexity in the browser's code base. The *Scalable Vector Graphics (SVG) 1.1 (Second Edition)* standard, denoted by **SVG**, is an example of such a high-cost, low-benefit standard. The standards allows website authors to dynamically create and interact with embedded SVG documents through JavaScript. The standard is required for core functionality in approximately 0% of websites on the Alexa 10k, while adding a large amount of complexity to the browser's code base (at least 5,949 exclusive lines of code, more than our technique identified for any other standard).

4) Cost Metric Correlation: Finally, we measured whether there was a correlation between the the number of CVEs associated with the Web API standard since 2010 and the number of exclusive lines of code needed in the code base to implement the standard. Figure 7 shows this comparison, with the complexity cost metric on the X-axis, and the CVE cost metric on the Y-axis. We found a statistically significant, moderate correlation between these cost metrics using Kendall's tau ($\tau = 0.42$, $p < 10^{-6}$). Kendall's tau rank correlation coefficient is resilient to outliers and is non-parametric. It has also been used in previous investigations of the power of metrics like rate of empty commit messages or prior bug reports in predicting the appearance of future software bugs [14].

C. Threats to Validity

The main threat to validity in this experiment is the accuracy of our human-executed casual browsing scenario. With respect to internal validity, the high agreement between the two users performing tasks on the same sites lends credence to the claim that the users were able to successfully exercise most or all of the functionality that a casual browser might encounter. The students who worked on this project spent over 500 hours combined performing these casual browsing tasks and recording their results, and while they were completely separated while actively browsing, they spent a good deal of time comparing notes about how to fully exercise the functionality of a website within the 70 second time window for each site.

External validity, the extent to which our results can be generalized, is also a concern. Firstly, visiting a website for 70 or fewer seconds encapsulates 80% of all web page visits according to [38], thus accurately representing a majority of web browsing activity, especially when visiting untrusted websites. Furthermore, while our experiment does not evaluate the

JavaScript functionality that is only available to authenticated users, we posit that protection against unknown sites—the content aggregators, pop-up ads, or occasionally consulted websites that a user does not interact with enough to trust—are precisely the sites with which the user should exercise the most caution.

VI. Hardened Browser Configurations

In this section we describe two hardened browser configurations we created and evaluated, using a cost-benefit approach to deciding which Web API standards to enable. This section proceeds by first describing the two hardened browser configurations, and then comparing the usability of these configurations against other browser hardening approaches.

A. Selecting Configurations

To evaluate the utility and usability of our fine grained, standards-focused approach to browser hardening, we created two hardened browser configurations. Our **conservative** configuration focuses on removing features that are infrequently needed by websites to function, while our **aggressive** configuration focuses on removing attack surface from the browser, even when that necessitates breaking more websites.

We selected these profiles based on the data discussed in section V, related previous work on how often standards are needed by websites [55], and prioritizing not affecting the functionality of the most popular sites on the web. We note that these are just two possible configurations, and that users (or trusted curators, IT administrators, or other sources) could use this method to find the security / usability tradeoff that best fit their needs.

1) Standard Selection: Table I lists the standards that we blocked for the conservative and aggressive hardened browser configurations. These policies were heuristically selected using benefit measurements from Section V, trying to minimize both the number of broken websites and the attack surface of the browser. The conservative configuration is fitting for users who desire more security than is typical of a commodity web browser, and are tolerant of a slight loss of functionality. The aggressive configuration is fitting for highly security sensitive environments, where users are willing to accept breaking a higher percentage of websites in order to gain further security.

2) Evaluation: Having selected the two browser configurations, we then evaluated their usability and the security gains they provided. Table II shows the results of this evaluation. As expected, blocking more standards resulted in a more secure browser, but at some cost to usability (measured by the number of broken sites).

Our evolution was carried out similarly to the per-standard measurement technique described in Section IV-C. First we created two sets of test sites, **popular** sites (the 200 most popular sites in the Alexa 10k that are in English and not pornographic) and **less popular sites** (a random sampling of sites from the Alexa 10k that are rank 201 or lower). This yielded 175 test sites in the popular category, and 155 in the less popular category.

