Measurement and Detection of Security Properties of Client-Side Web Applications

A Thesis Presented

by

Michael Weissbacher

to

The College of Computer and Information Science

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

Information Assurance

Northeastern University

Boston, Massachusetts

April 2018

For my parents.

Contents

List of Figures **v**

List of Tables **vii**

Acknowledgments **viii**

Abstract of the Thesis **ix**

1 Introduction **1**
1.1 Structure of the Thesis 4

2 Background **5**
2.1 HTTP Security Headers 5
2.1.1 Overview of Security Headers 5
2.1.2 Browser policy frameworks 6
2.1.3 Content Security Policy 7
2.1.4 Evasion and Attacks Against CSP 8
2.1.5 Beyond Level 1 of CSP 8
2.2 Browser Extensions 8
2.2.1 Extension Security Aspects for Browsers 8
2.2.2 Privacy leaks in Extensions 11
2.2.3 Privacy leaks in Other Platforms 11
2.2.4 Extension Ad Injection 12
2.2.5 Extension Analysis Systems 12
2.2.6 Tracking: Extensions and the Web 13
2.3 Vulnerabilities in Web Applications 14
2.3.1 Measuring and Reducing Complexity 14
2.3.2 Single Page Applications 15
2.3.3 Mash-ups or Widgets 15
2.3.4 Client-side communication 16
2.3.5 Popular JavaScript Frameworks 16
2.3.6 Server-Side Pre-Rendering 18
2.3.7 Vulnerability Scanners 19
2.3.8 Analysis of Web Vulnerability Scanners 20

3 Investigating Content Security Policy **22**
3.1 Introduction . . . 22
3.2 Content Security Policy . . . 24
3.2.1 Usage of HTTP Security Headers . . . 26
3.3 CSP Violation Reports . . . 31
3.3.1 Background . . . 31
3.3.2 Methodology . . . 32
3.3.3 Results . . . 34
3.3.4 Conclusions . . . 38
3.4 Semi-Automated Policy Generation . . . 39
3.4.1 Methodology . . . 39
3.4.2 Evaluation . . . 40
3.4.3 Conclusions . . . 44
3.5 Discussion . . . 45
3.5.1 Discussions with Security Engineers . . . 45
3.5.2 Suggested Improvements . . . 46
3.6 Chapter Summary . . . 46
3.7 Future Work . . . 47

4 Identifying History Leaking Browser Extensions **48**
4.1 Introduction . . . 48
4.2 Motivation . . . 50
4.2.1 HTTP URL Honeypot . . . 51
4.2.2 Types of Trackers . . . 52
4.2.3 Threat Model . . . 52
4.3 Case study of a large history data collector . . . 53
4.3.1 Origins of Data . . . 53
4.3.2 SimilarWeb Chrome Extension . . . 54
4.3.3 Finding More Extensions . . . 54
4.3.4 Network Information . . . 55
4.3.5 Reported Extensions . . . 56
4.4 Information Leaks in High-Profile Extensions . . . 56
4.4.1 WOT: Web of Trust, Website Reputation Ratings . . . 57
4.4.2 CouponMate: Coupon Codes & Deals . . . 57
4.5 Detection Approach . . . 57
4.5.1 Overview . . . 58
4.5.2 Network Counterfactual Analysis . . . 60
4.5.3 Extension Triage . . . 62
4.5.4 History Leakage Detection . . . 64
4.6 Ex-Ray Implementation . . . 65
4.6.1 Extension Containers . . . 65
4.6.2 Browser Instrumentation . . . 65
4.7 Evaluation . . . 67
4.7.1 Experimental Setting . . . 67
4.7.2 Ex-Ray Results . . . 68

4.8 Discussion . . . 74
4.8.1 Browser-enabled Tracking . . . 74
4.8.2 Foundations Towards Solutions . . . 75
4.8.3 Evasion . . . 75
4.9 Future Work . . . 76
4.10 Chapter Summary . . . 77

5 SPA Rewriting to Enhance Vulnerability Discovery 78
5.1 Introduction . . . 78
5.2 Motivation . . . 80
5.2.1 Threat Model . . . 80
5.3 System Overview . . . 81
5.3.1 JavaScript Instrumentation . . . 81
5.3.2 Vulnerability Scanner Capability Test and Log Analysis . . . 82
5.4 SPARE–SPA Rewriting for Exploitability . . . 84
5.4.1 State Management and Manipulation . . . 87
5.4.2 Accessibility of views . . . 87
5.5 Evaluation . . . 88
5.5.1 Setup . . . 89
5.5.2 Evaluation of Existing Black-box Systems . . . 90
5.5.3 Crawling Results . . . 90
5.5.4 Measuring and Comparing Capabilities . . . 94
5.5.5 Security Effectiveness . . . 95
5.5.6 SPARE Results . . . 96
5.6 Discussion . . . 96
5.7 Future Work . . . 98
5.8 Chapter Summary . . . 98

