Improving Web Privacy And Security with a Cost-Benefit Analysis of the Web API

by

Peter Edwin Snyder
B.A., Lawrence University, 2006

Thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2018

Chicago, Illinois

Dedicated to the Snyders and the Scrantons.

ACKNOWLEDGMENT

I am extremely grateful to Chris Kanich, my advisor, for his guidance, encouragement, and good humor. Without that hat-trick, every part of this work would be weaker, and I'd still be miserably building PHP websites at Caribou Coffee. A deep, sincere, mega-thanks Chris :).

I also grateful to the other members of my committee, Stephen Checkoway, Jakob Eriksson, Damon McCoy and Venkat Venkatakrishnan, for their generous time and help.

Thank you too, to the other collaborators I got to work with. Lara Ansari, Periwinkle Doerfler, Michael K. Reiter, Cynthia Taylor and Laura Watiker, I'm glad and grateful I got to work with you.

Thanks also to the other BITSLab folks. Even though we didn't research together, I'm glad I could take in some of your smarts, charm and general pleasantness though osmosis. Sara Amini, Tomas Gerlich, Mohammad Ghasemisharif, Yanzi Jin, Mohammad Taha Khan, Timothy Merrifield, ABM Musa and Sepideh Roghanchi, thanks for making the six-years more pleasant and fun than they probably should have been.

Finally, thanks to the folks who have been angels to me outside of UIC, without whom this wouldn't have gotten done. Khen Chee Leuk for being the closest thing to a nurse (beside the actual nurses) and helping me after I goofed it and couldn't move my arms for a bit. Alan Yu for being sweet, forgiving and keeping my PhD-ego level headed. And Sam and Molly Scranton for helping me not be such as sad-sack when I was being a doof.

Oh, and family too: Kristen, Julia, and Peter. I guess they played some role in all this.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CDF	Contained Document Format
CSP	Content Security Policy
CSS	Cascading Style Sheet
CVE	Common Vulnerabilities and Exposures Advisory
DOM	Document Object Model
ECMA	European Computer Manufacturers Association
ELoC	Effective Lines of Code
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
SVG	Scalable Vector Graphics
TBB	Tor Browser Bundle
URL	Universal Resource Locator
W3C	World Wide Web Consortium
WebIDL	Web Interface Definition Language
WHATWG	Web Hypertext Application Technology Working Group
WWW	World Wide Web
XSS	Cross-Site Scripting

SUMMARY

Over the last two decades, the web has grown from a system for delivering static documents, to the world's most popular application platform. As the web has become more popular and successful, browser vendors have added increasingly more functionality into the web platform. While some of this functionality has proven very useful and allowed site authors to create applications that users enjoy, a large subset of functionality in the browser goes largely unused. Another sizable subset of functionality has been leveraged by malicious parties to harm browser users.

This dissertation presents an effort to improve web privacy and security by applying a cost-benefit analysis to the Web Application Programming Interface (API), as it is implemented in popular web browsers. The goal of the work is to apply the principal of "least privilege" to the web, and restrict websites to functionality they need to carry out user-serving ends. The work pursues that end through a novel method of measuring the costs and benefits associated with each standard in the Web API, identifying different high-benefit and low-risk subsets of the Web API, and evaluating a variety of approaches for restricting websites to these safer subsets.

This dissertation covers four distinct research efforts, each of which contribute to the overall goal of improving privacy and security on the web. First, this dissertation describes an automated technique for measuring Web API use on the web by instrumenting the DOM in a commodity web browser, automating the browser to interact with websites in a manner that elicits most of the same feature use as human users encounter, and recording what functionality is triggered during this execution. This section also presents the results of applying this automated recording methodology to the entire Alexa 10k, both in default browser configurations, and with popular blocking extensions installed.

Second, this work presents a systematic measurement of the costs and benefits of each standard in the Web API. This work models a standard's benefit as the percentage of sites on the web that require the standard to carry out their core, user-serving functionality, and models a standard's cost as the security risk the standard poses to users (measured as the number of recent vulnerabilities relating to the standard's implementation, the additional complexity the standard's implementation brings to the browser code base, and the number of papers in recent top security conferences and journals that leverage the standard).

Third, this work presents a method to apply these cost-benefit measurements to the web as it exists today, to try and improve users' privacy and security. This technique entails the creation of a Web API-blocking browser extension, that restricts which features current websites are able to access. This section also presents findings from making this tool available to general web users.

Finally, this work describes an alternate system for designing web applications that provides client-enforced privacy and security guarantees. The design of this system builds on the previously discussed per-standard cost-benefit methodology to determine which Web API features sites generally need.

Each of these works support the overarching finding that privacy and security on the web can be improved with only a small cost to the user experience. In contrast to the current practice of giving every site access to every feature in the browser (with only minor exceptions), this work presents a data driven approach to restricting websites to a subset of safer, user-serving functionality. This dissertation further shows that the privacy and security benefits of enforcing this “least privilege” approach to the Web API would be meaningful, and real world deployment of these techniques shows that at least some web users find the approach useful in protecting their privacy and security.

CHAPTER 1

INTRODUCTION

The World Wide Web (WWW) is possibly the world's largest open system, allowing information to be transferred, and individuals to interact, with a speed and ease that would have been unimaginable only a generation ago. This growth in popularity has occurred alongside an explosion in the type of functionality provided to websites, as both cause and effect. Where websites were initially limited to static documents of images and hypertext, websites now rival traditional applications in terms of size and capability. Web applications are frequently megabytes in size, and can access graphics cards, web cameras, microphones, perform low-level audio synthesis, and carry out parallel computation, just to name a few examples.

Each of these features brings some plausible benefit to users, and from a narrow point of view, web users benefit as the web gains more functionality. More functionality means web site authors can create richer, more capable applications. However, this point of view ignores half of the ledger.

In practice, feature growth brings both costs and benefits; it benefits users by enabling new types of web applications that users may enjoy, but harms users by adding risk to the platform. Increased complexity can harm users by expanding the user's trusted computing base (making bugs and vulnerabilities more likely), broadening the attack surface of the platform, and making information flows more difficult for users to understand.

Instead of providing websites with a maximal set of features, web users would be best served by restricting what functionality websites can access, and only allow websites to access features where the benefits outweigh the costs. Put more casually, web browsers should only allow websites access to functionality when the cargo is worth the freight.

This work presents an attempt to measure the costs and benefits of feature growth in the browser, both to understand how to improve the web as it exists today, and to explore alternate ways of deploying web applications with an emphasis on security and privacy. This approach of improving browser privacy and security by restricting websites to a reduced feature set makes it very different from most other research in the area, which focuses primarily on either changing the implementations of a small number of features in the browser or developing new methods for identifying implementation errors.

1.1 Common Terms Used in Work

This section presents terms that will be used throughout this work. These terms are presented upfront to ease the reading of the rest of the work.

A browser **feature** is a piece of functionality, implemented a JavaScript function, method or property, implemented in a web browser, by a browser vendor. Browser features are intended to be used by websites, through JavaScript delivered by the website, to carry out interactivity and functionality on the website. Examples of browser features include `HTMLCanvasElement.prototype.toBlob`, used to read the contents of a `<canvas>` element in a website into a string, and `Document.prototype.getElementById`, used to query an element in a website.

A **standard** is a set of related features, defined by a standards organization, describing what features browser vendors should implement, and how those features should function. Standards generally define many features, intended to be used together to accomplish related goals. For example, the *SVG* (1) standard defines features used for creating and modifying Scalable Vector Graphics (SVG) elements, and the *WebGL* (2) and *WebGL 2.0* (3) standards define features for performing 3D graphics operations in web pages.

The **Web API** refers to the set of all the features in all of the standards implemented in modern web browsers. While at any point in time different browsers will have implemented different parts of the Web API, those differences tend to be temporary, minor and mainly due to differing organizational priorities regarding newly introduced standards.

1.2 Contributions

The underlying motivation, and core thesis, of this work, is that privacy and security on the web can be improved, with only small effects on usability, by restricting what functionality websites can access. This conclusion is built to in four steps, and through the following contributions:

- A web-scale measurement of how the Web API is used on the web today. This study contained an automated measurement of browser functionality used on sites in the Alexa 10k (at time of measurement), with a novel method of distinguishing between features used for core-site-functionality, and features used for non-user-serving purposes (e.g. advertising and tracking).

- A comprehensive measurement of the costs and benefits of each standard in the Web API. This work models a standard's benefit as the number of sites on the web that require the standard to carry out the site's core functionality. It models a standard's cost in three ways: as the number of recent publicity disclosed vulnerabilities relating to the standard, the number of lines of code uniquely needed to implement the standard, and the number of papers describing attacks that leverage the standard in top security conferences and journals.
- Findings learned and vulnerabilities discovered in the development and maintenance of a browser extension to restrict Web API access on the web. This work also presents usability measurements taken during an in-lab evaluation of the tool, and broader findings from real-world use of the extension.
- The design and evaluation of Contained Document Format (CDF), a system of developing and deploying web applications with functionality similar to most modern websites, but providing client-enforced protections and dataflow guarantees.

1.3 Organization

The remainder of this dissertation is organized in the following manner.

Chapter 2 provides background material on attacks and defenses related to browser security.

Chapter 3 describes both an automated method for measuring browser feature use on the web, and the results of applying that technique to the Alexa 10 (as it stood at the time of measurement). This chapter also includes the description of a method for distinguishing user-serving feature use from non-user-serving (e.g. advertising and tracking related) feature use.

Chapter 4 presents a method for measuring the costs and benefits of the Web API standards implemented in modern web browsers, and the results of applying that methodology to a modern, representative web browser.

Chapter 5 describes efforts to apply the findings from Chapters 3 and 4 to the web as it exists today, in the form of a publicly-released browser extension that is being used by approximately 1,000 real-world users. This chapter also includes a usability measurement of this browser extension-approach, comparisons with other popular web security and privacy tools, and the description of a security vulnerability discovered in popular security and privacy tools as a result of this work.

Chapter 6 presents the design of an alternative system for developing and deploying web applications that, building on the findings presented in Chapters 3 and 4, is intended to provide site authors with the expressiveness needed to design interactive modern web sites, but while providing client-enforced security and privacy guarantees to web users.

Chapter 7 concludes with some discussion of these findings, and how they could be pursued further.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 The Web Application Model

This section provides a brief overview of how web applications are designed, and the role of the Web API in building modern web applications. This description is not intended to be comprehensive, but to provide enough context so that the rest of this dissertation can be understood by readers without experience in web development.

Browsers allow JavaScript code to interact with websites in several steps. First, browsers parse the received Hyper Text Markup Language (HTML) into a JavaScript accessible data-structure, that roughly-mirrors the tree-based structure of the original document. This structure allows JavaScript code to access and modify the document-tree, using JavaScript properties and functions. These tree-modifying JavaScript features are collectively called the Document Object Model (DOM) and are standardized across browsers.

Browsers also provide websites access to a large number of JavaScript features that are not directly related to modifying the document-tree. These features range widely in purpose, including allowing sites to access device hardware, take high-resolution timing measurements, and perform network IO. The term "Web API" refers to the union of functionality related to modifying the rendered HTML document, and these additional JavaScript features.

Modern web applications are thus the combination of two sources of code: first, the Web API functionality implemented in the browser, and second, the website's JavaScript code that uses the Web API to implement the site's functionality.

Web applications have unique properties that make them difficult to secure. First, every JavaScript code unit has access to nearly all features in the Web API. There are a few exceptions, where users are prompted to allow access to certain functionalities, relating to tasks like accessing hardware devices and geo-locating the user. These restrictions apply to a tiny fraction of features; websites have access by default to the vast majority of the Web API.

Second, the web application model lacks a formal way of allowing code units to interact. Instead, all code units are executed in a common namespace, and code units collaborate by accessing and modifying a single global variable (implemented in the browser as `window`), or by modifying the globally accessible representation of the HTML document (implemented in the browser as `window.document`). This model makes it difficult to execute a code unit without allowing it to read and modify the execution environment. All code executed in the page (whether that code was intended by the page author, loaded by a third party to implement an advertising system, or maliciously include as part of a Cross-Site Scripting (XSS) attack) gets equal access to the capabilities, secrets and data available to the website or application.

2.2 Web API Standardization and Growth

The current Web API is the result of many years of growth and standardization. The standardization process aims to ensure that each browser's implementation of the Web API is compatible and that web developers only need to support a single code base to have their applications work in all modern browsers.

The current standardization process grew out of frustrations and incompatibilities in earlier web browsers. The two early major browsers, Netscape's "Netscape Navigator" and Microsoft's "Internet Explorer", initially provided websites with very different systems for building interactive websites. These models differed in ways that were both trivial (e.g. different names for properties and methods that provided identical functionality) and fundamental (e.g. inverse event delegation models).

To keep the browsers' API s from drifting further apart, and to make the web a more appealing platform for developers, the tasks of standardizing and growing the Web API was moved from the browser vendors to standards organizations. The two most significant standards organizations for the web are the World Wide Web Consortium (W3C), which oversaw the original web standards, and is the main body overseeing the development of new Web API standards, and the Web Hypertext Application Technology Working Group (WHATWG), which was formed as a response to what was seen as slow progress in the W3C. Other groups, such as the Kronos Group and European Computer Manufacturers Association (ECMA), also manage relevant standards (the *WebGL* (2) and *JavaScript* (4) standards, respectively).

2.3 Client Side Browser Defenses

There are many techniques to "harden" the browser by limiting what JavaScript pages are allowed to execute. These defenses can be split into two categories: those configured by the user, and those configured by the website author.

In the user-configured category, Adblock (5) and NoScript (6) are popular browser extensions that control what JavaScript is executed in the browser, based on the Universal Resource Locator (URL) the JavaScript came from. Adblock's primary function is to block ads for aesthetic purposes, but it can also prevent infection by malware being served in those ads (7; 8). Adblock blocks feature use by preventing the loading of resources from certain domains.

NoScript decides whether JavaScript can execute, based on the URL the code came from. By default, NoScript prevents JavaScript execution from all origins, rendering a large swath of the web unusable. In its default configuration, NoScript allows JavaScript execution from a built-in set of trusted origins. This built-in allow-list has resulted in a proof of concept exploit via purchasing expired, allowed domains (9). Beyond these popular tools, IceShield (10) dynamically detects suspicious JavaScript calls within the browser, and modifies the DOM to prevent attacks.

The Tor Browser (11) disables, or prompts the user before using a number of, features by default. Tor Browser disable many JavaScript features, most significantly *SharedWorkers* (12), and prompts users before allowing pages to use the Canvas, GamePad API, WebGL, Battery API, and Sensor standards (13). These particular features are disabled because they enable techniques which violate the Tor Browser's security and privacy goals.

On the site-author side, Content Security Policy (CSP) allows server operators to limiting what kinds of JavaScript functionality, and sources of code, can be executed, through either Hypertext Transfer Protocol (HTTP) headers, or HTML *meta* tags. CSP allows web developers to constrain code on their sites so that potential attack code cannot access functionality deemed unnecessary or dangerous (14). Conscript is another client-side implementation which allows a hosting page to specify policies for any third-party scripts it includes (15). There are also a number of technologies selected by the website author but enforced on the client side, including Google Caja (16) and GATEKEEPER (17).

There are also models for enforcing policies to limit functionality outside of the web browser. Mobile applications use a richer permission model, where permission to use certain features is asked of the user at either install or run-time (18; 19).

2.4 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 (20).

User tracking is an insidious part of the modern web. Recent work by Radler found that users were less aware of cross-website tracking than they were about of data collection by first party sites, like Facebook and Google. Radler also found that users who were aware of it had greater concerns about unwanted access to private information than those who weren't aware (21). 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 (22). 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 (23).