Standard	Conservative	Aggressive
Beacon	X	X
DOM Parsing and Serialization	X	X
Fullscreen API	X	X
High Resolution Time Level 2	X	X
HTML: Web Sockets	X	X
HTML: Channel Messaging	X	X
HTML: Web Workers	X	X
Indexed Database API	X	X
Performance Timeline Level 2	X	X
Resource Timing	X	X
Scalable Vector Graphics 1.1	X	X
UI Events Specification	X	X
Web Audio API	X	X
WebGL Specification	X	X
Ambient Light Sensor API		X
Battery Status API		X
CSS Conditional Rules Module Level 3		X
CSS Font Loading Module Level 3		X
CSSOM View Module		X
DOM Level 2: Traversal and Range		X
Encrypted Media Extensions		X
execCommand		X
Fetch		X
File API		X
Gamepad		X
Geolocation API Specification		X
HTML: Broadcasting		X
HTML: Plugins		X
HTML: History Interface		X
HTML: Web Storage		X
Media Capture and Streams		X
Media Source Extensions		X
Navigation Timing		X
Performance Timeline		X
Pointer Lock		X
Proximity Events		X
Selection API		X
The Screen Orientation API		X
Timing control for script-based animations		X
URL		X
User Timing Level 2		X
W3C DOM4		X
Web Notifications		X
WebRTC 1.0		X

TABLE I: Listing of which standards were disabled in the evaluated conservative and aggressive hardened browser configurations.

Statistic	Conservative	Aggressive
Standards blocked	15	45
Previous CVEs #	89	123
Previous CVEs %	52.0%	71.9%
LOC Removed #	37,848	53,518
LOC Removed %	50.00%	70.76%
% Popular sites broken	7.14%	15.71%
% Less popular sites broken	3.87%	11.61%

TABLE II: Cost and benefit statistics for the evaluated conservative and aggressive browser configurations.

Next we had two evaluators visit each of these 330 websites under three browsing configurations, for 60 seconds each. Our decision to use 60 seconds per page is based on prior research [38] finding that that users on average spend under a minute per page.

Our evaluators first visited each site in an unmodified Firefox browser, to determine the author-intended functionality of the website. Second, they visited in a Firefox browser in the above mentioned conservative configuration. And then finally, a third time in the aggressive hardened configuration.

For the conservative and aggressive tests, the evaluators recorded how the modified browser configurations affected each page, using the same 1–3 scale described in Section IV-C. Our evaluators independently gave each site the same 1–3 ranking 97.6% of the time for popular sites, and 98.3% of the time for less popular sites, giving us a high degree of confidence in their evaluations. The "% Popular sites broken" and "% Less popular sites broken" rows in Table II give the results of this measurement.

To further increase our confidence the reported site-break rates, our evaluators recorded, in text, what broken functionality they encountered. We were then able to randomly sample and check these textual descriptions, and ensure that our evaluators were experiencing similar broken functionality. The consistency we observed through this sampling supports the internal validity of the reported site break rates.

As Table II shows, the trade off between gained security and lessened usability is non-linear. The conservative configuration disables code paths associated with 52% of previous CVEs, and removes 50% of ELoC, while affecting the functionally of only 3.87%-7.14% of sites on the internet. Similarly, the aggressive configuration disables 71.9% of code paths associated with previous CVEs and over 70% of ELoC, while affecting the usability of 11.61%-15.71% of the web.

B. Usability Comparison

	% Popular sites broken	% Less popular sites broken	Sites tested
Conservative Profile	7.14%	3.87%	330
Aggressive Profile	15.71%	11.61%	330
Tor Browser Bundle	16.28%	7.50%	100
NoScript	40.86%	43.87%	330

TABLE III: How many popular and less popular sites break when using our conservative and aggressive hardening profiles, compared against other popular browser security tools.

We also tested the usability of our sample browser configurations against other popular browser security tools. We compared our conservative and aggressive configurations first with Tor Browser and NoScript, each discussed in Section II-B. We find that the conservative configuration has the highest usability of all four tested tools, and that the aggressive hardened configuration is roughly comparable to the default configuration of the Tor Browser. The results of this comparison are given in Table III.