6 Papers 100
6.1 Thesis Publications . . . 100
6.2 Other Work . . . 101

7 Conclusion 104
7.1 Future Work . . . 106

Bibliography 108

List of Figures

1.1 Thesis overview. The areas of work are focusing on three topics of the web security landscape. Analysis of security HTTP headers for hardening web applications, detection of privacy leaks in browser extensions, and client-side web application penetration testing tools. 2

2.1 Contributors to JavaScript frameworks on GitHub [41]. 17
2.2 Ecosystem of frameworks, demonstrating overall momentum and usage [41] . 18

3.1 Popularity of security headers in the Alexa Top 10K. 29
3.2 Fraction of new policy entries discovered over time on site B (measurement inactive during the dashed intervals). It can take some time until all legitimate resources have been accessed at least once; in the meantime, many injected resources are reported. 36
3.3 Frequency of legitimate and invalid violations being reported on site D. Some injected resources occurred orders of magnitude more often than legitimate resources. 37

4.1 Extension execution with unique URLs vs. incoming connections to those URLs from the public Internet. These connections confirm that leaked browsing history is used by the receivers, often immediately upon execution. . . . 51
4.2 Neighboring relationships of IPs between seemingly unrelated domains used for monitoring. 55
4.3 Graph linking domain names by IP relationships used in 42 extensions to covertly collect browsing history. 55
4.4 Domains using `upalytics.com` library reported to a network of domains that can be linked by IP neighborship. 55
4.5 Ex-Ray architectural overview. A classification system combines unsupervised and supervised methods. After triaging unsupervised results, a vetted dataset is used to classify extensions based on n-grams of API calls. 58
4.6 Tracking extension. 60
4.7 Benign extensions. 60

4.8 Comparison in change of traffic between executions leaking history and benign extensions. Each bar displays the change of traffic sent relative to executions with increased history. Sent data projects an ascending slope based on size of history. Received data did not reflect this trend. 60
4.9 Extensions interact with multiple servers on the Internet, sending and receiving data. Trackers that receive browsing history behave differently than other servers. By varying browsing history over repeated executions, patterns of trackers become apparent. 66
4.10 Ex-Ray extension execution overview. 66

5.1 Function before performing instrumentation. 82
5.2 Function after instrumentation 82
5.3 Data-collection function which is invoked on each function entry. The call sends all relevant tracing information to a trace collection server. The exit-function operates similarly, except that it is invoked with the return value, which is passed back. 83
5.4 SPAs, consisting of HTML and JavaScript code are transferred to the client. Users interact with the SPA locally, state changes are not visible to the server. Except, if data is explicitly sent to the server. 85
5.5 After rewriting a SPA, the DOM state is stored on the server-side, encapsulating client-side features. Users (or penetration testing tools) can interact with it in the request/response paradigm. The DOM is rendered server-side and the result returned as a static document. The state becomes visible to the server. 85
5.6 Overview of data transferred and actions taken with each state change for a rewritten SPA. 88
5.7 Route and action-specific code injection into JSDOM object. The code is added to the DOM as a new `script` element, executed immediately, and removed from the DOM before being rendered for a response. 88

List of Tables

3.1 The types of directives supported in the current W3C standard CSP 1.0. . . 24
3.2 Number of websites with security-related HTTP response headers, grouped by intervals of site popularity, for the Alexa Top 1M ranking. 28
3.3 Overview of enforced policies. 30
3.4 Overview of the CSP violation report data sets received from partner websites in early 2014, after removing inconsistent reports. 33
3.5 Length of policies when whitelisting all violations from the report data set (a), and with an additional filter for URL schemes of browser extensions (b). Most of the policy entries correspond to injected resources; only few are intended to be included. (In brackets, the number of unique policy entries when disregarding the protocol HTTP(S) or alternative domains, such as the www subdomain.) 34
3.6 Most frequent Chrome extensions observed at site D. 35
3.7 Overlap between the sets of policy entries generated by the crawler, through manual browsing and from user-submitted reports. (In brackets, the number of common/different policy entries when disregarding alternative domain names or HTTP(S).) No method was fully reliable. 41
3.8 Additional policy entries discovered in repeated crawls. The high variability due to advertising on the BBC precludes CSP from being used effectively. CNN's way of including advertisement results in a relatively stable (and enforceable) policy. 44

4.1 Instrumented files, functions, and collected parameters after running the LibTooling program on the Chromium source code. 67
4.2 Top five extensions connecting to the honeypot with highest installation numbers which are still available in the Chrome Web Store. 69
4.3 n-gram classification results for varying n. 73

5.1 Characteristics of the scanners evaluated 89
5.2 Pages reached without any modifications to the testing application 91
5.3 Crawling challenges successfully completed 94
5.4 Redundant reports and redundancy rate of Google Cloud Security Scanner . 95

Acknowledgments

This work could not have been realized without my advisors Engin Kirda and William Robertson who offered me the opportunity to pursue this very thesis. I am thankful for having been able to interact with lab-colleagues past and present: Collin Mulliner, Patrick Carter, Matthew Clarke-Lauer, Andrea Mambretti, Tobias Lauinger, Amin Kharraz. They offered feedback and discussion on my projects as we gradually became friends. Especially to Collin, who showed me the ropes of doing research - thank you!