Tracking was traditionally 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. A variety of tracking blockers prevent these non-cookie tracking mechanisms, including browser fingerprinting (24), JavaScript fingerprinting (25; 26), Canvas fingerprinting (27), clock skew fingerprinting (28), history sniffing (29), cross origin timing attacks (30), evercookies (31), and Flash cookie respawning (32; 33). A variety of these tracking behaviors have been observed in widespread use in the wild (32; 33; 34; 35; 36; 37; 38).

Especially relevant to this work is the use of JavaScript API s for tracking. While some API s, such as the *Beacon standard* (39), are designed specifically for tracking, other API s were designed to support benign functionality, but has been co-opted into tracking purposes (27; 40). Balebako et al. evaluated tools which purport to prevent tracking and found that blocking add-ons were effective (41).

CHAPTER 3

MEASURING BROWSER FEATURE USE

This chapter includes excerpts and figures from a preprint version of material that was later published in *Proceedings of 2016 IMC. Snyder, Peter; Ansari, Laura; Taylor, Cynthia; Kanich, Chris;* The dissertation author was the primary investigator and author of this work.

3.1 Introduction

The first step in understanding the security and privacy implications of the current (and quickly expanding) Web API is to understand what features are being used on the web, and for what purposes. If every site on the web uses a given Web API feature, that is a suggestive (though not determinative) signal that a feature may be useful to web users. Conversely, if a user never visits a website that uses a given Web API feature, then that feature is (trivially) providing no direct benefit to the user.

The design and development of the web makes these kinds of conceptually simple measurements complex to carry out in practice. First, with the exception of trivial cases, one cannot simply “download a website” and count invocations of functions calls, the way one might be able to approximate with traditional applications. Websites are downloaded (and sometimes, generated) dynamically, one portion at a time, depending on user input. The server side of an application can even change while the client is interacting with the application! This means there is no way to meaningfully know if you've downloaded the entire web application, making static analysis of a website difficult, if not impossible.

Similarly, the highly-dynamic nature of JavaScript (how the client-side of web applications are implemented) means that the language is difficult to statically analyze. Even if one could download all of the JavaScript code that comprises a web application, its a non-trivial task to determine what Web API functions are called in a piece of code, let alone determining which code paths the user would execute during a likely interaction with the website.

To confound matters more, all feature invocations on a website are not equally desirable to the user. A user may benefit if a feature is being used to, say, render a news story she wishes to read, while the user might experience harm if the feature is being used to fingerprint her browser for tracking purposes. A useful measurement of Web API use on the web should distinguish between these kinds of cases.

Finally, the scale of the internet means that manual interaction is not feasible to measure a representatively large portion of the web. An automated technique is needed.

For these reasons, determining what Web API features users are likely to encounter on the web, and distinguishing harmful from beneficial feature invocations, is a difficult problem.

This work presents a solution to this problem, in the form of an automated measurement technique that interacts with websites in a manner that approximates how humans would interact with the site. By counting which features are invoked during each automated interaction, we're able to estimate which features real users would encounter when interacting with a website. We then repeated this automated measurement technique with popular advertising and tracking-blocking extensions installed. Comparing the difference in features that are executed under these configurations allows us to distinguish user-serving Web API use from non-user-serving Web API use.

Applying this automated measurement technique to the Alexa 10k answers the original question; which Web API features do web users use when browsing the web. We find, for example, that 50% of the Web API features in the browser are never used by the ten thousand most popular websites, when users are browsing in low trust, non-authenticated scenarios.

We were also able to identify features that are used predominantly for non-user-serving purposes; 10% of Web API features in the browser are blocked 90% of the time when ad and tracking-blocking extensions are installed, and over 83% of features are executed on less than 1% of websites in the presence of these popular blocking extensions.

The data described in this work has been publicly shared and is freely available. The dataset contains our measurements of what JavaScript features are used in the Alexa 10k, both by default browsers, and when ad and tracking blocking extensions are installed. The database with these measurements, along with documentation describing the database's schema, is available online (42).

The rest of this chapter is organized as follows. Section 3.2 describes the data sources used to conduct this work. Section 3.3 describes the methodology used in this chapter in greater detail, and Section 3.4 presents the results of this automated measurement, including how frequently features are used, and which features are blocked by popular blocking extensions. Section 3.5

describes steps taken to verify that our results are correct, and Section 3.6 concludes with some discussion of the significance of these results.

3.2 Data Sources

This work relied on several prior existing sets of data. This Section describes these existing data sets, and how we used them to measure what JavaScript features are used on the modern web.

3.2.1 Alexa Website Rankings

The Alexa rankings are a well-known ordering of websites by traffic. Researchers typically use Alexa's rankings of the worldwide million most popular sites. Alexa also provides other data about these sites, including popularity rankings at country granularity, breakdowns of which sub-sites (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 as representative of the web in general. This set of 10,000 websites represents approximately one-third of all web visits.

3.2.2 Web API Features

As mentioned in Section 1.1, this work uses the term **feature** to denote a browser capability that is accessible by calling a JavaScript function or setting a property on a JavaScript object.

We determined the set of JavaScript-exposed features by reviewing the Web Interface Definition Language (WebIDL) definitions included in the Firefox version 46.0.1 source code. WebIDL is a language that defines JavaScript interfaces implemented in browsers. These WebIDL files are included in the Firefox 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.

3.2.3 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, although occasionally standards organizations delegate responsibility for writing standards to third parties, such as the Khronos group who maintains the current *WebGL* standard. As mentioned in Section 1.1, this work uses the term **standard** to refer to these sets of features, generally grouped by a common purpose.

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.

[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 is to define a mapping between JavaScript API endpoints and the underlying implementations, generally in C++.

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.

In the case of extremely large standards, we identify sub-standards, which define a subset of related features intended to be used together. For example, we treat the subsections of the *HTML* (43) standard that define the basic *Canvas API*, or the *WebSockets API*, as their own standards.

We treated these significant sub-standards as separate units of analysis because they have separate, coherent purposes in the Web API, independent of their parent standard. Many of these significant subs-standards were also 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* (44), *Document Object Model (DOM) Level 2 Core Specification* (45) and *Document Object Model (DOM) Level 3 Core Specification* (46) standards. In such cases, the feature is attributed to the earliest containing standard.

3.2.4 Historical Firefox Builds

We determined when features were implemented in Firefox by examining the 186 versions of Firefox that were released between 2004 and when this work was conducted in 2016, and finding the earliest version that each of the 1,392 features 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 in some cases it took months or years for all features in a standard to be implemented in Firefox. We, therefore, treated the introduction of a standard's most popular feature as the standard's implementation date. For ties (especially relevant when no feature in a standard is used), we used the date of the earliest implemented feature.

3.2.5 Blocking Extensions

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

This work used 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 so as 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 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 used each extension's default configuration, including the default rule sets for which elements and resources to block. No changes were made to the configuration or implementation of either extension.

3.3 Measurement 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 during each visit. 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 the browser was instrumented to determine which features were used on a site, and then concludes with how we used our instrumented browser to measure feature use across the entire Alexa 10k.

3.3.1 Goals

The goal of this automated survey was to determine which browser features are used on the web as it is commonly experienced by users. This required 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 web-page 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 we did not gather such such measurements in this work. As a result, care should be taken before generalizing the following results to browsing in general.

3.3.2 Measuring Extension

We instrumented a recent version of Firefox (version 46.0.1) with a custom browser extension that records each time a JavaScript feature is used. The extension achieves this by injecting 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 Web API before the visited page's code executes.

The injected JavaScript modifies the page's environment to count whenever 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.

3.3.2.1 Measuring Method Calls

The browser extension counted method invocations by overwriting the method on the containing object's prototype. This approach allowed us to shim in logging functionality for each method call, and then call the original method to preserve the original functionality. We replaced each reference to each instrumented method in the Web API with an extension managed, instrumented method.

Our method used JavaScript closures to ensure that web pages were not able to bypass our instrumented methods by looking up–or otherwise directly accessing–the original versions of each method.

3.3.2.2 Measuring Property Writes

Properties were more difficult to instrument. JavaScript provides no clean 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, without affecting the execution of the page. However, through the use of the non-standard `Object.watch()` (47) 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`). The `Object.watch()` method allowed the extension to capture and count all writes to properties on singleton objects in the Web API.

There are a small number of features in the DOM where we were not able to intercept property writes. We could not 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.prototype.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. They are noted here as a small but significant limitation of our measurement technique.

3.3.2.3 Other Browser Features

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

In the case of standard defined 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.

3.3.3 Eliciting Site Functionality

We then measured which browser features were used on the most popular 10k websites with our feature-detecting extension. 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.

3.3.3.1 Default Case

To understand which features are used in a site's execution, we installed the instrumenting extension described in Section 3.3.2. We then 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 code paths will only be used as a result of user interaction, either within the page or by navigating to different areas of the site. Thus, an accurate measurement of feature use requires interacting with and crawling each site. This sub-section describes the strategy used 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 instrumenting a page to click, touch, scroll, and enter text on random elements or locations on the page. To accomplish this, we used a modified version of gremlins.js (48), a library built for monkey testing front-end website interfaces. We modified the gremlins.js library to distinguish between when the gremlins.js script used a feature, and when the visited site used a feature. The former feature use was omitted from further consideration.

Each measurement started by visiting the site’s home page 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 executed a breadth first search of the site using the URLs that would have been visited by the actions of the monkey testing. We selected three 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 three 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 URL s, where the path structure URL had not been previously visited. In contrast to traditional interface fuzzing techniques, which have as a goal finding unintended or malicious functionality (49; 50), we aimed to find just the features site visitors would use. Selecting URLs with different path-segments was a heuristic-based approach 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. This work discusses the robustness and validity of this strategy in Section 3.5.

3.3.3.2 Blocking Extension Measurements

We then repeated the same measurement technique with ad-blocking and tracking-blocking extensions in place (AdBlock Plus and Ghostery, respectively), to generate a second, 'blocking', set of measurements. We treated 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.

3.3.3.3 Automated Crawl

[1]Estimated based on the number of pages visited and 30 seconds of page interaction per visit.

Domains measured	9,733
Total website interaction time	480 days [1]
Web pages visited	2,240,484
Feature invocations recorded	21,511,926,733

TABLE II: AMOUNT OF DATA GATHERED REGARDING JAVASCRIPT FEATURE USE ON THE ALEXA 10K.

For each site in the Alexa 10k, we repeated the above procedure ten times to measure all features used on the page: five times in the default case, and then five times in the blocking case. The crawl took two days, using 64 simultaneous Firefox instances executing on four machines.

Section 3.5 discusses why five crawls per site, per condition, were sufficient to induce all relevant functionality. Table II 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.

3.4 Results

This section presents the results of carrying out the methodology described in Section 3.3 to the Alexa 10k, including measurements of the popularity distribution of JavaScript features with and without blocking, a feature's popularity in relation to its age, and which features are frequently blocked.

3.4.1 Definitions

This chapter uses the term **feature popularity** to denote the percentage of sites that use the 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, the term **standard popularity** denotes 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. Features that are used on roughly the same number of websites in the presence of blocking extensions have low block rates.

3.4.2 Standard Popularity

This subsection presents 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.

3.4.2.1 Overall

Figure 1 displays the cumulative distribution of standard popularity. Some standards are extremely popular, others are extremely unpopular: six standards are used on over 90% of all websites measured, while 28 of the 75 standards measured were used on 1% or fewer sites; eleven are not used at all. The remaining standards saw intermediate popularity levels.

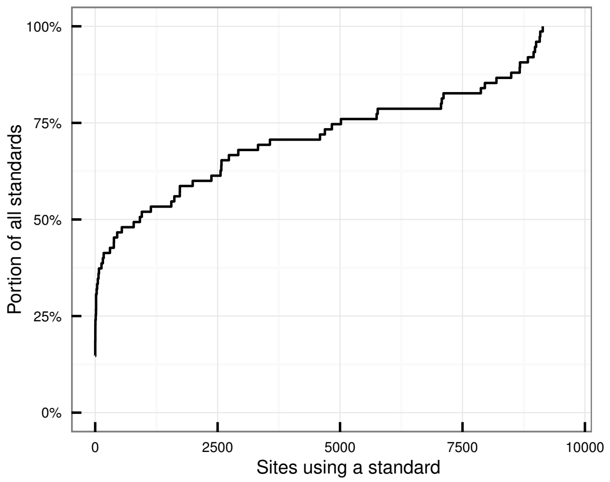

Figure 1: Cumulative distribution of standard popularity within the Alexa 10k.

3.4.2.2 Standard Popularity By Feature

Browser features are not equally used on the web. Some features are extremely popular, such as the `Document.prototype.createElement` method, used 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 measured 1,392, were never observed on the 10k most popular sites. An additional 416 features are used on less than 1% of the 10k most popular websites. This means that over 79% of the features available in the browser are used by less than 1% of the web.

Browser features also 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.

3.4.3 Standard Popularity vs. Site Popularity

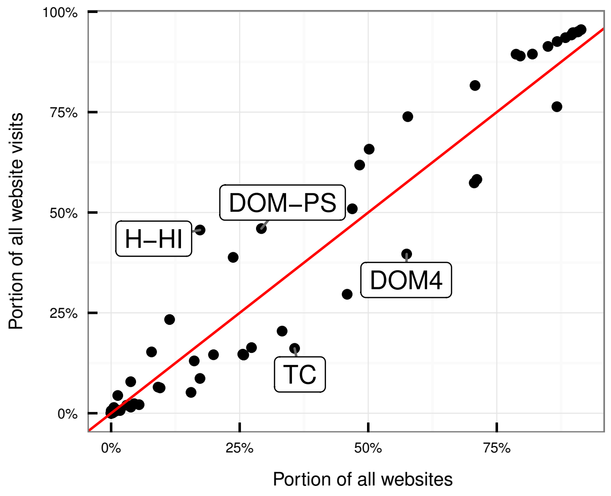

Figure 2: Comparison of percentage of sites using a standard versus percentage of web traffic using a standard.

Most of the results presented in this Section give equal weight to all sites in the Alexa 10k. Put differently, if the most popular and least popular sites use the same standard, both uses are given equal consideration. This Section considers the difference between the number of sites using a standard, and the percentage of site visits using a standard.

Figure 2 presents 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.

The graph, in general, 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 appear to be substantial.

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 also collected data from the less popular sites in the Alexa one-million (sites with rank less than 10k), to determine whether the less-popular websites use different features than popular websites. We found no significant difference between these two groups. The rest of this work, therefore, treats the Alexa 10k as representative of the web in general, as a simplifying assumption.

3.4.4 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 3 indicates, some standards have

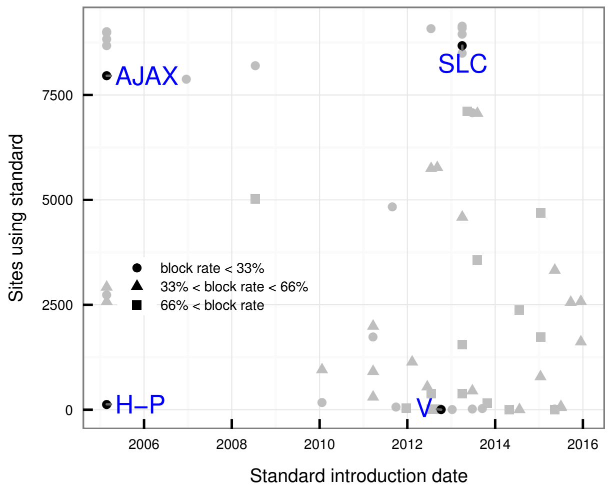

Figure 3: Comparison of a standard's availability date, and its popularity.

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 been adopted by many web authors.

Old, Popular Standards: For example, point **AJAX** depicts the *XMLHttpRequest* (51), 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: Plu-*

gins (52) 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* (53) 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* (54) 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.

3.4.5 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 commonly made by installing browser extensions.

The following sub-sections present 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.