We note that this comparison is not included to imply which method is the most secure. The types of security problems addressed by each of these approaches are largely intended to solve different types of problems, and all three compose well (i.e., one could use a cost-benefit method to determine which Web API standards to enable *and* harden the build environment and route traffic through the Tor network *and* apply per-origin rules to script execution). However, as Tor Browser and NoScript are widely used security tools, comparing against them gives a good baseline for usability, especially for security conscious users.

We tested the usability using the same technique we used for the conservative and aggressive browser configurations, described in Section VI-A2; the same two evaluators visited the same 175 popular and 155 less popular sites, but compared the page in an unmodified Firefox browser with the default configuration of the NoScript extension.

The same comparison was carried out for default Firefox against the default configuration of the Tor Browser bundle[6]. The evaluators again reported very similar scores in their evaluation, reaching the same score 99.75% of the time when evaluating NoScript and 90.35% when evaluating the Tor Browser. We expect this lower agreement score for the Tor Browser is a result of our evaluators being routed differently through the Tor network, and receiving different versions of the website based on the location of their exit nodes.[7]

As Table III shows, the usability of our conservative and aggressive configurations is as good as or better than other popularly used browser security tools. This suggests that, while our Web API standards cost-benefit approach has some affect on usability, it is a cost some users would be willing to bear, given the popularity of these security tools.

VII. Browser Extension

As part of this work, we are also releasing a Firefox browser extension that allows users to harden their browsers using the same standard disabling technique described in this paper. The extension is available both as source code[8] and on Mozilla's Add-On register for easy, one-click install[9].

Our browser extension uses the same Web API standard disabling technique described in Section III to dynamically control the DOM-related attack surface to expose to websites. The extension allows users to deploy the same conservative and aggressive hardened browser configurations described in Section VI-A. Extension users can also create their own hardened configurations by selecting any permutation of the 74 measured Web API standards to disable.

Hardened configurations can be adjusted over time, as the relative security and benefit of different browser features changes. This fixed-core-functionality, updated-policies deployment model works well for popular web-modifying browser extensions (such as AdBlock, PrivacyBadger and Ghostery). Our browser-hardening extension similarly allows users to subscribe to configuration updates from external sources

[6]Smaller sample sizes were used when evaluating the Tor Browser because of time constraints, not for fundamental methodological reasons.

[7]We chose to *not* fix the Tor exit node in a fixed location during this evaluation to accurately recreate the experience of using the default configuration of the TBB.

[8]URL Redacted for review.

[9]URL Redacted for review.

(trusted members of the security community, a company's IT staff, security-and-privacy advice groups, etc.), or allows users to create their own configurations.

If a browser standard were found to be vulnerable to new attacks in the future, security sensitive users could update their hardened configurations to remove it. Likewise, if other features became more popular or useful to users on the web, future hardened configurations could be updated to allow those standards. The extension enables users to define their own cost-benefit balance in the security of their browser, rather than prescribing a specific configuration.

Finally, the tool allows users to create per-origin attack-surface policies, so that trusted sites can be granted access to more JavaScript-accessible features and standards than unknown or untrusted websites. Similar to, but finer grained than, the origin based policies of tools like NoScript, this approach allows users to better limit websites to the least privilege needed to carry out the sites' desired functionality.

VIII. Discussion

Below we outline some techniques which can be used with our extension to maximize functionality for trusted websites while simultaneously limiting the threat posed by unknown, untrusted sites.

A. Potential Standards for Disabling

Standards that impose a large cost to the security and privacy of browser users, while providing little corresponding benefit to users, should be considered for removal from the browser. While the history of the web shows such steps to be rare, Mozilla's decision to remove the *Battery API* shows that Web API standard removal is feasible.

We identify several standards as candidates for removal from the browser, based on the low benefit they provide (i.e. few websites suffer reduced functionality when the standard is removed), and the high risk they pose to users' privacy and security. The *High Resolution Time Level 2*, *Canvas* and *Web Audio* APIs have all been leveraged in attacks in academic security research and have been associated with CVEs (several severe). And with perfect agreement, our testers did not encounter any sites with broken functionality when these standards were removed.

While its easy to imagine use cases for each of these standards, our measurements indicate that such use cases are rare. The overwhelming majority of websites do not require them to deliver their content to users. Disabling these standards by default, and requiring users to actively enable them, much like access to a user's location or webcam, would improve browser security at a minimal cost to user convenience.