Friends and colleagues at Northeastern and elsewhere, who provided different viewpoints and generally made this time more enjoyable: Arash Molavi, Ancsa Hannak, Konstantinos Athanasiou, Piotr Sapieżyński, Chaima Jemmali, Lucianna Kiffer, Erik-Oliver Blass, Matthias Neugschwandtner, Georg Merzdovnik, Lydia Zakynthinou, Clemens Kolbitsch, Gerald Wodni, Ammar Ammar, Paul Coote, and Lukas Fischer.

Also, to the co-authors of my papers that were eager to walk the extra mile: William Blair, Enrico Mariconti, Yan Shoshitaishvili, Gianluca Stringhini, Guillermo Suarez-Tangil, and Fish Wang. To Christopher Kruegel and Giovanni Vigna for hosting me to work on my Master's thesis, the Shellphish hacking collective, and the UCSB SecLab in it's entirety. I learned many things I still benefit from. *Hack the planet!*

None of this would have been possible without my close friends and family who have been supporting me throughout this pursuit.

Abstract of the Thesis

Measurement and Detection of Security Properties of Client-Side Web Applications

by

Michael Weissbacher

Doctor of Philosophy in Information Assurance

Northeastern University, April 2018

Dr. Engin Kirda, Advisor

Modern Web applications are increasingly moving program logic to the client-side. With the growing adoption of HTML5 APIs, vulnerabilities and hidden functionality are becoming increasingly important to address. However, while detecting and preventing attacks against Web applications is a well-studied topic on the server, considerably less work has been performed for the client.

An early example of a client-side exploit was the 2005 MySpace worm, it spread a million times within 20 hours. While Web applications nowadays are generally more secure than MySpace at the time, software vulnerabilities are still common. Ideally, software would be free of vulnerabilities, however, this currently seems an elusive goal. Nowadays, behavior similar to the MySpace worm would have a more far-reaching impact as client-side Web applications are omnipresent.

By measuring and building detection tools for vulnerabilities and hidden software, exploitation can be prevented. For example, Content Security Policy (CSP) is a technology designed to prevent exploitation of client-side vulnerabilities. However, after measuring and building detection tools, these defenses are often ineffective.

For my thesis I propose novel research into measurement and detection of client-side Web applications security properties. In particular I will address three fields of interest, CSP, history leaks in browser extensions, and black-box Web vulnerability scanners.

In the first part of my thesis, I show that CSP, a promising technology for hardening of Web applications, is used on a low number of websites and rarely used to it's full potential. I perform a long-term measurement, and further determine challenges in deployments that prevent wide adoption. Next, I outline feasibility of semi-automated policy generation, both

from the perspective of a website operator, or an external third party. Finally, I explore barriers to suggest improvements that could help ease CSP adoption.

In the second part, I investigate methods of detection for history-leaking browser extensions. I show that established security measures by browsers are insufficient to prevent such attacks, as extensions can leak history even with modest permissions. I introduce a novel method of detecting such leaks, with a prototype implementation for Chrome extensions, Ex-Ray. Using my method for pre-screening of extension before store admission, browsers can prevent such extensions from being used.

For the third part, I develop methods of assessing black-box Web penetration testing tools under the aspect of Single Page Applications. Unlike server-side penetration testing tools, this area has not yet been researched enough. I use shortcomings which were identified through the assessment to guide enhancements, and present a prototype to overcome some of them.

Chapter 1

Introduction

Both business and personal software used to be delivered as stand-alone packages. Applications such as mail clients or spreadsheet processing were separate programs on client computers. While the early Web was not offering much more than markup for text, the Web nowadays is capable of largely replacing such single-purpose packages. Modern technologies such as HTML 5 have transformed the browser into a platform that handles much of the logic previously performed by stand-alone programs. Online-banking, video conferences, or text editing are often handled through Web applications. The shift of applications from the desktop to the Web has made the browser the de-facto operating system. However, this transfer of program logic also means more software vulnerabilities in Web applications.

While in 1997 only 20% of the top 500 sites used JavaScript, this number increased to 98% in 2016. Furthermore, these pages had a steadily growing number of JavaScript inclusions, statements per file, and cyclomatic complexity [89]. Other sources of complexity are the code base of browsers and how exposed they are to the Web.