3.4.5.1 Popularity vs. Blocking

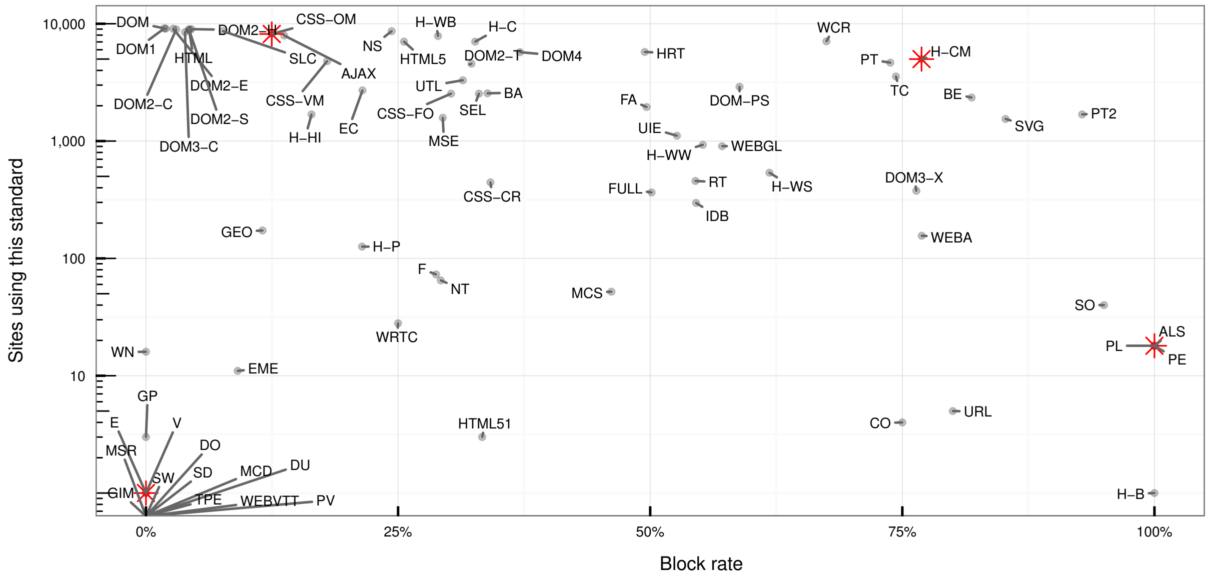

Figure 4: Popularity of standards versus their block rate, on a log scale.

Ad and tracking blocking extensions do not block the use of all standards equally. Figure 4 shows the relationship between a standard's popularity (represented by the number of sites the standard was used on, in log scale) and its block rate. Recall that this work measures a standard's popularity as the number of sites where a feature in a standard is used at least once. Therefore, 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* (55) 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 the measurement. The standard is positioned to the left of the graph, because the standard has a low block rate (12.6%). This means 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* (56) 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 (57), 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* (58) 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* (58) 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.

3.4.5.2 **Blocking Frequency**

As discussed in 3.4.5.1, blocking extensions do not block all browser standard usage equally. Figure 4 shows that some standards are greatly impacted by installing advertising and tracking blocking extensions, while others are not impacted at all.

For example, the *Beacon* (39) 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[1], 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.

3.4.5.3 **Blocking Purpose**

In addition to measuring which standards were blocked by extensions, we also distinguished which extension did the blocking. Figure 5 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).

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

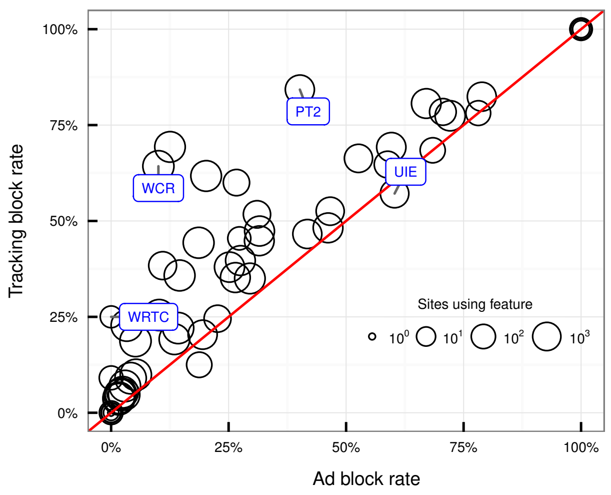

Figure 5: Comparison of block rates of standards using advertising vs. tracking blocking extensions.

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* (59) (which is associated with attacks revealing the user's IP address), *WebCrypto API* (60) (which is used by some analytics libraries to generate identifying nonces), and *Performance Timeline Level 2* (61) (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* (62) standard, which specifies new ways that sites can respond to user interactions.

3.5 Validity

This work measured the features executed over repeated, automated interactions with a website, in a low-trust, unauthenticated browsing scenario. We treat these automated measurements as representative of the features that would be executed when a human visits the website in a similar scenario.

Thus, the work relies on the 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.

3.5.1 Internal Validation

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

TABLE III: AVERAGE NUMBER OF NEW STANDARDS ENCOUNTERED ON EACH SUBSEQUENT AUTOMATED CRAWL OF A DOMAIN.

As discussed in Section 3.3.3.1, 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 performed five measurements 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 were sufficient to capture all potentially encountered features per site, or whether additional measurements would have triggered new standards. To test this, 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 five measurements were insufficient to measure all of the features a site used.

Table III 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 fifth 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 use.

3.5.2 External Validation

We also tested whether our automated technique observed the same feature use as human web users. To do so, we randomly selected 100 sites to visit from the Alexa 10k and interacted with each of them for 90 seconds in a casual web browsing fashion. This included reading articles

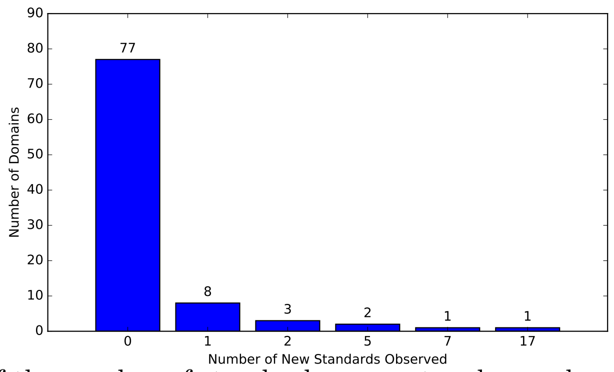

Figure 6: Histogram of the number of standards encountered on a domain under manual interaction that were not encountered under automated interaction.

and blog posts, scrolling through websites, browsing site navigation listings, and generally attempting to exercise what we thought was key functionality on each site.

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 6 is 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 many standards that our automated technique missed. 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 manual evaluation of this site triggered standards not observed during automated testing on the rest of the Alexa 10k, and did not find any.

From this we concluded that our automated measurement technique did a generally accurate job of eliciting 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.

3.6 Conclusions

This chapter presents an automated technique for measuring what parts of the Web API is used in typical low-trust, non-authenticated browsing scenarios, both when using a stock browser, and when using a browser with popular ad and tracking blocking extensions installed. This chapter also presents the results of applying the automated measurement technique to the Alexa 10k.

The most significant results of this work, as it relates to this dissertation's overarching goal of improving privacy and security on the web, are two related insights. First, this work shows there are significant portions of the Web API that are not used when browsing non-trusted websites (over 50%, as measured by Web API features). This strongly suggests that there is little benefit for browser users in enabling these features, at least until users have authenticated with the site, or otherwise established some trust.

Second, this work also documents that there are other parts of the Web API that websites often want to use, but which are blocked by advertising and tracking extensions. This indicates that some Web API functionality is primary used for purposes that the users of those web browsers do not approve of. These frequently-blocked features also seem to provide little benefit to browser users, at least in the case of non-trust, unauthenticated browsing scenarios.

Chapter 4 builds on these findings by measuring the costs and benefits each standard in the Web API carries with it. Chapters 5 and 6 then use these findings to build tools and systems to better protect the privacy and security of web users.

CHAPTER 4

MEASURING FEATURE COST AND BENEFIT

This chapter includes excerpts and figures from a preprint version of material that was later published in *Proceedings of 2017 AMC CCS. Snyder, Peter; Taylor, Cynthia; Kanich, Chris;* The dissertation author was the primary investigator and author of this work.

4.1 Introduction

A second step in improving web security and privacy is to understand the trade-offs each standard in the Web API carries with it. Each feature added to the web platform brings some benefit, by allowing websites to create new types of applications that users may enjoy. Each new feature also carries some cost, in the form of additional security risk. This risk can take a variety of forms, such as bugs in the features implementation, or the feature being exploited to enabling new forms of tracking or privacy loss.

This implies that web users are not best served by browsers with the maximum set of features, but by browsers that only include features where the benefit of doing so outweighs the risks.

This chapter presents a measurement of the costs and benefits each standard brings to web users. This chapter is focused only on a global measurement, or the costs and benefits of enabling the standard in the browser for all websites. The more complicated question of how to deal with the fact that costs and benefits may differ by site (i.e. the risk of allowing `good-`

`folks.org` to access the *Canvas* standard may differ from the risk of allowing `evil-jerks.net` to access the standard) is considered in Chapter 5.

This chapter builds on the feature-use work presented in Chapter 3 by using Web API use measurements as one input to a larger framework for assessing per-standard cost and benefit. After all, how frequently a standard is used on the web is an important signal to understand if a standard is beneficial to users (a Web API standard that is never used is trivially not beneficial to the user), but it is only one part of a more complicated story. A standard, for example, could be used by every site on the internet, but only to carry out functionality the user does not desire. Conversely, the standard could be used by only a small number of websites, but it necessary to carry out functionality that drew the users to the website in the first place. Third, a standard could be *both* very beneficial to users, but expose the web-user to attacks and vulnerabilities so severe that she would still be better off without it. In short, understanding how often a Web API standard is used is just one piece of information needed to assess its impact on the browser.

On the other hand, improving browser privacy and security cannot be the *only* goal to consider when trying to improve web browsers. If one were to set out with so narrow a goal, then she would end up stripping out all functionality from the browser, since, trivially, a browser with no functionality cannot be attacked! This is not likely to be seen by anyone but the most Luddite-minded researcher as an improvement!

The goal then is to identify a subset of the Web API where the benefit to the user outweighs the associated security and privacy risks. This Chapter builds towards that goal by presenting a systematic cost-benefit analysis of each standard in the Web API. By drawing on the measurements described in Chapter 3, this chapter models a standard's benefit as the number of websites in Alexa 10k that require that standard to function correctly. Each standard's costs is modeled in three ways: first, as a function of the number of previously reported Common Vulnerabilities and Exposures Advisory (CVE) reports related to the implementation of the standard, second, as the number of academic papers which leverage the standard to carry out an attack, and third, as a function of complexity added to the code base by implementing the standard.

The rest of this chapter is organized as follows: Section 4.2 presents a technique for removing Web API features from the browser, with a minimal effect on existing code. Section 4.3 describes the full methodology used for measuring the costs and benefits of enabling a Web API standard in the browser. Section 4.4 presents the results of applying the cost-benefit-measurement methodology to a representative modern web browser. Section 4.5 discusses this work's place in the context of the larger goals of this dissertation.

4.2 Intercepting JavaScript Functionality

This section presents a technique for interposing on, and optionally preventing access to, Web API features. The technique is novel in the way it attempts to minimize the effect on existing code that expects the now-removed feature to be present. This approach is used both in building cost-benefit measurements described in this chapter, and in the implementation of the browser-hardening extension described in Chapter 5.

4.2.1 Removing Features from the DOM

Each webpage and iframe gets its own global `window` object. Changes made to this 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 scripts running in the page. If one script, for example, deletes or overwrites the `window.alert` function, 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, earlier code can arbitrarily modify the execution environment seen by later code. Browsers allow extensions to inject JavaScript code into pages and frames before any page controlled code runs[1]. Since 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 a 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 exceptions to be thrown if future code attempts to call the now-removed method.

[1]Section 5.3 discusses in much greater detail how and when extensions can modify page execution environments. This brief discussion is only intended to be enough to support the discussion of how Web API features can be interposed on.

```
1 var ps, p5;
2 ps = document.getElementsByTagName("p");
3 p5 = ps[4];
4 p5.setAttribute("style", "color: red");
5 alert("Success!");
```

Figure 7: Trivial JavaScript code example, changing the color of the text in a paragraph.

Consider the code in Figure 7: removing the `window.document.getElementsByTagName` method will cause an error on line one, as the code is attempting to call the now-missing property as if it were a function. Replacing `getElementsByTagName` with a new, empty function solves the problem on line one, but only pushes the error to 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 depend on the properties of the return value, and fail when expectations are not met.

4.2.2 ES6 Proxy Configuration

Our technique solves this problem through a novel use of a new capability introduced in ES6, the `Proxy` object. The `Proxy` object can intercept operations and optionally pass them along to another object. Relevant to this work, proxy objects also allow code to trap on general language-operations. Proxies can register functions 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 in two ways. First, we use it to prevent websites from accessing browser features, without breaking existing code. This use case is described in detail in Subsection 4.2.3. Second, we use the `Proxy` object to enforce policies on runtime created objects. This use case is described in further detail in Subsection 4.2.4.

4.2.3 Proxy-Based Approach

Our technique uses the `Proxy` object to solve the general problem demonstrated in Section 4.2.1. We create a specially configured proxy object that registers callback functions for *all* possible JavaScript operations, and have 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 having the proxy coerce 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.

Returning to the example in Figure 7, replacing `getElementsByTagName` with our proxy 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 `p5`. Calling the `setAttribute` method 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 the original `getElementsByTagName` functionality.

The complete proxy-based approach to graceful degradation can be found in the source code of our browser extension[1], which is discussed in detail in Chapter 5.

Most state changing features in the browser are implemented through methods which we interpose on using the above described method. However, this approach does not work for the small number of features implemented through property sets. For example, assigning a string to `document.location` redirects the browser to the URL represented by the string. When the property is being set on a singleton object in the browser, as is the case with the `document` object, we interpose on property sets by assigning a new "set" function for the property on the singleton using `Object.defineProperty`.

4.2.4 Sets on Non-Singleton Objects

A different approach is needed for property sets on non-singleton objects. Property sets on Web API defined objects can be imposed on by using `Object.defineProperty` to overwrite the "get" and "set" attributes of the property. However, this approach does not allow for capturing the value being set in the "set" case. Therefore, our approach only allows for blocking sets on non-singleton, Web API defined objects. It can't be used to make more general, runtime policy decisions, where decision logic would need to be made at execution time on whether to block or allow the "set" operation.

[1] `https://github.com/snyderp/web-api-manager`

4.3 Methodology

This section presents a general methodology for measuring the costs and benefits of allowing a website to access a Web API standard. We measure per-standard benefit by using the described feature degradation technique to block page access to the standard, manually browsing sites that use standard, and observing the result. We measure the per-standard cost 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 the lines of code needed to implement that standard.

4.3.1 Representative Browser Selection

This section describes a general methodology for evaluating the cost and benefit of enabling a Web API standard in web browsers, and then applies that general approach to a specific browser, **Firefox 43.0.1**. We use this browser to represent modern web browsers generally for several reasons.

First, Firefox's implementation of Web API standards is representative of how Web API standards are implemented in other popular web browsers (e.g. Chrome, Edge, Safari). These browsers use WebIDL to define Web API interfaces, and implement the underlying functionality mostly in C++, with some newer standards implemented in JavaScript. Many modern browsers even share significant amount of code, both through shared third party libraries and by explicitly copying code from each other's projects (for example, very large portions of Mozilla's WebRTC implementation is taken or shared with the Chromium project in the form of the "webrtc" and "libjingle" libraries).

Second, the standardized nature of the Web API means that measures of Web API costs and benefits performed against one browser will roughly generalize to all modern browsers; features that are frequently used in one browser will be as popular when using any other recent browser. Similarly, most of the attacks documented in academic literature exploit functionality that is operating as specified in these cross-browser standards, making it further likely that this category of security issue will generalize to all browsers.