B. Dynamic Policy Configuration

By default, our design uses a global policy for all websites. Users can allow functionality on a per-origin basis with a workflow similar to that used in popular ad blocking extensions. This approach could also be modified to apply different levels of trust to different origins of code. The Tor Browser Bundle does something similar with the concept of a *URL bar origin*. NoScript similarly controls script execution depending on the URL serving the script. Our Web API standards firewalling approach could be employed in a similar way, with our conservative and aggressive profiles serving as sensible defaults. A set of community-derived feature rules could also be maintained for different websites, much like the EasyList ad blocker filter [23]. Rather than blocking DOM tree elements or URLs (for ad blocking), the filter list would function as a list of allowed Web API standards for each website.

One could also apply heuristics to infer a user's level of trust with a given website. When visiting a site for the first time, a user has no preexisting relationship with that origin. Under this insight, different features could be exposed depending on how often a user visits a site, or whether the user has logged in to that website. Google's Progressive Web Apps incorporate frequency of visit metrics in the amount of functionality that they expose to users: when a user visits a page a sufficient number of times, it gains the ability to prompt the user to install that website as a shortcut on their home screen [26].

Another simple contextual cue for providing additional security is the use of private browsing modes. When a user explicitly enables such a feature, they are implicitly signaling some change in security posture. This signal is already being used, as Firefox enables enhanced tracking protection features when a user enables private browsing mode. Whether a user's posture change is due to privacy, security, or simply history retention purposes, it could also be used to activate a "reduced functionality" browsing mode.

C. Allowing Features For Trusted Apps

Other contexts exist for which hardening a browser through feature removal is a logical decision. The web is used in single-application graphical user interface in situations, like information kiosks or electronic medical record access, or in complex-but-well-defined web applications, like mapping systems or web games.

The developers of such "well defined" applications could know a priori which features their application needs, and allow only those. The user would not experience any degraded experience, and the amount of features available to a potential attacker would be significantly reduced.

Such a system could be exploited by malicious site asking the user to enable risky feature. However, the hardened browser still provide more security than the current, default browser, for two reasons. First, user education could help some users become more careful about which sites to grant feature-access too. And second, requiring users to allow access to risky features creates a "speed bump" in the attack process. This may give users a chance to think twice and back out of the suspicious site they are visiting. A "speed bump" would also prevent such attacks from being the default action of visiting the website, and at least reduce the number of vulnerable web users.

D. Malicious Extensions

Beyond protecting the user from websites, this approach could also be used to protect the user from malicious exten-

sions [31]. Although extensions have their own fine grained permission model, this does not extend to what functionality they can access within the DOM.

Unfortunately, in browsers where extension load order is not well defined, a malicious extension could load before our extension and have access to all of the browser's functionality before it is replaced with Proxy objects. Browser vendors could solve this by either allowing users to define an extension load order, or creating an additional permission that allows permitted extensions to disable certain DOM functionality for both pages and extensions before loading the page or any other extension would enable this use case.

IX. Conclusion

As browser vendors move away from plugins and provide more functionality natively within the DOM, the modern web browser has experienced a terrific growth in features available to every web page that a user might visit. Indeed, part of the appeal of the web is the ability to deploy complex, performant software without the user even realizing that it has happened.[10]

However, the one size fits all approach to exposing these features to websites has a cost which is borne in terms of vulnerabilities, exploits, and attacks. Simplistic approaches like ripping out every feature that isn't absolutely necessary are not practical solutions to this problem. We believe that enabling users to contextually control and empirically decide which features are exposed to which websites will allow the web to continue to improve the browser's feature set and performance, while still being usable in high risk situations where the security of a reduced feature set is desired.

References

[1] "Adblock plus," https://adblockplus.org/, [Online; accessed 16-October-2015].

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

[3] "Android developer's guide: System permissions," https://developer.android.com/guide/topics/security/permissions.html, 2015, [Online; accessed 17-February-2016].

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

[5] "High resolution time level 2," https://www.w3.org/TR/hr-time-2/, 2016, [Online; accessed 11-November-2016].