The more complex software becomes, the more likely bugs are to riddle it [61, 80]. Data of value attracts attackers who are eager to exploit vulnerabilities in systems. Unfortunately, the shift of complexity from the server to the client also means more programming mistakes are introduced to the client-side. Such mistakes can be exploited by attackers to act on behalf of users, or steal personal information.

Through a higher level of abstraction, systems also become more homogeneous. As a result, attacks against client-side applications can impact large numbers of users, reaching

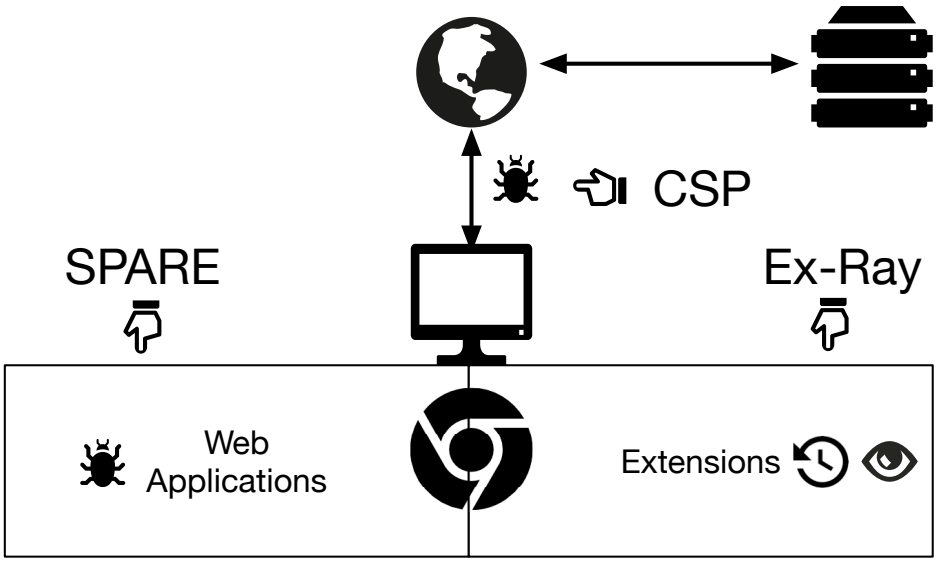

Figure 1.1: Thesis overview. The areas of work are focusing on three topics of the web security landscape. Analysis of security HTTP headers for hardening web applications, detection of privacy leaks in browser extensions, and client-side web application penetration testing tools.

beyond boundaries of underlying hardware or operating system.

In this thesis, I seek to address real-world problems with practical systems in the field of Web security. As software vulnerabilities and hidden functionality in programs are widespread, my primary interest is developing novel methods to measure and detect such behavior.

Thesis Statement. Exploitable software vulnerabilities and hidden functionality permeate software. Analysis of client-side software in search of vulnerabilities and unintended behavior becomes increasingly important as the Web program logic shifts from the server to the client. In this thesis I develop novel methods and automated mechanisms to reduce the impact of client-side vulnerabilities and hidden privacy invasions. I show that such an approach is both feasible and effective. I investigate shortcomings and possible avenues for enhancement of CSP, Web vulnerability scanners, and privacy of browser extensions.

For the purpose of my thesis, I show that it is possible to develop novel methods and automated mechanisms for improved client-side security. I back the claim through contributions in the following three areas:

Content Security Policy (CSP): CSP is a principled and robust browser security mechanism against content injection attacks. A policy is delivered to the browser on a secure channel, and enforced on how the site has to behave. This technique can render exploitation of vulnerabilities impossible, and make even a buggy program resilient to attacks. For this thesis I performed a long-term measurement, finding that usage is low. I further determine challenges in deployments that prevent wide adoption, as it is lagging comparable security headers. Further I will outline feasibility of deploying CSP from the perspective of a website operator, and develop a method of semi-automated policy generation through crawling. Finally, I will explore barriers and suggest improvements that could help ease its adoption.

Browser extension privacy leaks: Browser extensions are a useful mechanism for allowing third-party additions to core browser functionality. However, they pose a security risk in this regard since they have access to privileged browser APIs that are not necessarily restricted by SOP or CSP. This level of access is prone to misuse, as even modest permissions are sufficient to collect and exfiltrate browsing history. Leaking extensions often hide their behavior from users who are unaware of the privacy leak. I developed a methodology for detecting privacy leaks in browser extensions. This approach allows extensions full API access, but can detect history-leaking extensions before they are used, making the browser resilient to such attacks. I further developed a prototype called Ex-Ray, which found extensions in the official Chrome Web store that were immune to previous state-of-the-art privacy-leak systems.