Third, using Firefox allows for a direct comparison with the measurements discussed in Chapter 3, which were taken with a similar version of Firefox. It also allows for drawing on other related research conducted on Firefox (e.g. (63)).

Finally, we stress that the approach described in this chapter would work with any modern browser; the discussed techniques are not tied to Firefox 43.0.1.

4.3.2 Measuring by Standard

To measure the costs and benefits of the Web API, we first identified a large, representative set of browser features implemented across all modern web browsers. We extracted the 1,392 standardized Web API features implemented in Firefox, and categorized those features into 74 Web API standards, using the same technique as described in Section 3.2.2.

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. This again mirrors the set of features described in Section 3.2.3.

We use these 74 standards as our unit of Web API 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, standards are organized around high level cohesive purposes, which are easier to convey to users who might be interested in blocking parts of the Web API. However, the decision to focus on standards also came with some drawbacks, some of which are discussed later in Section 5.3.2.

4.3.3 Determining When a Website Needs a Feature

This chapter models how beneficial each Web API standard is by measuring how many websites require the standard to function. Enabling features that sites do not use provides little benefit to users. These measurements focus on low-trust, unauthenticated, casual browsing scenarios, and do not attempt to capture more app-like experiences, like video chat or rich user to user messaging.

We focus on this casual browsing scenario because it closely matches the situations where users need to be most careful: when users first visit a new site, and have little basis to judge the site's trustworthiness. Users can only gauge whether to trust a site with greater capabilities once they have some familiarity with it.

Determining if a website needs a feature to function is difficult. If a website does not use a feature, the site trivially does not need the feature to run correctly. As established in Chapter 3, 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 a user's goals on the site. In many cases website will still function as desired (from the perspective of the user), even after the site is prevented from accessing the functionality the site desires. For example, a blog may use the *Canvas* standard to invisibly fingerprint the visitor. If a visitor's browser prevents the site from using the *Canvas* functionality, the visitor will still be able to read the desired postings on the blog, even though the fingerprinting attempt will fail.

This measure of "need" is intentionally focused on the *the perspective of the browser user*. The usefulness of a feature to a website author is not considered beyond the ability of the site author to deliver a user-experience to the 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 to evaluate whether a website needs a browser standard to function. We instructed two undergraduate workers to visit the same website, twice in a row. Each first visit was conducted in an unmodified Firefox browser and treated as the control condition. 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 (64).) On a news site this might mean skimming articles or watching videos, while on an e-commerce sites it might mean searching for products and adding them to the cart.

The worker then visits the same site a second time. This time, the worker's browser is modified to disable all of the features in a Web API standard, using the technique described in Section 4.2. For another minute, the worker attempts to perform the same actions they did during the first visit. They then assign a score to their experience on the site: **1** if there was no perceptible difference between the control and treatment conditions, **2** if they noticed differences, but were still able to complete the same tasks as during the control visit, or **3** if they were not able to complete the same tasks as during the control visit.

We treated a site as broken if the user could not accomplish their intended task (i.e., the visit was coded as a **3**). To account for the inherent subjectivity in this approach, we had both workers test the same sites, independently, and record their score without knowledge of the other's experience. Our workers averaged a 96.74% agreement ratio. This high agreement validates that the workers were able to consistently gauge whether Web API standards were necessary during casual web browsing.

4.3.4 Determining Per-Standard Benefit

We used the above described methodology to determine the benefit of each Web API standards in four steps.

First, we selected 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 sampled 40 sites from the Alexa 10k that use the standard, using the measurements from Section 3. Where there were less than 40 sites using the standard, we selected all using sites. Because there are only small differences between the Web API use of popular and unpopular sites, as described in Section 3.4.3, we made the simplifying assumption to treat these randomly sampled 40 sites using the standard from the Alexa 10k as representative of all sites on the web using the standard.

Third, we used the technique described in Section 4.2 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 4.3.3. We carried out the above process twice for each of the 1,679 sites tested for this purpose, and for each of the 74 Web API standards. This yielded the **site break rate** for each Web API standard, defined as the percentage of times the site broke with the feature 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 4.4.

4.3.5 Determining Per-Standard Cost

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

The first cost metric for enabling a Web API standard is as a function of reported CVEs against the standard's implementation. Past CVEs are an indicator of present risk for three reasons. First, multiple past vulnerabilities indicate that the problem domain addressed by this code is difficult to code securely. These code areas therefor deserve heightened scrutiny, and carry additional risk. Second, prior research (65; 66) found that bug fixes introduce nearly as many bugs as they address, suggesting that code that has been previously patched carries heightened risk for future vulnerabilities. Third, recent industry practices suggest that project maintainers assess security risk similarly; that codebases with many past vulnerabilities should be treated with increased caution (67).

Second, we measure a standard's cost as a function of how many recent academic works document security and privacy issues in the standard. We searched for attacks leveraging each Web API standard in security conferences and journals between 2010 and 2015 (i.e. the five years preceding when this work was conducted).

Third, we measure a Web API standard 's cost by the number of lines of code needed solely to implement the standard in the browser. We base this metric on previous research that found that code complexity (measured as the number of lines of code in function definitions) has had moderate predictive power for discovering where future vulnerabilities will occur in the Firefox codebase (63).

4.3.5.1 CVEs

We determined the number of CVEs previously associated with each Web API standard in three steps:

First, we searched the MITRE CVE database for all references to Firefox in CVEs issued between 2010 and 2016, resulting in 1,554 CVE records.

Second, we reviewed these CVEs and discarded 41 CVEs that were predominantly about other pieces of software, where the browser was only incidentally related (e.g. the Adobe Flash Player plugin (68), or vulnerabilities in web sites that are exploitable through Firefox (69)).

Third, 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. This step's goal was to exclude vulnerabilities relating to non-Web API parts 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 document vulnerabilities in multiple standards.

We identified which Web API standard a CVE related to by reading the text description of each CVE. We attributed CVEs to Web API 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 uniquely related to a larger standard.
- 21 (12%) CVEs named a C++ class or method uniquely related to the implementation of a Web API standard, using the methodology described in Section 4.3.5.2.
- 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 (43)).

We were careful to distinguish CVEs associated with Web API functionality from CVEs associated with lower level functionality. This was done to narrowly measure the cost of *only* the Web API implementation of the standard. For example, the SVG standard (1) allows site authors to use JavaScript to manipulate SVG documents embedded in websites. We counted CVEs like `CVE-2011-2363` (70), a "use-after-free vulnerability" in Firefox's implementation of the JavaScript function for manipulating SVG documents, as part of the cost of including the

SVG Web API standard in Firefox. In contrast, CVEs relating to non-Web API aspects of SVG handing, were excluded from our measurements. For example, `CVE-2015-0818` (71), a privilege escalation bug in Firefox's SVG rendering, is not related to the Web API[1], and so is not counted in these measurements.

4.3.5.2 Implementation Complexity

The third method used to evaluate the per-standard cost relates to how much complexity the standard's implementation adds to the code base. To measure this, we performed a static analysis of the Firefox source to generate lower-bound code-complexity approximations, as a count of significant lines of C/C++ code. This measurement rests on the intuition that more complex implementations carry greater security-costs.

This measurement considers lines of C/C++ code used *only* to implement the measured Web API standard. Lines of code shared between multiple standards are ignored in this measurement. We call this metric Effective Lines of Code (ELoC). We compute the ELoC for each Web API standard in three steps.

First, web built a call graph of Firefox using Mozilla's DXR tool (72). DXR has two related functions. First, DXR includes a clang compiler plugin that builds a call graph of C/C++ code bases. Second, DXR provides tools for querying the call graph, most significantly a web

[1]SVG documents can be rendered statically, without JavaScript interaction.

application.[1] We used this call graph to understand how code paths depend on each other. We modified DXR to record the number of lines of code for each function.

Second, we determined the entry points for the standard in the call graph. Each property, method or interface defined by a Web API standard has two categories of underlying code in Firefox code, **implementation code** (hand written code that implements 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. By mapping each feature in each WebIDL document to a Web API standard, we are able to associate each binding code function with a Web API standard.

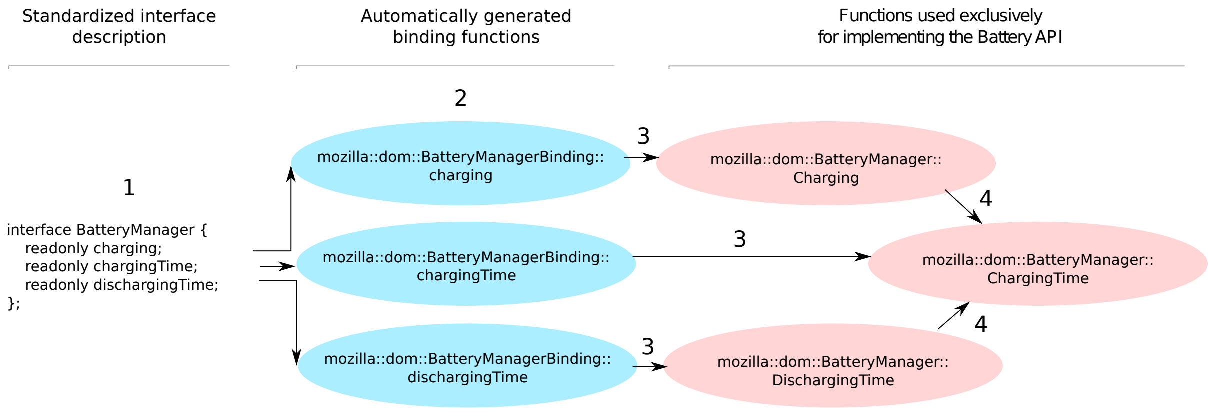
Standardized interface description
Automatically generated binding functions
Functions used exclusively for implementing the Battery API
1
interface BatteryManager {
readonly charging;
readonly chargingTime;
readonly dischargingTime;
};
2
mozilla::dom::BatteryManagerBinding:: charging
mozilla::dom::BatteryManagerBinding:: chargingTime
mozilla::dom::BatteryManagerBinding:: dischargingTime
3
3
3
mozilla::dom::BatteryManager:: Charging
mozilla::dom::BatteryManager:: ChargingTime
mozilla::dom::BatteryManager:: DischargingTime
4
4

Figure 8: An example of applying the graph pruning algorithm to a simplified version of the *Battery API*.

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

Once we determined the call graph entry points for each Web API feature, we used a recursive graph algorithm to identify the implementation code for each standard. Figure 8 illustrates this approach. First, we programmatically extract the standard's definitions for its binding functions, here represented with a a simplified version of the *Battery API*. Second, we located the build-time generated binding code the Firefox call graph, here denoted by blue nodes. Third, using the call graph, we identified which implementation functions the binding functions call into, which are denoted by pink nodes. If an implementation-code (pink) node had only a single incoming edge, we determined that this method / function was only in the code because of the Web API standard associated with those binding functions.

In the Figure 8 example, the algorithm begins by finding that the only code in the Firefox code base that calls the `Charging` and `DischargingTime` methods are the binding functions generated by the *Battery API* standard. The algorithm then marks these methods as uniquely related to the `Battery API`. The algorithm then repeats, again looking for nodes with only callers from known `Battery API` methods. On the second iteration, the algorithm identifes the `ChargingTime` method as 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*.

4.3.5.3 Third Party Libraries

The ELoC algorithm gives a precise lower bound measurement of the lines of code *in the Firefox source* included only to implement a given Web API standard. It does not include code from third-party libraries, which are compiled as a separate step in the Firefox build process, and thus excluded from DXR's call-graph.

This limitation was not significant in practice. In nearly all cases, third party libraries are used in multiple places in the Firefox codebase and cannot be uniquely attributed to any single standard, and thus are not relevant to our per-standard ELoC counts.

The sole exception is the *WebRTC* standard, which uniquely uses over 500k lines of third party code. While this undercount is large, it too 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.

4.4 Results

This section presents the results of applying the methodology discussed in Section 4.3 to Firefox 43.0.1. The section first describes each Web API standard 's benefit, and follows with each standard's cost.

4.4.1 Per-Standard Benefit

As explained in Section 4.3.4, our workers conducted up to 40 measurements of websites in the Alexa 10k known to use each 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.

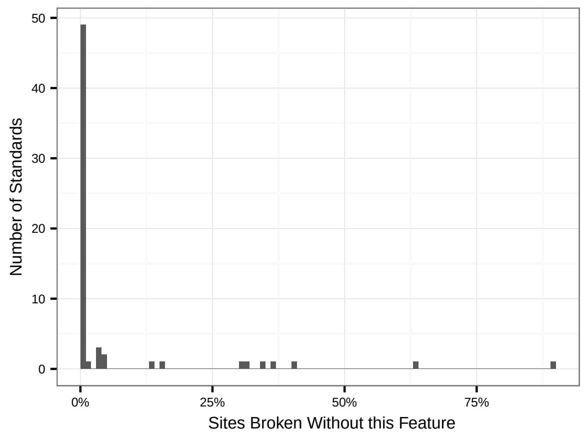

Figure 9: A histogram giving the number of standards binned by the percentage of sites that broke when removing the standard.

Figure 9 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.

In some cases, this was because the standard was never observed being used[1]. In other cases, it was because the standard is intended to be used in a way that users do not notice[2].

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.

[1]e.g. The *WebVTT* standard, which allows document authors to synchronize text changes with media playing on the page.

[2]e.g. The *Beacon* standard, which allows content authors to trigger code execution when a user browses away from a website.

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

For emphasis, we note again that these measurements only cover interacting with websites as an anonymous, unauthenticated user. It is possible that site feature use changes when users log into websites, since some sites only provide full functionality to registered users.

4.4.2 Per-Standard Cost

As described in Section 4.3.5, we measured the cost of a Web API standard being available in the browser in three ways: first as the number of times the standard is leveraged by attacks in high quality peer-reviewed research (Section 4.4.2.1), second as the number of CVEs reported against the standard between 2010 and 2015 (Section 4.4.2.2), and third with a lower bound estimate of the number of ELoC needed to implement the standard in the browser (Section 4.4.2.3).

4.4.2.1 Attacks from Related Research

We searched the work published at major research conferences and journals between 2010 and 2015 for research on browser weaknesses related to Web API standards. These papers either explicitly identified either a Web API standard, or a feature or functionality uniquely related to a Web API standard. In each case the standard was either necessary for the documented attack to succeed, or was used to make the attack faster or more reliable. Since academic attacks emphasize attack novelty, instead of only finding all existing vulnerabilities, the use of a Web API standard in academic literature suggests that the Web API standard allowed new browser exploits.

The most frequently cited standard was the *High Resolution Time Level 2* (94) 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 (75; 87; 88), learning information about other open browser windows (86; 79; 88), and gaining identifying information from other domains (30).

Other implicated standards include the *Canvas* standard, which was identified by researchers as allowing attackers to persistently track users across websites (34), learn about the browser's execution environment (75) or obtain information from other browsing windows (79), and the *Media Capture and Streams* standard, which was used by researchers to perform "cross-site request forgery, history sniffing, and information stealing" attacks (83).

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.

4.4.2.2 CVEs

Vulnerability reports are not evenly distributed across browser standards. Figure 11 compares per-standard benefit (measured by the number of sites that require the standard to function) on the y-axis, against the number of severe CVEs associated with the standard on the x-axis. Figure 10 shows a similar plot, but includes all CVEs, not only high and severe ones. Both figures show the same general relationship between break rate and CVEs.