[6] "Web workers," https://www.w3.org/TR/workers/, 2016, [Online; accessed 13-August-2016].

[7] "Webrtc 1.0: Real-time communication between browsers," https://www.w3.org/TR/webrtc/, 2016, [Online; accessed 11-August-2016].

[8] 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*. ACM, 2014, pp. 674–689.

[9] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, "Fpdetective: dusting the web for fingerprinters," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1129–1140.

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

[11] F. Alaca and P. van Oorschot, "Device fingerprinting for augmenting web authentication: Classification and analysis of methods," in *Proceedings of the 32th Annual Computer Security Applications Conference*, 2016.

[12] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 623–639.

[13] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie, "Short paper: a look at smartphone permission models," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 63–68.

[14] A. Bachmann and A. Bernstein, "When process data quality affects the number of bugs: Correlations in software engineering datasets," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 62–71.

[15] 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].

[16] V. Blue, "You say advertising, i say block that malware," http://www.engadget.com/2016/01/08/you-say-advertising-i-say-block-that-malware/, 2016, [Online; accessed 15-February-2016].

[17] M. Bryant, "The noscript misnomer - why should i trust vjs.zendcdn.net?" https://thehackerblog.com/the-noscript-misnomer-why-should-i-trust-vjs-zendcdn-net/index.html, 2015, [Online; accessed 12-August-2016].

[18] E. Dahlstrm, 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.

[19] A. Das, N. Borisov, and M. Caesar, "Tracking mobile web users through motion sensors: Attacks and defenses," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[20] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," DTIC Document, Tech. Rep., 2004.

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

[22] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1388–1401.

[23] Fanboy, MonztA, Famlam, and Khrin, "Easylist," https://easylist.adblockplus.org/en/, [Online; accessed 16-October-2015].

[24] N. Gelernter and A. Herzberg, "Cross-site search attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1394–1405.

[25] Google, "boringssl - git at google," https://boringssl.googlesource.com/boringssl/, 2016, [Online; accessed 12-November-2016].

[26] Google Developers, "Progressive web apps," https://developers.google.com/web/progressive-web-apps/, 2016.

[27] D. Gruss, D. Bidner, and S. Mangard, "Practical memory deduplication attacks in sandboxed javascript," in *European Symposium on Research in Computer Security*. Springer, 2015, pp. 108–122.

[28] S. Guarnieri and B. Livshits, "Gatekeeper: mostly static enforcement of security and reliability policies for javascript code," in *Proceedings of the 18th conference on USENIX security symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 151–168. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855778

[10] https://xkcd.com/1367/

[29] M. Heiderich, T. Frosch, and T. Holz, "Iceshield: detection and mitigation of malicious websites with a frozen dom," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 281–300.

[30] G. Ho, D. Boneh, L. Ballard, and N. Provos, "Tick tock: building browser red pills from timing side channels," in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, 2014.

[31] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 641–654.

[32] H. Kim, S. Lee, and J. Kim, "Exploring and mitigating privacy threats of html5 geolocation api," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 306–315.

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

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

[35] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: timing attacks using css filters," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1055–1062.

[36] M. Lamouri and M. Cceres, "Screen orientation," http://www.w3.org/TR/screen-orientation/, 2105.

[37] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *37th IEEE Symposium on Security and Privacy (S&P 2016)*, 2016.

[38] C. Liu, R. W. White, and S. Dumais, "Understanding web browsing behaviors through weibull analysis of dwell time," in *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2010, pp. 379–386.

[39] G. Maone, "Noscript - javascript/java/flash blocker for a safer firefox experience!" https://noscript.net/, 2015, [Online; accessed 08-February-2015].

[40] L. A. Meyerovich and B. Livshits, "Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 481–496.

[41] M. S. Miller, "Google caja," https://developers.google.com/caja/, 2013.

[42] "Cve-2011-2363," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-2363, MITRE, 2011, [Online; accessed 11-August-2016].

[43] "Cve-2012-4171," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4171, MITRE, 2012, [Online; accessed 11-August-2016].

[44] "Cve-2013-2031," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2031, MITRE, 2013, [Online; accessed 11-August-2016].