Client-Side Web Penetration Testing Tools: Web Penetration Testing tools for Web services are widely available. However, they focus mostly on server-side attacks such as SQL injection or stored XSS. Client-side attacks such as DOM-based XSS or injection to localStorage are often not in scope, as JavaScript capabilities of these tools are lacking. I first evaluate the state-of-the-art of penetration testing tools on Single Page Applications (SPA), identifying shortcomings and avenues to address them. Second, to overcome some of the identified shortcomings, I develop a prototype, SPARE, to rewrite Angular SPAs to become more accessible to such scanners. Applications are

rendered in a server-side DOM and potentially inaccessible views are exposed to the client. The response is a static HTML file.

The proposed research identifies shortcomings in the state of the art in client-side security and advances it. It furthermore develops new methods of finding misuse of private data. The results can be applied to further secure the modern client-side of the Web.

1.1 Structure of the Thesis

The remainder of the thesis is structured as follows: First, Chapter 2 provides an overview on background information and related work relevant to further chapters. Chapter 3 covers the HTTP security header Content Security Policy, by performing a long-term study, finding shortcomings and suggesting enhancements. In Chapter 4 I discuss history leaks in browser extensions, measure their impact and introduce a detection tool. Chapter 5 performs a measurement of the state of the art of black-box Web vulnerability scanners, identifies shortcomings and addresses some of them. The thesis concludes in Chapter 7, summarizing the findings, outlining future avenues for research and offering general discussion.

Chapter 2

Background

This chapter describes prior work related to the three main areas covered in this thesis, Content Security Policy in Chapter 3, Browser Extensions in Chapter 4, and vulnerability testing of Web applications in Chapter 5.

2.1 HTTP Security Headers

2.1.1 Overview of Security Headers

To discuss CSP in context, I provide a brief overview of the other security relevant HTTP response headers. Details about them can be found at IETF, W3C, or in the browser specifications. CSP itself will be discussed afterwards in more detail.

Platform for Privacy Preferences (P3P) [2]. Websites can use this header to describe their privacy policy. However, it is not supported by major browsers and has not been actively developed for several years. The header is still in use as Internet Explorer blocks third-party cookies by default if no policy is present. P3P is legally binding and has been used in litigation in the past.

DNS Prefetch Control [1]. DNS prefetching is a technique for browsers to reduce latency by resolving referenced hostnames before a user follows a link. This header allows websites to override the default behavior of the browser.

XSS Protection [3]. This header can be used to enable or disable client-side heuristic XSS filtering. The `reflected-xss` directive of CSP 1.1 is functionally equivalent.

Content Type Options [4]. As the `Content-Type` header is often not set correctly, MIME type sniffing can be used to detect the actual response content type. The `nosniff` directive is the only option available for this header and disables MIME type sniffing, preventing possible type confusion.

Frame Options [10]. This header allows a website to restrict iframing to prevent UI redressing attacks. CSP draft 1.1 includes these features under the `frame-ancestors` directive, and may replace this header.

HTTP Strict Transport Security (HSTS) [5]. By using HSTS, websites can specify that in the future, the browser should only connect to them via a secure connection, thereby preventing SSL stripping.

Cross-Origin Resource Sharing (CORS) [8]. SOP has proven to be an obstacle for modern Web applications, and has been worked around by various methods such as JSONP. CORS allows websites to operate outside the limitations of SOP by extending it, while not completely side-stepping it.

2.1.2 Browser policy frameworks

CSP was the first widely deployed browser policy framework to mitigate content injection attacks. However, it was not the first one to be suggested. SOMA (Same Origin Mutual Approval) [67] reduces the impact of XSS and CSRF by controlling information flows. Website operators need to approve content sources in a manifest file, as well as content providers need to approve websites to include their content. BEEP [48] can prevent XSS attacks with a whitelist approach for JavaScript and a DOM sandbox for possibly malicious user content. Script tags are whitelisted by hash, a feature that is also proposed in the 1.1 draft of CSP. BLUEPRINT [95] enforces restrictions on the document parse tree in the browser. Web application server components make parsing decisions and transport the DOM structure to

the client. By enforcing a consistent document structure, misuse of browser rendering quirks is eliminated. CONSCRIPT [62] supports a variety of policies for JavaScript enforcement, which can be generated automatically. Static policy generation is supported for Script#, a Microsoft tool that generates JavaScript from C# code, as well as a dynamic training mode for other platforms. Weinberger et al. [104] performed an evaluation of browser-side policy enforcement systems. They concluded that security policies for HTML should be a central mechanism for preventing content injection attacks, but need more research to become effective. I performed the first study on CSP adoption in the wild, analyzing how usage has evolved in the past year on the most popular websites. Also, I investigate how report-only mode can be used to devise policies, and whether those are effective.