Points in the upper-left of each figure denote standards that are high benefit and low cost (i.e. standards that are frequently required on the web but have rarely been implicated in

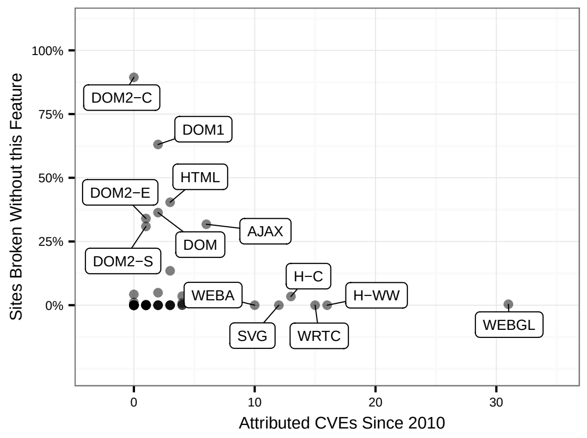

Figure 10: A scatter plot showing the number of CVEs filed against each standard since 2010, by how many sites in the Alexa 10k break when the standard is removed.

CVEs). For example, consider the *Document Object Model (DOM) Level 2 Events Specification* standard, denoted by **DOM2-E** in Figure 11. This standard allows website authors to trigger functionality to occur with common browser events, like button clicks and mouse movement. This standard is highly beneficial to browser users, being required by 34% of pages to function correctly. The standard entails little risk to web users, being associated with zero CVEs since 2010.

Standards in the lower-right section of the figures, by contrast, bring low benefit and high cost to users, when using CVE counts as a metric for security cost. For example, the *WebGL* standard, denoted by **WEBGL** in Figure 11, allows websites to take advantage of graphics hardware on the browsing device. Less than 1% of sites in the Alexa 10k need the standard, but the standard is implicated in 22 high or severe CVEs since 2010. This suggests that the standard poses a high security risk to users, with little attenuating benefit.

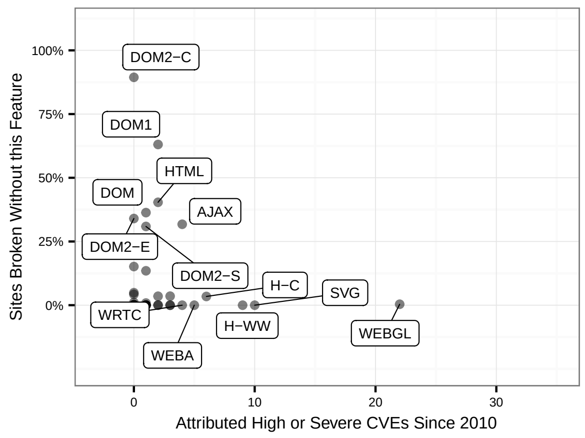

Figure 11: A scatter plot showing the number of “high” or “severe” CVEs filed against each standard since 2010, by how many sites in the Alexa 10k break when the standard is removed.

4.4.2.3 Implementation Complexity

The amount of complexity each standard added to the browser code base varied widely. Figure 12 presents a comparison of each standard’s benefit (measured by the number of sites that require the standard to function) against and number of lines of code uniquely needed to implement the standard, using the method described in Section 4.4.2.3.

Points in the upper-left of Figure 12 depict standards that are frequently needed on the web, but which have relatively non-complex implementations. One example of such a standard is the *DOM-Level 2 Core* standard, denoted by **DOM2-C**. This standard extends the browser’s basic document modification methods. This standard is needed for 89% of websites to function correctly, suggesting it is highly beneficial to web users. The standard comes with a low security cost; our technique identifies only 225 lines of code that are only included to enable

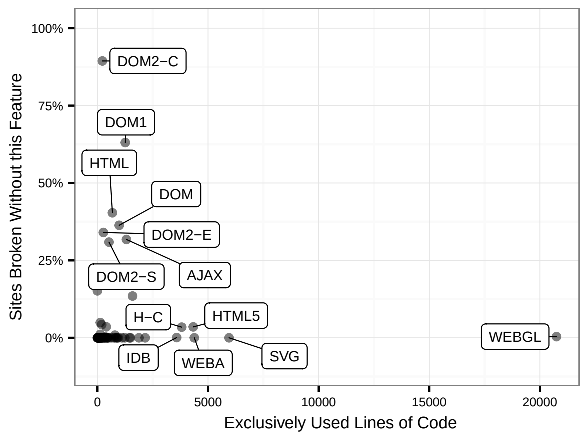

Figure 12: A scatter plot showing the LOC measured to implement each standard, by how many sites in the Alexa 10k break when the standard is removed.

this standard (most of the code that implements this standard is shared by the implementations of other standards).

Points in the lower-right of the figure depict standards that provide little benefit, but which are responsible for a great deal of complexity in the browser's code base. The *Scalable Vector Graphics* standard, denoted by **SVG**, is an example of such a high-cost, low-benefit standard. The standard 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).

4.4.3 Threats to Validity

The main threat to validity in these measurements is the accuracy of our human-executed casual browsing scenario. Regarding internal validity, the high agreement between the two site-measuring workers suggests that our technique was constant and replicable. 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. Visiting a website for 70 or fewer seconds encapsulates 80% of all web page visits according to (64), thus accurately representing a majority of web browsing activity, especially when visiting untrusted websites. Furthermore, while our experiment does not evaluate functionality that is only available to authenticated users, we believe 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 that deserve the most caution.

4.5 Conclusions

This chapter presented a systematic evaluation of the costs and benefits that each standard in the Web API brings to browser users in low-trust, non-authenticated situations. This work measured the benefit of each standard as a function of the number of sites in the Alexa 10k that require the standard to carry out the site's main purpose (from the perspective of the browser user). The work measured the cost of each standard in three ways: first, as the number of peer-reviewed papers in top security conferences and journals that level the standard in an attack, second, as the amount of complexity that the standard's implementation adds to the code base, and third, as the number of vulnerabilities the standard's implementation is responsible for.

The significance of this chapter to this dissertation's overarching goal of improving privacy and security on the web, is two related insights. First, this work suggests a significant subset of functionality in the Web API where the cost to users (in terms of security risk) is much higher than the benefit (in terms of sites that need the functionality to do what users care about). This, in turn, suggests that web privacy and security could be improved by restricting which sites can access these low-benefit, high-risk features.

Second, this work documents that a small number of standards in the Web API provide the majority of the benefit to browser users. These same standards, with a few exceptions, carry very little security risk. This suggests that an application system based around just these safe, core-web features would provide users with most of the benefits of modern web applications, with significant security and privacy improvements.

The next two chapters follow these insights to build tools and systems that improve the privacy and security of web users. Chapter 5 pursues the first insight by exploring methods to restrict which parts of the Web API that existing web applications can access. Chapter 6 builds on the second insight by exploring new ways of developing and deploying web applications that restrict websites to core web functionality, and emphasize privacy and security over developer flexibility and application functionality.

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	(73; 74; 75; 76)
HTML: Web Workers	H-WW	856	0%	100%	16	9	1.63	(75; 77)
WebRTC	WRTC	24	0%	93%	15	4	2.48	(78; 74)
HTML: The canvas element	H-C	6935	0%	100%	14	6	5.03	(78; 73; 74; 34; 79; 75; 76)
Scalable Vector Graphics	SVG	1516	0%	98%	13	10	7.86	
Web Audio API	WEBA	148	0%	100%	10	5	5.79	(78; 74)
XMLHttpRequest	AJAX	7806	32%	82%	11	4	1.73	
HTML	HTML	8939	40%	85%	6	2	0.89	(35; 80)
HTML 5	HTML5	6882	4%	97%	5	2	5.72	
Service Workers	SW	0	0%	-	5	0	2.84	(81; 82; 30)
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	(74; 34)
Web Cryptography API	WCR	7048	4%	90%	4	3	0.52	
Media Capture and Streams	MCS	49	0%	95%	4	3	1.08	(83)
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	(81; 82; 30)
CSS Object Model	CSS-OM	8094	5%	94%	1	0	0.17	(35)
DOM	DOM	9050	36%	94%	1	1	1.29	
HTML: Plugins	H-P	92	0%	100%	1	1	0.98	(74; 80)
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	(84; 85)
High Resolution Time Level 2	HRT	5665	0%	100%	1	0	0.02	(82; 86; 87; 30; 79; 75; 88; 77)
HTML: Channel Messaging	H-CM	4964	0%	0.025	1	0	0.40	(89; 90)
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	(75)
CSSOM View Module	CSS-VM	4538	0%	100%	0	0	2.85	(80)
Battery Status API	BA	2317	0%	100%	0	0	0.15	(78; 74; 35; 91)
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	(74; 80)
DeviceOrientation Event	DO	0	0%	-	0	0	0.06	(92; 74)
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	(74; 84; 75)
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	(35)
Ambient Light Sensor API	ALS	18	0%	89%	0	0	0.00	(35; 93)

TABLE IV: DATA ON ALL 74 MEASURED Web API standards, EXCLUDING 20 STANDARDS WITH A 0% BREAK RATE, 0 ASSOCIATED CVES AND ACCOUNTING FOR LESS THAN ONE PERCENT OF MEASURED EFFECTIVE LINES OF CODE.

- The standard's full name
- The abbreviation used when referencing this standard in this work
- The number of sites in the Alexa 10k using the standard (Section 3.4)
- The portion of measured sites that were broken by disabling the standard. (Section 4.3.4)
- The agreement between testers' evaluation (Section 4.3.4)
- The number of CVEs since 2010 associated with the feature (Section 4.4.2.2)
- The number of CVEs since 2010 ranked as "high" or "severe" (Section 4.4.2.2)
- The percentage of ELoC for this standard, as a percentage of all attributed lines (Section 4.3.5.2)
- Citations for papers describing attacks relying on the standard (Section 4.4.2.1)

CHAPTER 5

RETROFITTING FEATURE-LEVEL ACCESS CONTROLS ON THE WEB

This chapter includes excerpts and figures from a preprint version of material that was later published in *Proceedings of 2017 AMC CCS. Snyder, Peter; Taylor, Cynthia; Kanich, Chris;* The dissertation author was the primary investigator and author of this work.

5.1 Introduction

The third step towards this dissertation's over-arching goal of improving web security and privacy is to consider how the per-standard cost and benefit measurements described in Chapter 4 can be used to improve the web as it exists today. More specifically, it is to explore how the data and techniques discussed in the previous chapters can be used to better protect web users, given how websites are currently deployed.

This chapter describes one such effort, through the design and development of a browser-extension that imposes access controls on which Web API features websites can access. The tool's goal is to modify the DOM to prevent websites from accessing features that risk user's security and privacy. The extension attempts to limit websites to a minimal set of features, judged to be high benefit and low cost.

This chapter discusses the design of this tool, a usability evaluation of the tool, and a discovered vulnerability the Firefox and Chrome implementations of the WebExtension standard, which significantly limits the security and privacy guarantees our extension can enforce. This vulnerability affects many other privacy-oriented WebExtensions, and most of these tools are still vulnerable at the time this work is being written.

The rest of this chapter is organized as follows: Section 5.2 describes the design of a browser extension based on the findings discussed in Chapter 4, along with a usability analysis of the extension. Section 5.3 presents issues and discoveries that were made after the browser extension was released to the public and began being used by approximately 1k users.

5.2 Browser Hardening Extension

This section presents efforts at using the cost-benefit measurements from Section 4 to design and develop a browser extension that allows users to control website access to Web API functionality. The extension has been released to the public and is being used by almost 1k users at the time of this writing (555 Firefox users[1] and 417 Chrome users[2]). This section describes how the extension was designed, evaluated and implemented before it was publicly released. Section 5.3 discusses lessons and findings that were gained only once the extension was available for the public.

The source of the extension has been released publicly[3], and the project continues to be refined with the help and input of other developers and privacy activists.

---

[1]`https://addons.mozilla.org/en-US/firefox/addon/webapi-manager/`

[2]`https://chrome.google.com/webstore/detail/webapi-manager/hojoljbhkebfjalcbnfmoiljfidcmcmj`

[3]`https://github.com/snyderp/web-api-manager`

5.2.1 Implementation

Our browser extension uses the same Web API standard disabling technique described in Section 4.2 to dynamically control the Web API functionality exposed to websites. The extension allows users to apply hardened configurations that we designed (based on the findings discussed in Section 4 and detailed in Section 5.2.3.1), or design and deploy their own hardened configurations by selecting any permutation of the measured Web API standards to disable (along with several additional Web API standards that were deployed since the cost-benefit measurements were performed).

We emphasize the dynamic nature of the hardened browser configurations for several reasons. First, if a given standard was 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.

5.2.2 Trade-offs and Limitations

Deploying our approach as a browser extension entailed significant trade-offs. On the benefit side, browser extensions are easy for users to install and update. Our browser extension is compatible with current popular web browsers, minimizing the amount of additional engineering work needed to get the approach implemented and usable by security and privacy concerned users. Additionally, browser extensions are powerful enough to (in principal) successfully protect users from most vulnerabilities that depend on accessing a JavaScript-exposed method or data structure (of which there are many, as documented in Section 4.4.2.2). The WebExtension standard, which standardizes a common interface for writing cross-browser extensions, defines hooks that allow for disabling large portions of high-risk functionality, which could protect users from not-yet-discovered bugs, in a way that ad-hoc fixing of known bugs could not.

However, the choice to deploy as an extension also entails significant limitations. First, there are categories of browser exploits that our extension-based approach cannot guard against. For example, our approach cannot provide protection against exploits that rely on Web API functionality that is reachable through means other than JavaScript. The extension would not provide protection against, for example, exploits in the browser's CSS parser, TLS code, or image parsers (since the attacker would not require JavaScript to access such code-paths).

Second, the choice to implement as a browser extension made our approach vulnerable to a weakness common to all DOM-modifying browser extensions privacy and security tools that was discovered in this work (discussed in detail in Section 5.3).

Third, the extension approach does not have access to some information that could be used to make more sophisticated decisions about when to allow a website to access a feature. An alternate approach that modified the browser could use factors such as the state of the stack at call time (e.g. distinguishing between first-and-third party calls to a Web API standard), or where a function was defined (e.g. whether a function was defined in JavaScript code delivered over TLS from a trusted website). Because such information is not exposed through JavaScript, our extension is not able to take advantage of such information.

5.2.3 Usability Evaluation

This section presents an evaluation of the usability of our Web API-standard blocking extension. The goal of this evaluation was to measure how the extension both improved and harmed users' browsing experiences, to see if the extension's cost (measured in "number of sites that no longer function correctly") would be worth the extension's benefits (measured as "security risk reduction from blocking Web API standards").

The extension allows users to develop custom, per-site configurations of which Web API standards to block. Since there were 74 standards considered in Section 4, there are 2^{74} possible configurations of the extension, yielding an impossible number of configurations to test for even a single site, let alone a large enough sample of websites to represent the web as a whole.

Instead, we created and evaluated two plausible extension configurations, based on the data described in Section 4, to represent users with different privacy needs, and thus users willing to accept different security/usability trade-offs. The following subsections present an analysis of the usability of these two selected configurations.

5.2.3.1 Selecting Configurations

To address the combinatorial impossibility of evaluating all possible extension configurations on all sites, we created two configurations to represent configurations that users might create. We refer to these configurations as the **conservative** and **aggressive** configurations, each intending to represent users with different privacy-functionality trade-offs.

Table V lists the standards we blocked for the conservative and aggressive hardened browser configurations. Our **conservative** configuration focuses on removing features that are infrequently needed by websites to function, and would be fitting for users who desire more security than is typical of a commodity web browser, and are tolerant of a slight loss of functionality. Our **aggressive** configuration focuses on removing further attack surface from the browser, even when that necessitates breaking more websites. This configuration would fit highly security sensitive environments, where users are willing to accept breaking a higher percentage of websites in order to gain further security.

We selected these profiles based on the data discussed in Section 3.4, Section 4.4, and prioritizing not affecting the functionality of the most popular sites on the web. We further chose to not restrict the *Web Crypto* standard, to avoid affecting the possibly-security-sensitive code.

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 trade-off that best fits their needs.

5.2.3.2 Configuration Evaluation

We evaluated the usability and the security gains the hardened browser configurations provided. Table VI 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 4.3.4. 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 ranked 201 or lower). This yielded 175 test sites in the popular category, and 155 in the less popular category.