[45] "Cve-2015-0818," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0818, MITRE, 2015, [Online; accessed 11-August-2016].

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

[47] Mozilla Corporation, "Dxr," https://github.com/mozilla/dxr, 2016.

[48] 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.

[49] L. Olejnik, G. Acar, C. Castelluccia, and C. Diaz, "The leaking battery a privacy analysis of the html5 battery status api," Cryptology ePrint Archive, Report 2015/616, 2015, ht tp://eprint. iacr. org, Tech. Rep., 2015.

[50] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1406–1418.

[51] A. Ozment and S. E. Schechter, "Milk or wine: does software security improve with age?" in *Usenix Security*, 2006.

[52] A. Patrizio, "How forbes inadvertently proved the anti-malware value of ad blockers," http://www.networkworld.com/article/3021113/security/forbes-malware-ad-blocker-advertisements.html, 2016, [Online; accessed 15-February-2016].

[53] M. Perry, E. Clark, and S. Murdoch, "The design and implementation of the tor browser," https://www.torproject.org/projects/torbrowser/design/#fingerprinting-linkability, 2015, [Online; accessed 15-February-2016].

[54] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[55] P. Snyder, L. Ansari, C. Taylor, and C. Kanich, "Browser feature usage on the modern web," in *Proceedings of the 2016 Internet Measurement Conference (to appear)*, 2016.

[56] S. Son and V. Shmatikov, "The postman always rings twice: Attacking and defending postmessage in html5 websites." in *NDSS*, 2013.

[57] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International Conference on World Wide Web*. ACM, 2010, pp. 921–930.

[58] Y. Tian, Y. C. Liu, A. Bhosale, L. S. Huang, P. Tague, and C. Jackson, "All your screens are belong to us: attacks exploiting the html5 screen sharing api," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 34–48.

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

[60] 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*. ACM, 2015, pp. 1382–1393.

[61] T. Van Goethem, M. Vanhoef, F. Piessens, and W. Joosen, "Request and conquer: Exposing cross-origin resource size," in *Proceedings of the Usenix Security Symposium*, 2016.

[62] M. Vanhoef and T. Van Goethem, "Heist: Http encrypted information can be stolen through tcp-windows," Blackhat, 2016.

[63] Web Hypertext Application Technology Working Group (WHATWG), "Html living standard," https://html.spec.whatwg.org/, 2015.

[64] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Zigzag: Automatically hardening web applications against client-side validation vulnerabilities," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 737–752.

[65] M. Xu, Y. Jang, X. Xing, T. Kim, and W. Lee, "Ucognito: Private browsing without tears," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 438–449.

[66] T. Zimmermann, N. Nagappan, and A. Zeller, "Predicting bugs from history," in *Software Evolution*. Springer, 2008, pp. 69–88.