Currently, inline scripts are as popular with websites as they are bad for the effectiveness of CSP to prevent XSS. Bugzilla and HotCRP required substantial changes to support CSP [104], while `addons.mozilla.org` required an effort of several hours [87]. Previous work performed automatic rewriting of .NET applications to better support CSP [32]. Changes to the CSP draft, such as nonce and hash whitelisting of scripts, represent an approach that relieves developers of removing inline scripts while allowing for control over code. Trust relationships in external script sources have been analyzed by Nikiforakis et al. [65]. 88% of the Alexa Top 10K most visited websites included scripts from remote sources, and the most popular single library was included from 68% of the sites. An outlook on the possible future of Web vulnerabilities has been summarized by Zalewski [9]. While CSP addresses a wide range of vulnerabilities, it cannot prevent out-of-order execution of scripts, code reuse through JSONP interfaces, and others.

2.1.3 Content Security Policy

CSP was proposed by Stamm et al. [87], who provided the first implementation in the Firefox browser. Subsequently, CSP became a W3C standard [7] and was adopted by most major browsers. Other publications have addressed limitations of CSP and suggested extensions or modifications to the standard. For instance, Soel et al. [85] proposed an extension of CSP to address shortcomings in `postMessage` origin handling.

2.1.4 Evasion and Attacks Against CSP

Content injection is the main goal of CSP, to prevent third parties from execution of code or adding other unwanted content. Injection is also supposed to prevent leaks of data, which could happen via referrer. Van Acker et al. developed attacks against CSP leaking data bypassing this objective via DNS and resource prefetching [98].

Weichselbaum et al. performed a large scale study on CSP, developing bypasses against deployed policies. Their results indicate that 94.68% of policies that attempt to block XSS fail at it, and 99.34% of policies offer no XSS protection [103].

2.1.5 Beyond Level 1 of CSP

CSP has evolved since the beginning of this study. Recursively enforcing CSP on included documents is supported by embedded enforcement.[1] CSP Level 2 has been adopted by all major browsers and currently Level 3 is an editor's draft.[2]

Research interest in CSP has been active in multiple areas. Finding issues with deployments and fixing them [103], further monitoring growth in deployment [23], generating policies automatically [71], or directly addressing issues with browser extensions [43].

CSP enforces which resources can be included, but dictates no order. However, out of order execution of scripts can lead to unintended behavior. Zalewski outlined this in "Postcards from the post-XSS world" [9]. Lekies et al. [51] have shown this to be a practical attack.

2.2 Browser Extensions

2.2.1 Extension Security Aspects for Browsers

Chrome itself is considered the most secure widely used browser[3], the extension architecture in particular is also stronger than their competitors [42]. Extensions are executed

[1] https://w3c.github.io/webappsec-csp/embedded/

[2] https://w3c.github.io/webappsec-csp/

[3] https://theintercept.com/2016/07/29/a-famed-hacker-is-grading-thousands-of-programs-and-may-revolutionize-software-in-the-process/

in isolation from each other and the content of the page, separated in content scripts and background scripts. They operate on separate JavaScript heaps and can only communicate by message passing. Some aspects of Chrome Extensions are similar to apps for mobile phones, and in particular to Android.

Both stores are confronted by two types of programs that can harm users.[4] One, accidentally introduced mistakes in programs which are created with no harm in mind by the authors. Two, purposefully malicious extensions that harm the user through various means such as leaking browsing history. Users can be lured to install by offering benign functionality. These extensions are in scope for this project.

Security mechanisms that work for one do not necessarily work for the other. For example Content Security Policy (CSP) can prevent bugs in programs from becoming exploitable, but will not help against intentionally malicious extensions. The discussed security measures are all technical, with one exception: the Single Purpose rule. In this section I discuss several mechanisms that secure Chrome extensions.

Permission System Chrome extensions use a permissions system where permissions are statically declared in a manifest file. Extension developers should follow the principle of least privilege, permissions are separated into three levels (low, medium, high).[5] Users need to approve permissions before installing, and updates that add permissions require additional approval. Unfortunately browser extensions have shown to over-specify required permissions [20], and over-reaching permissions are not known to deter users from installation.

Permissions can be used to both protect the author of an extension as the privacy of a user. A benign-but-buggy extension that is exploited reduces the impact of the attack. Conversely, extensions that require low permissions can assure the user that less of their data is accessed.

Content Security Policy for Extensions Chrome extensions leverage Content Security Policy (CSP) [87] to reduce the impact of Cross-Site-Scripting vulnerabilities (XSS). CSP is

[4] https://blog.chromium.org/2009/12/security-in-depth-extension-system.html

[5] https://support.google.com/chrome_webstore/answer/186213?hl=en

a recent web standard with the goal to reduce attacks on web applications. Using CSP for Chrome extensions started as optional feature, and became mandatory in 2012.[6]

Appstore Review of Extensions Before becoming available for download in the Chrome extension store, submitted extensions will be processed by an automated review process. Manual reviews are also possible in some cases.