Second, 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 (64) finding that users on average spend under a minute per page.

These 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. Finally, they visited 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 1–3 scale described in Section 4.3.4. 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 VI give the results of this measurement.

To further increase our confidence in the reported site-break rates, our evaluators recorded, in text, what broken functionality they encountered. We then randomly sampled and checked these textual descriptions, to verify 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 VI shows, the trade off between gained security and lessened usability is non-linear. The conservative configuration disabled code paths associated with 52% of previous CVEs, and removed 50% of ELoC, while affecting the functionally of only 3.87%-7.14% of sites on the internet. Similarly, the aggressive configuration disabled 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.

5.2.3.3 Usability Comparison

We compared the usability of our conservative and aggressive configurations against Tor Browser Bundle (TBB) and NoScript, to measure how the Web API-blocking approach compared to other popular security and privacy tools. We found that the conservative configuration had the highest usability of all four tested tools, and that the aggressive hardened configuration was roughly comparable to the default configuration of the TBB. The results of this comparison are given in Table VII.

This comparison does not 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, since TBB and NoScript are widely used security tools, comparing against them gives a good baseline for usability for security conscious users.

We tested the usability using the same technique we used for the conservative and aggressive browser configurations, described in Section 5.2.3.1. 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 TBB[1]. 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 the lower agreement score for the TBB 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.[2]

[1]Smaller sample sizes were used when evaluating TBB because of time constraints, not for fundamental methodological reasons.

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

As Table VII 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 blocking approach has some effect on usability, it is a cost security-sensitive users would accept.

5.2.3.4 Allowing Features for Trusted Applications

We further evaluated our approach by attempting to use several popular, complex JavaScript applications in a browser in the **aggressive** hardened configuration. We then created application-specific configurations to allow these applications to run, but with access to only the minimal set of features needed for functionality.

This process of creating site-specific feature configurations is roughly analogous to granting trusted applications additional capabilities (in the context of a permissions-based system), or allowing trusted domains to run JavaScript code (similar to how NoScript functions).

We built these application specific configurations using a tool-assisted, trial and error process: first, we visited the application with the browser extension in “logging” mode, which caused the extension to log blocked functionality. Next, when we encountered a part of the web application that did not function correctly, we reviewed the extension’s log to see what blocked functionality seemed related to the error. We then iteratively enabled the related blocked standards and revisited the application to see if enabling the standard allowed the app to function correctly. We repeated the above steps until the app worked as desired.

This process is would be beyond what typical web users would be capable of, or interested in, doing. Users who were interested in improving the security of their browser, but not interested in creating hardened app configurations themselves, could subscribe to trusted, expert curated policies, similar to how users of AdBlock Plus receive community created rules from EasyList.

For the first application-specific configuration example, we watched videos on YouTube, by first searching for videos on the site's homepage, clicking on a video to watch, watching the video on its specific page, and then expanding the video's display to full-screen. This required enabling three standards that are blocked in our **aggressive** configuration: the *File API* standard[1], the *Media Source Extensions* standard[2], and the *Fullscreen API* standard. Once we enabled these three standards, we were able to search for and watch videos on the site, while still having 39 other standards disabled.

Second, we used the Google Drive application to write and save a text document, formatting the text using the formatting features provided by the website (creating bulleted lists, altering justifications, changing fonts and text sizes, embedding links, etc.). Doing so required enabling two standards that are by default blocked in our **aggressive** configuration: the *HTML: Web Storage* standard[3] and the *UI Events* standard[4]. Allowing Google Docs to access these two

---

[1]YouTube uses methods defined in this standard to create URL strings referring to media on the page.

[2]YouTube uses the `HTMLVideoElement.prototype.getVideoPlaybackQuality` method from this standard to calibrate video quality based on bandwidth.

[3]Google Drive uses functionality from this standard to track user state between pages.

[4]Google Drive uses this standard for finer-grained detection of where the mouse cursor is clicking in the application's interface.

additional standards, but leaving the other 40 standards disabled, allowed us to create rich text documents without any user-noticeable affect in site functionality.

Third and finally, we used the Google Maps application to map a route between Chicago and New York. We did so by first searching for “Chicago, IL”, allowing the map to zoom in on the city, clicking the “Directions” button, searching for “New York, NY”, and then selecting the “driving directions” option. Once we enabled the *HTML: Channel Messaging* standard[1] we were able to use the site as normal.

5.3 Real-World Extension Deployment

The previous section described the design of the Web API blocking extension, and how that design was based on the cost-benefit measurements from Chapter 4. This section describes discoveries that were made once the extension was released to the public, and further developed with the help of other privacy and security focused developers.

5.3.1 Vulnerability in WebExtension Implementations

While working on an issue that was reported against the blocking extension, we discovered a security-related vulnerability in the WebExtension implementations in Firefox and Chrome. This vulnerability allows determined websites to access browser functionality blocked by our extension, and a comprehensive defense against it required modifications to our approach that breaks more websites than we originally accounted for. This same WebExtension weakness also

[1] Which Google Maps uses to enable communication between different sub-parts of the application.

affects other security and privacy improving extensions (e.g. PrivacyBadger, canvas blockers, the "Shields Up" defenses in the Brave browser).

This section provides some background information on what the WebExtension standard is, describes the vulnerability in the most popular implementations of the WebExtension standard we discovered, and how authors of privacy and security enhancing extensions can move forward.

5.3.1.1 The WebExtension Standard

The *WebExtension* (95) standard defines a way to write browser-modifying extensions that run on all major, modern browsers. All of the major browser vendors have pledged support for the standard, but Firefox and Chrome have the most complete and popular implementations. The standard is largely based on the original Chrome Extension API, and is managed by the W3C.

The WebExtension standard allows authors to modify the browsing environment in many ways. Most relevant to our extension is the ability to inject JavaScript into frames, before the frame's own JavaScript has executed. This allows the extension to modify the environment the website executes in. Our extension uses this technique to add access controls to Web API features, using a method described in greater detail in Section 4.2. Many other privacy and security enhancing extensions use this same technique to prevent pages from accessing parts of the Web API, to protect users (e.g. to detect or prevent fingerprinting attempts).

5.3.1.2 How the Vulnerability Works

The vulnerability we discovered allows pages to access unmodified versions of the browser environment, even after the extension's JavaScript has run. The vulnerability works by exploiting how frames interact in Chrome and Firefox, and the timing of when WebExtension's script injection hooks execute.

Frames in HTML Documents: The term **frame** refers to an execution environment in the browser. Each page loaded by the browser gets a frame, with each browser tab depicting a single frame. For example, loading `example.org` in a browser tab will create a single frame, showing the HTML document returned by `example.org`. Opening a second tab and loading `other-example.org` will likewise create a new frame, this time depicting the document returned from `other-example.org`.

Importantly, each frame gets its own DOM instance, each with its own version of each Web API feature. Each Web API feature is implemented by a JavaScript object, with functions represented by objects inheriting from `Function.prototype`. Changing a function in one frame will have no effect on similar objects in other frames. Put differently, deleting the `Document.prototype.getElementById` object in the `example.org` frame will prevent code executing in the `example.org` frame from querying for elements by their `id`, but that change will be invisible to code running in the `other-example.org` frame.

In general, frames are not able to directly access the resources of other frames. In the above example, code running in the `example.org` frame cannot access the DOM of the `other-example.org` frame. However, this rule does not hold in all cases. In some cases, a frame can access the DOM of its child frames (e.g. `iframes`). When the child frame is rendering content from the same domain as the parent frame, the parent frame can access the DOM of the child frame through the child frame's `contentWindow` property.

Injecting Script from a WebExtension: The WebExtension standard provides several opportunities for extensions to inject JavaScript into frames. Relevant to this vulnerability is that scripts can register to run at `loading` time, which corresponds to a point when the DOM for the frame has been prepared, but no page contents have yet executed. The WebExtension standard defines this as the `document_start` hook.

Because the WebExtension standard guarantees that `document_start` will run before any other content is executed in the frame, many extensions use this opportunity to inject script into a frame, to modify the DOM of the frame to achieve some security or privacy improvement. Our extension uses this hook to inject JavaScript that interposes on the features of blocked standards. Since the WebExtension standard guarantees that this will happen before any page content executes, the extension can be sure that the page will only see the DOM as its been modified by the extension, and that page code will not be able to access the original, non-interposed-on versions of the blocked features.

Many other extensions function similarly, and use the `document_start` hook to achieve security or privacy goals. PrivacyBadger, for example, uses this opportunity to replace Web API functions associated with canvas fingerprinting with new functions that record the fingerprinting attempt.

Exploiting the Vulnerability: The vulnerability arises because of how the above two issues (child-frame access and extension code injection timing) interact in common WebExtension implementations. One might expect that because the `document_start` hook runs before page content, then page content will only be able to access the DOM after its been modified. However, this proves to be incorrect. While frames are unable to access *their own* DOM before it is modified by extensions, frames can access the DOM of child frames before the `document_start` hook fires in the child frame.

This occurs because, while the `document_start` hook is guaranteed to fire before a frame's content runs, there is a period of time between when the child frame's DOM is created, and the child's `document_start` hook fires. If a parent frame accesses the child frame's `contentWindow` property during this interim period, the parent frame will be able to access the child frame's DOM before it is modified. The parent frame can then extract references to blocked functionality and execute them in the context of the parent frame. The end result is that pages can bypass extensions that attempt to restrict access to Web API features.

5.3.1.3 Addressing the Vulnerability

We addressed this vulnerability in the WebExtension standard in several ways. First, we filed bugs with both Firefox[1] and Chromium[2], notifying them of the issue. Firefox has acknowledged the issue but so far the issue has not been addressed. The issue is still waiting for triage in Chromium's system.

Second, we notified other similar privacy and security minded projects that use the same WebExtension approach (i.e. Brave and PrivacyBadger) of the issue. In both cases, the develop-

[1] `https://bugzilla.mozilla.org/show_bug.cgi?id=1424176`

[2] `https://bugs.chromium.org/p/chromium/issues/detail?id=793217`

ers acknowledged the issue, but are waiting on action from the browser vendors before pursuing the matter further, largely because currently available solutions break too many websites.

Third, we modified the extension to give users an option to protect themselves against this vulnerability, at the cost of breaking some benign sites. Versions of ECMAScript (the technical name for the standard that defines JavaScript) 5.1 and later define a `Object.defineProperty` (96) function, which can be used to prevent code from reading from and writing to properties on certain types of objects. The extension can use this function to prevent frames from accessing the content of child frames. This prevents websites from bypassing the extension, with the downside of breaking sites that benignly access child frames in this way. While this is not a common technique online (measured by domains that use the technique), some very popular sites use this technique to coordinate across origins (e.g. Google's authentication flow).

At a fundamental level, the correct solution is likely to freeze the event loop of the parent frame until the `document_start` hook of the child frame has fired. This would incur some performance loss (since the parent frame would momentarily freeze), but the cost would be small. It is difficult to think of scenarios when frames need to create and insert large numbers of child frames into documents, particularly in performance-sensitive tasks. However, further exploring and evaluating solutions to this issue is beyond the scope of this dissertation.

5.3.2 Feature-level Granularity

A second insight gained from making the browser extension public, and improving it with other privacy and security-minded developers, is that, in some cases, the "standard" may be the wrong level of analysis when evaluating the Web API. In some cases, users are better served by being able to define finer-grained policies, such as blocking a small number of features in a larger standard. These finer-grained policies allow dedicated users to drive down the number of sites the extension breaks, while still preventing sites from accessing problematic Web API features.

For example, many users of the extension were interested in blocking the *Canvas* standard, since it is often implicated in fingerprinting attacks. While the functionality in the standard is rarely needed to deliver the main content on a site, it is sometimes used when rendering peripheral but pleasant content. Some users wanted to be able to protect themselves against fingerprinting attacks without giving up some of the flashier parts of the sites they visited.

The solution was to allow users the option to block just some features in a standard, but leave the rest of the features unmodified. In the *Canvas* example, the standard includes 53 features, only 4 of which are used for fingerprinting[1]. By blocking methods that allowed for reading from a canvas, but allowing the rest of the standard to function as normal, users were able to better protect themselves against privacy and security violations, while still allowing trusted sites to provide their user-benefiting functionality.

More broadly, by giving the users the option to block at the feature level, instead of the standard level, users were able to push the extension's break rate down without reducing their security.

[1] `HTMLCanvasElement.prototype.toDataURL`, `HTMLCanvasElement.prototype.toBlob`, `CanvasRenderingContext2D.prototype.isPointInPath` and `CanvasRenderingContext2D.prototype.isPointInStro`

5.3.3 Web API Standard Growth

Finally, maintaining the extension has emphasized how quickly the Web API grows, and the need for security and privacy researchers to consider how quickly the web changes as an application platform. Since the research in Chapter 4 was conducted, browsers have implemented partially or fully implemented Web API standards for interacting with VR headsets (97), interacting with USB devices (98), and synthesizing and recognizing speech (99), among many others.

Each of these features, while adding potentially useful new capabilities to the platform, also expands the browser's attack surface and makes the system more complex. More work like that described in Chapter 4 will be needed to understand if the benefit of each new powerful feature is worth the cost.

5.4 Conclusions

This chapter builds on the findings described in Chapters 3 and 4 to build a publicly available tool that allows web users to restrict which parts of the Web API websites can access. This chapter also presents evaluations of the usability of the tool under common usage scenarios, some quantification of the security benefits of using the tool under those scenarios, and some insights that were only gained when the tool was used by unaffiliated users.

The most significant results of this study, as it relates to this dissertation's overarching goal of improving privacy and security on the web, are three insights. First, that restricting website access to the Web API is a realistic and implementable way of protecting the security and privacy of users on the web today, and with trade-offs that real world users are willing to accept. Second, that with relatively minor changes, browser vendors could allow the security and privacy benefits of this Web API blocking technique to be enjoyed at even lower cost. Finally, that standards authors should consider whether all websites need access to new Web API functionality, or if users would be better served by a restrict-by-default, opt-in-when-needed model.

This chapter has focused on applying the findings from Chapters 3 and 4 to the web as it is currently designed. Chapter 6 explores what security and privacy benefits can be achieved by applying these findings to a deeper redesign of how web applications are designed and deployed.

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
WebVTT		
Geometry Interfaces		
Vibration		
WebVR		
WebUSB		
WebSpeech		

TABLE V: 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 VI: COST AND BENEFIT STATISTICS FOR THE EVALUATED CONSERVATIVE AND AGGRESSIVE BROWSER CONFIGURATIONS.

	% 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 VII: COMPARISON OF THE USEABILITY OF THE FEATURE-ACCESS-CONTROL IMPOSING EXTENSION, COMPARED AGAINST VERSUS OTHER POPULAR BROWSER SECURITY TOOLS.

CHAPTER 6

TOWARDS MORE SECURE WEB APPLICATIONS

This chapter includes excerpts and figures from a preprint version of material that was later published in *Proceedings of 2017 ConPro. Snyder, Peter; Watiker, Laura; Taylor, Cynthia; Kanich, Chris;* The dissertation author was the primary investigator and author of this work.

6.1 Introduction

The final step this dissertation makes in improving web privacy and security is to explore other ways web-like applications could be developed and deployed, in light of the findings discussed in previous chapters. Chapter 5 presented ways of improving web security and privacy while maintaining compatibility with existing websites. This chapter considers the further improvements that could be achieved with a system designed from the start with the findings from Chapters 3 and 4.

This chapter presents CDF, an alternative method for describing interactive websites. The design requires site authors to describe sites using a declarative, statically checkable format, that trades a loss in author-expressiveness for gains in client-enforceable security guarantees. The design is as a proof of concept, to demonstrate that many kinds of websites users enjoy on the modern web can be implemented with only a subset of the functionality in the browser, and in a manner that allows the client to enforce a greater number of protections and guarantees.