Standard Name	Abbreviation	# Alexa 10k Using	Site Break Rate	Agree %	# CVEs	# High or Severe	% ELoC	Enabled attacks
WebGL	WEBGL	852	<1%	93%	31	22	27.43	[11, 30, 37]
HTML: Web Workers	H-WW	856	0%	100%	16	9	1.63	[30]
WebRTC	WRTC	24	0%	93%	15	4	2.48	[11, 22]
HTML: The canvas element	H-C	6935	0%	100%	14	6	5.03	[8, 11, 22, 30, 35, 37]
Scalable Vector Graphics	SVG	1516	0%	98%	13	10	7.86	
Web Audio API	WEBA	148	0%	100%	10	5	5.79	[11, 22]
XMLHttpRequest	AJAX	7806	32%	82%	11	4	1.73	
HTML	HTML	8939	40%	85%	6	2	0.89	[9, 48]
HTML 5	HTML5	6882	4%	97%	5	2	5.72	
Service Workers	SW	0	0%	-	5	0	2.84	[24, 60, 61]
HTML: Web Sockets	H-WS	514	0%	95%	5	3	0.67	
HTML: History Interface	H-HI	1481	1%	96%	5	1	1.04	
Indexed Database API	IDB	288	<1%	100%	4	2	4.73	[8, 11]
Web Cryptography API	WCR	7048	4%	90%	4	3	0.52	
Media Capture and Streams	MCS	49	0%	95%	4	3	1.08	[58]
DOM Level 2: HTML	DOM2-H	8956	13%	89%	3	1	2.09	
DOM Level 2: Traversal and Range	DOM2-T	4406	0%	100%	3	2	0.04	
HTML 5.1	HTML51	2	0%	100%	3	1	1.18	
Resource Timing	RT	433	0%	98%	3	0	0.10	
Fullscreen API	FULL	229	0%	95%	3	1	0.12	
Beacon	BE	2302	0%	100%	2	0	0.23	
DOM Level 1	DOM1	9113	63%	96%	2	2	1.66	
DOM Parsing and Serialization	DOM-PS	2814	0%	83%	2	1	0.31	
DOM Level 2: Events	DOM2-E	9038	34%	96%	2	0	0.35	
DOM Level 2: Style	DOM2-S	8773	31%	93%	2	1	0.69	
Fetch	F	63	<1%	90%	2	0	1.14	[24, 60, 61]
CSS Object Model	CSS-OM	8094	5%	94%	1	0	0.17	[48]
DOM	DOM	9050	36%	94%	1	1	1.29	
HTML: Plugins	H-P	92	0%	100%	1	1	0.98	[9, 11]
File API	FA	1672	0%	83%	1	0	1.46	
Gamepad	GP	1	0%	71%	1	1	0.07	
Geolocation API	GEO	153	0%	96%	1	0	0.26	[32, 65]
High Resolution Time Level 2	HRT	5665	0%	100%	1	0	0.02	[12, 24, 27, 30, 35, 50, 60]
HTML: Channel Messaging	H-CM	4964	0%	0.025	1	0	0.40	[56, 64]
Navigation Timing	NT	64	0%	98%	1	0	0.09	
Web Notifications	WN	15	0%	100%	1	1	0.82	
Page Visibility (Second Edition)	PV	0	0%	-	1	1	0.02	
UI Events	UIE	1030	<1%	100%	1	0	0.47	
Vibration API	V	1	0%	100%	1	1	0.08	
Console API	CO	3	0%	100%	0	0	0.59	[30]
CSSOM View Module	CSS-VM	4538	0%	100%	0	0	2.85	[9]
Battery Status API	BA	2317	0%	100%	0	0	0.15	[11, 22, 48, 49]
CSS Conditional Rules Module Lvl 3	CSS-CR	416	0%	100%	0	0	0.16	
CSS Font Loading Module Level 3	CSS-FO	2287	0%	98%	0	0	1.24	[9, 11]
DeviceOrientation Event	DO	0	0%	-	0	0	0.06	[11, 19]
DOM Level 2: Core	DOM2-C	8896	89%	97%	0	0	0.29	
DOM Level 3: Core	DOM3-C	8411	4%	96%	0	0	0.25	
DOM Level 3: XPath	DOM3-X	364	1%	97%	0	0	0.16	
Encrypted Media Extensions	EME	9	0%	100%	0	0	1.91	
HTML: Web Storage	H-WB	7806	0%	83%	0	0	0.55	[11, 30, 65]
Media Source Extensions	MSE	1240	0%	95%	0	0	1.97	
Selectors API Level 1	SLC	8611	15%	89%	0	0	0.00	
Script-based animation timing control	TC	3437	0%	100%	0	0	0.08	[48]
Ambient Light Sensor API	ALS	18	0%	89%	0	0	0.00	[48]

TABLE IV: This table includes data on all 74 measured Web API standards, excluding the 20 standards with a 0% break rate, 0 associated CVEs and accounting for less than one percent of measured effective lines of code:

1) The standard's full name
2) The abbreviation used when referencing this standard in the paper
3) The number of sites in the Alexa 10k using the standard, per [55]
4) The portion of measured sites that were broken by disabling the standard. (see Section IV-C)
5) The mean agreement between two independent testers' evaluation of sites visited while that feature was disabled (see Section IV-C)
6) The number of CVEs since 2010 associated with the feature
7) The number of CVEs since 2010 ranked as "high" or "severe"
8) The percentage of lines of code exclusively used to implement this standard, expressed as a percentage of all 75,650 lines found using this methodology (see Section IV-D2).
9) Citations for papers describing attacks relying on the standard