Automatic Updates Chrome extensions are updated automatically without user interaction, except if extensions request additional permissions. Stale software is under threat by vulnerabilities, and automatic updates can close the window of vulnerability. However, through auto-update and change of ownership Chrome extensions can change content without a user noticing. Miscreants are buying popular extensions on the Chrome store and adding unwanted features.[7]

Visual Indicators Since Chrome version 49, all extensions are represented with an icon right of the URL bar. Using icons was optional before, making it possible for extensions to hide from users while still running in the browser. The move to more prominently highlight possibly forgotten extensions made deinstallation of unwanted extensions easier, a step towards usable security

Centralized Installation and Removal Installation of Chrome extensions is by default only possible through the Chrome store. It used to be possible to install them directly from websites, called sideloading. To prevent accidental installation of malicious extensions, Google moved to only allow installation from the Chrome store, or when operating in developer mode.[8] As a result, malicious extensions cannot spread as traditional malware.

Furthermore, centralized deletion of extensions can be carried out through Chrome. If an extension in the store is detected as malicious after users installed it, it can be removed from clients. Chrome will request updates for the extension, and delete it if necessary.

[6] https://blog.chromium.org/2012/02/more-secure-extensions-by-default.html

[7] http://arstechnica.com/security/2014/01/malware-vendors-buy-chrome-extensions-to-send-adware-filled-updates

[8] https://chrome.googleblog.com/2014/05/protecting-chrome-users-from-malicious.html

Single Purpose The single purpose rule is designed to discourage extensions from misleading users. An example is offering a service to the user (e.g., change color of Facebook) only to obfuscate the true purpose of the extension. For example stealing cookies, or replacing ads to generate revenue for the extension author. This rule, introduced in 2013, is part of the Chrome webstore program policies[9] and was further discussed in a post on the Chromium blog.[10] As an example, replacing ads is not generally disallowed, however, it has to be the primary purpose of an extension.

As a result, authors are motivated to fully disclose the intent of their programs and are disincentivized from hiding potentially unwanted features. This measure is powerful, yet not technical, which distinguishes it from the other security rules listed here. Technical measures cannot reason about the intent of a program. The repercussion is that their extensions can be deleted from the store and will be eradicated through a central deinstallation process. Authors of extensions invest their time and money, an extension with a large userbase can be a financial asset. This is used as leverage, the threat of losing their investments discourages authors from bundling spyware with their programs.

2.2.2 Privacy leaks in Extensions

Previous work on privacy-violating browser extensions has found them to be a prevalent problem. It was shown that official extension store quality checks fail to remove such perpetrators. Blog posts have manually analyzed extensions [28, 105] and Starov et al. studied leaks based on keyword search [88]. Furthermore, Aggarwal et al. created a Recurrent Neural Network in application to browser API calls to identify privacy leaks in browser extensions [14]. In contrast, Ex-Ray does not require searching for particular strings and is oblivious to the protocols used by extensions.

2.2.3 Privacy leaks in Other Platforms

Browsers are not the only platform prone to leaks of private data. In PiOS [33], Egele et al. statically analyzed applications from iOS app stores. While only a few applications were

[9] `https://developer.chrome.com/webstore/program_policies`

[10] `https://blog.chromium.org/2013/12/keeping-chrome-extensions-simple.html`

identified as leaking private user data, more than half leaked unique phone identifiers that can be used by third parties to profile users. Similarly, AndroidLeaks [38] uses data-flow analysis to evaluate Android applications for leaks of private information; they verified leaks in 2,342 applications. Lever et al. show in a longitudinal study of malware [53] that analysis of network traffic is a key factor to early detection.

In the first step of Ex-Ray, I apply linear regression in order to evaluate causality relations [54]. Counterfactual analysis is a relatively simple, but powerful model which has been used in malware traffic analysis before [58]. The authors focused on distinguishing when a certain kind of malware sample acted differently from usual behavior due to certain user activity (triggers). As the work presented noise and some mislabeled conversations, the authors applied Bayesian Inference to assess causality between specific user actions and malware families. In this case, the absence of false positives among the extensions that were not leaking avoided the use of statistical methods to determine whether there is a relation of causality between being a browser extension leaking browser history or not. Linear regression [78] is widely used, for instance as a preparatory step before applying machine learning [86], or as an embedded technique as in SVM [82].

2.2.4 Extension Ad Injection

To monetize extensions, maliciously-inclined authors may add or replace ads in the browser with their own. In 2015, a study found 249 Chrome extensions in the Chrome Web store injecting unwanted ads [96]. The authors identified two drops in their measurement of ad injection. They correlate to Chrome blocking side loading of extensions, and introduction of the single purpose rule to the Chrome store.

2.2.5 Extension Analysis Systems

As with any Web application, browser extensions are third-party code. However, these programs operate with elevated privileges and have access to powerful APIs that can allow access to all content within the browser. Permission systems allow developers to restrict their programs, but extensions have been shown to over-request permissions, effectively de-

sensitivizing users. Heule et al. [44] showed that 71% of the top 500 Chrome extensions use permissions that support leaking private information. They proposed an extension design based on mandatory access control to protect user privacy.