The rest of this chapter is organized as follows: Section 6.2 presents the high level design of the system. Section 6.3 describes an implementation of CDF 's design that leverages the engineering and omnipresence of commodity web browsers. Section 6.4 gives an evaluation of the system's capabilities and security guarantees, a discussion of the limitations of the current design, and the types of applications the system could not implement, in its current state. Section 6.5 discusses this work's place in the overarching goals of this dissertation.

6.2 Design

CDF is an alternative system for creating modern, interactive websites, that provides greater security and privacy guarantees than the current HTML-and-JavaScript system. The principal features in the design of CDF are as follows.

First, CDF prevents websites from running arbitrary JavaScript on the client. Instead, CDF authors create interactive websites by composing trusted, client-controlled implementations of interactive functionality using an easily checked, declarative syntax. Second, CDF only uses a small subset of browser features, allowing websites to access only the "core", or most popular and frequently used parts of the Web API, when creating web sites. Third, CDF places stricter constraints on the kinds of web documents than current HTML -and-JavaScript applications enforce to restrict possible data flows through the application.

Table VIII provides a comparison of the capabilities and guarantees made by current HTML-and-JavaScript based applications, contrasted against CDF documents. The following subsections detail each aspect of CDF's design.

Capability	HTML + JS	CDF
Load static media from remote and local domains	✓	✓
Load non-client controlled JavaScript	✓	-
Can express common web design idioms	✓	✓
Server control over HTTP `referer` and related privacy settings	✓	-
Client guarantees over HTTP `referer` and related privacy settings	-	✓
Read sensitive values from cookies, local storage, etc.	✓	-
Sub-page / AJAX requests and updates	✓	✓
Allow form submissions and AJAX updates to remote domains	✓	-
HTML5 multimedia (`<audio>`, `<video>`)	✓	✓
Supports common browser plugins (Flash, Java, Silverlight)	✓	-
Advanced JavaScript tools (WebGL, `<canvas>`, ASM.js)	✓	-
Client side storage (IndexDB, localStorage, etc)	✓	-
Offline Applications	✓	✓

TABLE VIII: FEATURE COMPARISON BETWEEN HTML AND CDF DOCUMENT FORMATS.

6.2.1 Trusted Feature Implementation

CDF's main method for improving user security and privacy is by preventing websites from executing arbitrary JavaScript on the client. The current JavaScript-based system for providing interactive websites is the cause of many web browser security problems. Browsers must trust that code will carry out some non-malicious purpose when executing it, and that a given set of JavaScript instructions will benefit the user (by, for example, setting up a website's user interface elements), instead of harming the user (e.g. by fingerprinting the user, accessing a browser feature with a known security flaw, or sending a session token to a malicious destination).

Instead of try to verify that JavaScript code is benign before execution (a difficult-to-impossible task), CDF takes the simpler approach of not allowing applications to provide their own JavaScript. CDF instead provides a set of trusted, client-side implemented interactive primitives, which web authors can compose into higher-level functionality with a declarative, easy to verify syntax. CDF authors can, for example, tie a *mouse click* event to a *document attribute change* event, not by writing code directly, but through the structure of the document. CDF clients include their own trusted libraries that handle generating code and executing the relevant functionality on the clients, without trusting code provided by the website.

The result is that CDF documents are composed from functionality implemented in trusted, client-controlled libraries. These libraries are designed to compose safely, and pages can only access them through a simple, declarative syntax. This is in contrast to the typical JavaScript based approach, where websites can execute arbitrary code, and web browsers must judge if the resulting behavior seems safe through heuristics, like XSS filters and code origin reputation systems.

6.2.2 Feature Selection

CDF also protects user security and privacy by reducing the browser's attack surface by preventing websites from accessing browser functionality that is either rarely used, or predominantly used for non-user-serving purposes (e.g. browser fingerprinting).

As demonstrated in Chapters 3 and 4, modern web browsers implement a huge array of Web API features. While some of this functionality is closely related to the web's most-frequent purpose of delivering interactive documents, other functionality is never used, used in only rare niche situations, or used predominantly for malicious purposes.

CDF uses these findings to improve user security and privacy by restricting websites to only frequently-used, document-manipulation related Web API features. By preventing websites from accessing features that are not generally used for user serving interests (either because those features are primary used for advertising and tracking, or because the features are rarely used at all), CDF brings web browsers into closer alignment with the security principal of least privilege. The attack surface exposed to websites is dramatically reduced, with minimal impact to the user experience.

6.2.3 Document Constraints

Current HTML applications include several other design aspects that make them difficult to secure. To name only a few such examples: HTML and JavaScript applications allow scripts to be loaded from remote locations from any part of the HTML document, enabling many XSS attacks. HTML documents can contain complete sub-documents through the use of `<iframe>` elements, enabling drive by downloads and related attacks. HTML applications generally include a “referer” header when requesting remote resources, enabling some forms of user tracking.

CDF improves user security and privacy by tightly-controlling what kinds of resources documents can fetch, and what information is sent during the fetch request. CDF documents cannot include arbitrary code (either inline or hosted remotely), include sub-documents, or send information generated in the client directly to remote domains.

6.3 Implementation

We implemented CDF in two parts, first as a document specification, and second as several additions to the browser’s trusted base: a *parser* that converts CDF documents into trusted

HTML and JavaScript, a *HTTP proxy* that converts CDF documents for use in web browsers, and a set of *trusted JavaScript libraries* that run in the browser to implement the interactive aspects of CDF applications.

The described system was implemented to allow CDF documents to be run in web browsers today, with no additions or modifications needed to any recently released browser. The same design could be implemented by modifying a browser to be able to parse and understand CDF documents "natively", though at the cost of a much greater engineering task.

We also adopted CSS as is, to handle the presentation of CDF applications. We did so to minimize the engineering effort needed to implement the CDF concept, and because of the relative lack of security issues associated with CSS compared to JavaScript. While privacy issues have been raised concerning recent CSS features **??**, such issues are beyond this scope of this work, other than to note that similar sub-setting approaches could be implemented in future-CDF-like systems to address such attacks.

This section gives a high level explanation of one possible implementation of CDF 's design. Documentation for creating CDF documents, including type specifications, nesting rules, and the interactivity primitives included in CDF can be found in an open source implementation and accompanying documentation[1].

[1] https://github.com/bitslab/cdf.

6.3.1 Document Format

CDF uses JavaScript Object Notation (JSON) strings to represent documents. CDF documents are trees of typed objects. Types in CDF fall into one of four categories.

- **Elements.** The structure and text of the document.
- **Events.** New input from the network or the user.
- **Behaviors.** Descriptions of what should happen when an Event has triggered.
- **Deltas.** Changes to be applied to the document.

Each type defines the configuration it can receive (e.g. the URL that a `image` object can refer to), and the types it accepts as children in the tree. For example, `text` objects can be children of `button` objects (to create labels on buttons), but `button` objects cannot be children of `text` objects. Since the types in CDF are all well-defined, they can be strictly checked to ensure they will have predictable effects when rendered in the client.

Some types accept configuration parameters (e.g. the class names to add to the element when rendered in HTML, or the local URL to post a form's information to). These configuration parameters are also strictly typed, and are checked for safety and correctness before being rendered in the client.

Types are designed to emphasize predictable information flow and user privacy. For example, in CDF `form` elements are only allowed to send information to the origin domain, while in HTML applications, `<form>` elements can be configured to send information to any domain.

6.3.2 Trusted Base Additions

We implemented the CDF design through three additions to the current trusted web browser trusted base. These additions, in tandem, enforce the security and privacy properties discussed in Section 6.2.

6.3.2.1 Parser

The first addition CDF makes to the browser's trusted base is a CDF parser. The role of the CDF parser is to take strings and either identify them as invalid CDF documents, or to render an equivalent and safe HTML and JavaScript string that can be rendered in the browser. The parser also provides debugging information as a convenience to CDF authors.

If the parser is given a valid CDF document, it converts it into a combination of HTML tags, escaped text, `<script>` tags referencing JavaScript libraries that are part of the CDF trusted base, and `<script>` tags containing parameters to be passed to those trusted libraries. Invalid documents "fail closed", and return an error code and no output.

6.3.2.2 HTTP Proxy

The second addition CDF makes to the browser's trusted base is a HTTP proxy that sits between the browser and the internet. The proxy passes requests from the browser to the destination server unchanged. Once the server responds, the proxy examines the response. If the response appears to be a CDF document, the HTTP proxy extracts the body of the request and provides it to the parser. If the parser accepts the response as a valid CDF document, the proxy passes the parser-generated HTML and JavaScript back to the client. If the parser rejects the server's response as invalid CDF, the proxy instead passes back an error message to the client, informing the user that the server provided an invalid document.

6.3.2.3 Client JavaScript Libraries

The third addition to the browser's trusted base is in a small number of JavaScript libraries (14) that implement the interactive elements of each page. These libraries handle all the client-side logic and functionality needed for all of the event, behavior and delta types used in the system, plus some plumbing code needed to route the parameters extracted by the parser to the correct library implementations.

6.4 Evaluation

We tested the usability and expressiveness of CDF by implementing several popular types of web applications in the the system. We selected these applications (a blog modeled on `https://www.vogue.com/`, an online-banking site based on `https://www.bankofamerica.com/`, a social media site modeled on `https://twitter.com/`, and a collaborative web application similar to HotCRP (100)) to represent the range of sites that web users commonly interact with. In each case we were able to replicate the user-facing functionality of each page.

This section evaluates the security benefits of CDF's approach for describing interactive websites. For each issue, we briefly describe a vulnerability common in current web applications, and then describe how CDF improves the situation.

6.4.1 Cross-Site Scripting

XSS refers to when attackers are able to inject JavaScript code into an HTML document, so that the code is executed by all site visitors, trusted as if the code came from the site author.

The technique is used for many malicious purposes, including extracting session tokens from the client or redirecting the user to a domain the attacker controls.

CDF protects the client from XSS attacks. First, and most significantly, it removes the ability for a document to describe any kind of JavaScript code directly. Instead of arbitrary code, CDF documents can only describe a composition of trusted, safe types. While a malicious attacker could possibly corrupt a target server to present visitors a different composition of types than the application author intended, CDF's types constrain the functionality that can be described to only safe activities. CDF does not include, for example, any way to access cookie values or redirect the client to another location with JavaScript, common goals of CSS attacks.

6.4.2 Page Alteration / Defacement

When application authors do not adequately sanitize or validate the inputs users provide to their site, they risk giving users the ability to deface, or otherwise unexpectedly alter, the presentation of their website. This can lead to a blurring of the line between a message provided by the page author (which may be trusted by site visitors) and other web site visitors (which may be untrusted). This may happen when a naive application author concatenates the user's input, represented as a string, into a larger string the author is using for the returned content.

CDF's type system makes this kind of error more difficult to make. The CDF author must construct pages as trees of instances of types. The page structure and styling cannot be modified from within an individual child node in the document tree. In cases where page authors are taking inputs from users, and anticipate that input to be in the form of an unstructured piece of text (such as a comment on an article), the page author would do so by setting the user's input string as the content of a `text` element. When CDF then renders the document to send to visitors of the site, the CDF parser escapes all content in `text` instances to ensure that the content cannot change the structure of the page (such as by including JavaScript code or altering the balance of tags on the page).

While CDF does not make this kind of attack impossible (it is possible to conceive of ways that a sufficiently naive page author would construct a vulnerable document), it makes the attack much more difficult to execute. Instead of becoming relatively easy for page authors to be affected by this kind of attack, CDF instead makes it difficult and less likely.

6.4.3 Limited Trusted Base

A further source of vulnerability in HTML documents is that they allow attackers to take advantage of a greatly expanded trusted base, in the form of browser plugins like Java and Flash, and in the form of infrequently used Web API features. As the frequent rate of browser updates shows, securing just the browser is an extremely difficult task. Needing to trust the browser *in addition to* closed source, third party plugins with long histories of exploitability makes the problem of securing the web dramatically more difficult.

CDF further reduces the attack surface by removing the ability of CDF documents to include or refer to plugins. As previously discussed, CDF does not include any way to represent an `<object>`, `<embed>` or `<iframe>` tag on the page, nor does it have a `<script>` type that could be used to include the same client side. Earlier in the web's evolution, popular features like audio and video could only be provided by these third party plugins. Now that the web has matured and all popular browsers support standards for audio and video with hardware-accelerated playback, the absolute necessity of these extensions is limited. The CDF specification makes it impossible for CDF page authors to reference or interact with any plugins that might be on the system.

6.4.4 Client Side Fingerprinting

Web users who have not authenticated or intentionally identified themselves to a website expect to be semi-anonymous. Once a user discards any identifying tokens they've received from a website (e.g. deleting their browser cookies), they have a reasonable expectation that they are no longer known to the site. Such assumptions are even built into the state-less nature of HTTP, and required the addition of cookies to add state into the web.

Malicious websites violate this assumption through client side fingerprinting, or by including JavaScript code in their pages to take a large number of quasi-identifying measurements, and combine them in such a way that site visitors can be uniquely identified. These quasi-identifiers are not sensitive to users deleting their cookies, modifying their user agent string, or taking other similar steps, making it difficult for users to regain their privacy.

While not all of these techniques rely on client executed JavaScript code, many do, such as canvas based fingerprinting (27; 34), identifying the JavaScript engine being used(25) or font and plugin enumeration (24). CDF prevents these client-side fingerprinting techniques by removing the ability of page authors to include code that takes the relevant measurements. For example, there is no way for a CDF author to construct a CDF document that will query the versions of what plugins are installed on the system, or to use the `<canvas>` tag to take semiuniquely-identifying measures of the visitors browser. By removing the ability of document authors to include arbitrary JavaScript in their pages, and by making it impossible to create documents that take the same identifying measures, CDF prevents client-side fingerprinting and increases the amount of anonymity users can expect.

6.4.5 Predictable Information Flow

A final threat to the privacy and security of web users is that it is difficult, if not impossible, for the average user to predict what information they are sharing when they visit a website, and where that information is being sent. A user may visit a website on a domain they trust—and not intending to trust any other domains in doing so—only to later learn that the site (maliciously or unknowingly) notified a third party that they visited the site.

CDF addresses this issue in three ways. First, the most popular and intrusive tracking systems used today rely, at least in part, on JavaScript run on the client. Inclusion of third party tracking libraries is inexpressible in CDF, and thus the user automatically gains a great deal of privacy-preservation.

Second, the CDF parser sets the `Content-Security-Policy` of all documents to `referrer never` though an included `<meta>` tag element, instructing browsers to not send a referrer header when requesting remote resources, further protecting the privacy of the user.

Finally, it is extremely difficult for web users to know where their content will be sent when they interact with a website, whether that interaction is interacting with form element, clicking on a button, or scrolling through a page. Even inspecting the source HTML of the page being viewed is no guarantee, since JavaScript could have manipulated where the form values will be sent. CDF removes these uncertainties by only allowing forms and sub-page requests to send to the current domain.

Tracking pixels which load from third party domains with unique per-user IDs in their URL are still usable in CDF. While this allows some level of tracking to persist in CDF, a third party providing Google Analytics style functionality would need to synchronize the user IDs with every colluding site, rather than rely on JavaScript, cookies, and “referer” headers to reconstruct user browsing history.

6.5 Conclusions

This chapter presents an alternate system to designing and deploying web applications that emphasizes user privacy and security. The system, titled CDF, achieve these gains in two primary ways: first by restricting the set of Web API features websites have access to, and second, but using a statically verifiable, typed document system, instead of the HTML and arbitrary-JavaScript model used on the web today.

This system builds on several findings discussed earlier in this dissertation: that most of the Web API is not used by websites on the internet (Chapter 3), that allowing websites to access this rarely-needed functionality imposes unnecessary risks to users privacy and security (Chapter 4), and that most of the types of functionality users enjoy online can be provided by limiting websites to a subset of. “core”, low risk Web API features (Chapters 4 and 5).

While not intended to be used by web developers and users as is, the CDF system presented in this chapter demonstrates that most of the benefits of modern web applications can be implemented and enjoyed at much lower risk to web users than the current HTML and JavaScript system entails.

CHAPTER 7

CONCLUSION

The modern web is the result of a long series of largely uncoordinated iterative changes, driven by a combination of lineage[1], market competition[2], well-intentioned but incorrect efforts to predict where the web would go[3], the inability or refusal of standards committees to work together for the benefit of system cohesion[4], and efforts to push the browser as a near-replacement for the operating system[5], among many other well-intentioned reasons.

Given this chaotic, unplanned, and organic development history, its worth celebrating how successful and useful the web has been. The web is almost certainly the worlds largest, open application platform, and despite the many of security and privacy issues the web has suffered over its history, its worth noting that, in many ways, the web platform has proved to be more secure than its competitors.

---

[1]e.g. the adoption of Hypercard's event model in the early DOM standards.

[2]e.g. the approximately 10-day design and implementation cycle allowed for the development of JavaScript

[3]e.g. the entire SVG Web API standard, intended to compete with Flash's vector graphics system, but which received little traction

[4]e.g. the choice of the CSS committee to use kebab-case for style properties, v.s. the DOM committee's choice to use camel-case names.

[5]e.g. the large number of Web API standards for interacting with hardware, largely adopted because of the "Firefox OS" and "Chrome OS" projects

However, to say the web has been a huge success (for application users and developers alike) is not to say the system cannot be improved. As this dissertation has hopefully demonstrated, web users face a large number of unnecessary privacy and security risks.

This dissertation aimed to improve the state of web privacy and security by applying a cost-benefit analysis to one important part of the browser, the Web API, and seeing how those findings can be used to make the platform safer for users. Chapter 3 presented a technique for measuring what parts of the Web API are actually being used on the web, along with the results of applying that technique to the Alexa 10k. Chapter 4 built on these measurements to systematically measure the costs and benefits posed by each of the standards in the Web API, to distinguish highly beneficial functionality from functionality that placed web users at unnecessary, uncompensated risk.

Chapter 5 considered how these cost-benefit measurements could be used to improve privacy and security on the web as it exists today, and demonstrated the possibility for improvement through the development of a publicly available browser extension that enforces access-controls on the Web API.

Finally, Chapter 6 considered how these cost-benefit measurements could be used in the development of alternate networked application system, in order to further protect web users, and demonstrated the feasibility of this approach through CDF, a system for developing and deploying safer web applications, using commodity web browsers.

The author hopes that, collectively, the findings and data presented in this dissertation can play a small part in guiding standards committees, future researchers, and browser vendors in the development of a safer, more privacy preserving web.

CITED LITERATURE

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

2. Jackson, D.: Webgl specification. `https://www.khronos.org/registry/webgl/specs/1.0/`, 2014.

3. Jackson, D. and Gillbert, J.: Webgl 2.0 specification. `https://ww.khronos.org/registry/webgl/specs/latest/2.0/`, 2018.

4. International, E.: Ecmascript 2017 language specification. `https://www.ecma-international.org/publications/standards/Ecma-262.htm`, 2017.

5. eyeo GmbH: Adblock plus. `https://adblockplus.org/`, 2018.

6. Maone, G.: Noscript - javascript/java/flash blocker for a safer firefox experience! `https://noscript.net/`, 2015.

7. Patrizio, A.: 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.

8. Blue, V.: You say advertising, i say block that malware. `http://www.engadget.com/2016/01/08/you-say-advertising-i-say-block-that-malware/`, 2016.

9. Bryant, M.: 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.

10. Heiderich, M., Frosch, T., and Holz, T.: Iceshield: detection and mitigation of malicious websites with a frozen dom. In International Workshop on Recent Advances in Intrusion Detection, pages 281–300. Springer, 2011.

11. Dingledine, R., Mathewson, N., and Syverson, P.: Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

12. Web workers. `https://www.w3.org/TR/workers/`, 2016.

13. Perry, M., Clark, E., and Murdoch, S.: The design and implementation of the tor browser. `https://www.torproject.org/projects/torbrowser/design/#fingerprinting-linkability`, 2015.

14. Stamm, S., Sterne, B., and Markham, G.: Reining in the web with content security policy. In Proceedings of the 19th International Conference on World Wide Web, pages 921–930. ACM, 2010.

15. Meyerovich, L. A. and Livshits, B.: Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In 2010 IEEE Symposium on Security and Privacy, pages 481–496. IEEE, 2010.

16. Miller, M. S.: Google caja. `https://developers.google.com/caja/`, 2013.

17. Guarnieri, S. and Livshits, B.: Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In Proceedings of the 18th conference on USENIX security symposium, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

18. Android developer's guide: System permissions. `https://developer.android.com/guide/topics/security/permissions.html`, 2015.

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

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

21. Rader, E.: 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.

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

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

24. Eckersley, P.: How unique is your web browser? In Privacy Enhancing Technologies, pages 1–18. Springer, 2010.

25. Mowery, K., Bogenreif, D., Yilek, S., and Shacham, H.: Fingerprinting information in javascript implementations. Proceedings of W2SP, 2011.

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

27. Mowery, K. and Shacham, H.: Pixel perfect: Fingerprinting canvas in html5. Proceedings of W2SP, 2012.

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

29. Jang, D., Jhala, R., Lerner, S., and Shacham, H.: 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.

30. Van Goethem, T., Joosen, W., and Nikiforakis, N.: 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.

31. Kamkar, S.: Evercookie - virtually irrevocable persistent cookies. `http://samy.pl/evercookie/`, , 2015.

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

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

34. Acar, G., Eubank, C., Englehardt, S., Juarez, M., Narayanan, A., and Diaz, C.: 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.

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

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

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

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

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

40. Vasilyev, V.: fingerprintjs2. `https://github.com/Valve`, 2015.

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

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

43. Web Hypertext Application Technology Working Group (WHATWG): Html living standard. `https://html.spec.whatwg.org/`, 2018.

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

45. Hors, A. L., Hegaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., and Byrne, S.: Document object model (dom) level 2 core specification. https://www.w3.org/TR/DOM-Level-2-Core/, 2000.

46. Hors, A. L., Hegaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., and Byrne, S.: Document object model (dom) level 3 core specification. https://www.w3.org/TR/DOM-Level-3-Core/, 2004.

47. Mozilla Developer Network: Object.prototype.watch() - javascript — mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/watch.

48. Zaninotto, F.: Gremlins.js. https://github.com/marmelab/gremlins.js, 2016.

49. Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M.: 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.

50. Liu, B., Nath, S., Govindan, R., and Liu, J.: 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.

51. van Kesteren, A.: Xmlhttprequest. https://xhr.spec.whatwg.org/, 2016.

52. Hickson, I., Pieters, S., van Kesteren, A., Jgenstedt, P., and Denicola, D.: Html: Plugins. https://html.spec.whatwg.org/multipage/webappapis.html#plugins-2, 2016.

53. van Kesteren, A. and Hunt, L.: Selectors api level 1. https://www.w3.org/TR/selectors-api/, 2013.

54. Kostiainen, A.: Vibration. http://www.w3.org/TR/vibration/, 2015.

55. Pieters, S. and Glazman, D.: Css object model (css-om). https://www.w3.org/TR/cssom-1/, 2016.

56. Hickson, I., Pieters, S., van Kesteren, A., Jgenstedt, P., and Denicola, D.: Html: Channel messaging. https://html.spec.whatwg.org/multipage/comms.html#channel-messaging, 2016.

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

58. van Kesteren, A.: Encoding standard. `https://encoding.spec.whatwg.org/`, 2016.

59. Bergkvist, A., Burnett, D. C., Jennings, C., Narayanan, A., and Aboba, B.: Webrtc 1.0: Real-time communication between browser. `https://www.w3.org/TR/webrtc/`, 2016.

60. Hors, A. L., Hegaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., and Byrne, S.: Web cryptography api. `https://www.w3.org/TR/WebCryptoAPI/`, 2014.

61. Grigorik, I., Mann, J., and Wang, Z.: Performance timeline level 2. `https://w3c.github.io/performance-timeline/`, 2016.

62. Grigorik, I., Mann, J., and Wang, Z.: Ui events. `https://w3c.github.io/uievents/`, 2016.

63. Shin, Y., Meneely, A., Williams, L., and Osborne, J. A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Transactions on Software Engineering, 37(6):772–787, 2011.

64. Liu, C., White, R. W., and Dumais, S.: 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, pages 379–386. ACM, 2010.

65. Ozment, A. and Schechter, S. E.: Milk or wine: does software security improve with age? In Usenix Security, 2006.

66. Zimmermann, T., Nagappan, N., and Zeller, A.: Predicting bugs from history. In Software Evolution, pages 69–88. Springer, 2008.

67. Google: boringssl - git at google. `https://boringssl.googlesource.com/boringssl/`, 2018.

68. Cve-2012-4171. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4171`, 2012.

69. Cve-2013-2031. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2031, 2013.

70. Cve-2011-2363. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-2363, 2011.

71. Cve-2015-0818. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0818, 2015.

72. Corporation, M.: Dxr. https://github.com/mozilla/dxr, 2016.

73. Laperdrix, P., Rudametkin, W., and Baudry, B.: 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.

74. Alaca, F. and van Oorschot, P.: Device fingerprinting for augmenting web authentication: Classification and analysis of methods. In Proceedings of the 32th Annual Computer Security Applications Conference, 2016.

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

76. Cao, Y., Li, S., and Wijmans, E.: (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In Proceedings of the Symposium on Networked and Distributed System Security, 2017.

77. Gras, B., Razavi, K., Bosman, E., Bos, H., and Giuffrida, C.: ASLR on the Line: Practical Cache Attacks on the MMU. In Proceedings of the Symposium on Networked and Distributed System Security, 2017.

78. Englehardt, S. and Narayanan, A.: Online tracking: A 1-million-site measurement and analysis. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1388–1401. ACM, 2016.

79. Kotcher, R., Pei, Y., Jumde, P., and Jackson, C.: Cross-origin pixel stealing: timing attacks using css filters. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 1055–1062. ACM, 2013.

80. Acar, G., Juarez, M., Nikiforakis, N., Diaz, C., Gürses, S., Piessens, F., and Preneel, B.: Fpdetective: dusting the web for fingerprinters. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 1129–1140. ACM, 2013.

81. Van Goethem, T., Vanhoef, M., Piessens, F., and Joosen, W.: Request and conquer: Exposing cross-origin resource size. In Proceedings of the Usenix Security Symposium, 2016.

82. Gelernter, N. and Herzberg, A.: Cross-site search attacks. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 1394–1405. ACM, 2015.

83. Tian, Y., Liu, Y. C., Bhosale, A., Huang, L. S., Tague, P., and Jackson, C.: All your screens are belong to us: attacks exploiting the html5 screen sharing api. In 2014 IEEE Symposium on Security and Privacy, pages 34–48. IEEE, 2014.

84. Xu, M., Jang, Y., Xing, X., Kim, T., and Lee, W.: Ucognito: Private browsing without tears. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 438–449. ACM, 2015.

85. Kim, H., Lee, S., and Kim, J.: Exploring and mitigating privacy threats of html5 geolocation api. In Proceedings of the 30th Annual Computer Security Applications Conference, pages 306–315. ACM, 2014.

86. Andrysco, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., and Shacham, H.: On subnormal floating point and abnormal timing. In 2015 IEEE Symposium on Security and Privacy, pages 623–639. IEEE, 2015.

87. Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D.: 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, pages 1406–1418. ACM, 2015.

88. Gruss, D., Bidner, D., and Mangard, S.: Practical memory deduplication attacks in sandboxed javascript. In European Symposium on Research in Computer Security, pages 108–122. Springer, 2015.

89. Weissbacher, M., Robertson, W., Kirda, E., Kruegel, C., and Vigna, G.: Zigzag: Automatically hardening web applications against client-side validation vulnerabilities. In 24th USENIX Security Symposium (USENIX Security 15), pages 737–752, 2015.

90. Son, S. and Shmatikov, V.: The postman always rings twice: Attacking and defending postmessage in html5 websites. In NDSS, 2013.

91. Olejnik, L., Acar, G., Castelluccia, C., and Diaz, C.: The leaking battery a privacy analysis of the html5 battery status api. Technical report, Cryptology ePrint Archive, Report 2015/616, 2015, ht tp://eprint. iacr. org, 2015.

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

93. Olejnik, L.: Stealing sensitive browser data with the W3C Ambient Light Sensor API. `https://blog.lukaszolejnik.com/stealing-sensitive-browser-data-with-the-w3c-ambient-light-sensor-api/`, 2017.

94. Grigorik, L., Simonsen, J., and Mann, J.: High resolution time level 2. `https://www.w3.org/TR/hr-time-2/`, 2016.

95. Pietraszak, M.: Browser extensions. `https://browserext.github.io/browserext/`, 2018.

96. International, E.: Ecmascript language specification - ecma-262 edition 5.1. `https://www.ecma-international.org/ecma-262/5.1/`, 2011.

97. Vukicevic, V., Jones, B., Gilbert, K., and Wiemeersch, C. V.: Webvr. `https://immersive-web.github.io/webvr/spec/1.1/`, 2017.

98. Grant, R. and Rockot, K.: Webusb api. `https://wicg.github.io/webusb/`, 2018.

99. Shires, G. and Wennborg, H.: Web speech api specification. `https://w3c.github.io/speech-api/webspeechapi.html`, 2014.

100. Kohler, E.: Hotcrp conference management software. `http://www.read.seas.harvard.edu/~kohler/hotcrp/`, 2014.

VITA

NAME	Peter Edwin Snyder
EDUCATION	B.A., Political Science, Lawrence University, Appleton, WI, 2006
TEACHING	Department of Computer Science, University of Illinois at Chicago, Software Design (CS342, Summer 2017)
PUBLICATIONS	Peter Snyder, Cynthia Taylor, and Chris Kanich. "Most Websites Dont Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security." In Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS, 2017).
	Peter Snyder, Periwinkle Doerfler, Chris Kanich, and Damon McCoy. "Fifteen Minutes of Unwanted Fame: Detecting and Characterizing Doxing." In Proceedings of the 2017 Internet Measurement Conference (IMC, 2017).
	Peter Snyder, Laura Watiker, Cynthia Taylor, and Chris Kanich. "CDF: Predictably Secure Web Documents." In Proceedings of the 2017 IEEE Workshop on Technology and Consumer Protection (ConPro, 2017).
	Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. "Browser Feature Usage on The Modern Web." In Proceedings of the 2016 ACM Internet Measurement Conference (IMC, 2016).
	Peter Snyder and Chris Kanich. "Characterizing Fraud and Its Ramifications in Affiliate Marketing Networks." Journal of Cybersecurity (2016).
	Peter Snyder, Chris Kanich, and Michael K Reiter. "The Effect of Repeated Login Prompts on Phishing Susceptibility." In Proceedings of the Workshop on Learning from Authoritative Security Experiment Results (LASER, 2016).
	Peter Snyder and Chris Kanich. "No Please, After You: Detecting Fraud in Affiliate Marketing Networks." In Workshop on the Economics of Information Security (WEIS, 2015).

Jason W Clark, Peter Snyder, Damon McCoy, and Chris Kanich. “I Saw Images I Didn’t Even Know I Had: Understanding User Perceptions of Cloud Storage Privacy.” In Proceedings of the 33rd ACM Conference on Human Factors in Computing Systems (CHI, 2015).

Peter Snyder. “Yao’s Garbled Circuits: Recent Directions and Implementations.” In Written Critique and Presentation Exam (qualifier) Report (2014).

Peter Snyder and Chris Kanich. “Cloudsweeper and Data-centric Security.” ACM SIGCAS Computers and Society (2014).

Peter Snyder and Chris Kanich. “CloudSweeper: Enabling Data-Centric Document Management for Secure Cloud Archives.” In Proceedings of the 2013 ACM Workshop on Cloud Computing Security (CCSW, 2013).