IBEX [42] is a research framework to statically verify access control and data flow policies of extensions. Developers have to author their extensions in high-level type safe languages; .NET and a JavaScript subset are supported. Policies are specified in Datalog and allow for finer-grained control than contemporary permission systems.

Egele et al. [34] used a dynamic taint analysis approach based on the QEMU system emulator to detect spyware in Internet Explorer Browser Helper Objects (BHO). BHOs are classified as malicious if they leak sensitive information on the process level.

Hulk [50] is a system that was used for the first large-scale dynamic analysis of Chrome extensions. The authors introduced the concept of Honeypages. This technique generates Web content tailored to an extension to trigger malicious behavior driven by expectations of the extension.

2.2.6 Tracking: Extensions and the Web

Contemporary websites use a plethora of third-party services that enable developers to quickly add functionality. As a downside, user privacy suffers, since when websites include content from a remote source the trust a user puts into a website is delegated. Nikiforakis et al. [65] studied these delegations and highlighted how widespread this behavior is. As an example, Google Analytics was included in 68% of the top 10,000 websites.

Third-party tracking on websites has been studied extensively. Browsing on seemingly unrelated sites can be observed by third-party trackers and combined into a comprehensive browsing history. Mayer et al. introduced the FourthParty measurement platform [60], discussing privacy implications, technology, and policy perspectives of third-party tracking. Roesner et al. [75] developed client-side defenses to classify and prevent third-party tracking. Recent work has analyzed the history of Web tracking via the Internet Archive's Wayback Machine [52]. The authors found that tracking has steadily increased since 1996. Tracking on the Web has never been as pervasive as it is now.

2.3 Vulnerabilities in Web Applications

2.3.1 Measuring and Reducing Complexity

Software vulnerabilities are often introduced when code becomes too complex to maintain [61, 80]. Managing complexity, measuring it, and trying to reduce it when possible is pertinent to reduction of vulnerabilities. In context of the Web, which started with basic HTML pages that were processed by simple renderers, as it progressed to what we know today. Complexity was introduced on several layers, to the pages served, the backend generating them, and the browsers displaying them.

Stock et al. performed a longitudinal study on how security aspects of the client-side of Web applications evolved over 20 years (2007-2017) via `archive.org`. [89]. They confirmed JavaScript as the main source of growing complexity on the client, other client-side languages did not pass the test of time. Their study of top 500 archived websites concluded that early adopters of security features generally fare better in terms of vulnerabilities.

For the browser-side, Snyder et al. measured how popular usage of JavaScript features is on popular websites [83]. Out of all features offered by FireFox, only 50% were ever used, and 10% are disproportionately used by ads and trackers. In follow-up work the authors built a prototype reducing the APIs a browser exposes, effectively reducing the attack surface [84].

Browsers often only add APIs and rarely remove or reduce them. Rare examples are removal of ambient light sensing in FireFox[11] after it was identified as a privacy threat[12]. Another example is degradation of high resolution timers to prevent the impact of side-channels. All major browsers took action as result of Spectre/Meltdown[13][14][15].

[11] https://www.bleepingcomputer.com/news/software/firefox-gets-privacy-boost-by-disabling-proximity-and-ambient-light-sensor-apis/

[12] https://www.bleepingcomputer.com/news/security/ambient-light-sensors-can-be-used-to-steal-browser-data/

[13] https://www.chromium.org/Home/chromium-security/ssca

[14] https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/

[15] https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/

2.3.2 Single Page Applications

Traditionally Web applications execute program logic on the server based on a client's request, sending a response that is rendered in the browser. Each action the client takes based on that response triggers a new request, resulting in a new response. Continued interaction with a page results in roundtrips which could be perceived as slow.

Through the advent of Web client-side programming languages pages added features that allowed for functionality that would otherwise require sending new requests. Breaking with the request/response paradigm, Single Page Applications (SPAs) are Web applications where the predominant part of program logic is executed in the browser. The server transfers a program, often in JavaScript, which a user can interact with without resulting in a new request/response pair for every action. The browser is not merely rendering a response, it changes the content that is presented to the user, but can also manipulate internal state (e.g.: localStorage). While there is no clear cut-off point at which level of interactivity Web applications are considered as SPAs, several technologies are essential contributors. Most notably, advances of HTML5, in particular asynchronous requests (`XMLHttpRequest`) for background data retrieval, client-side communication channels that enable interaction between different origins, and worker threads that allow for background processing.

2.3.3 Mash-ups or Widgets

Before *Single Page Applications* became to be, *Mash-ups* or *Widgets* were terms for small programs that added limited functionality to a website. More generally, mash-ups enable applications combined of multiple origins.

MashupOS [46] builds on knowledge from Operating Systems proposing a system that can isolate distrusting principals, while establishing communication between trusted ones. The work identifies shortcomings at the time, and goals that MashupOS should achieve to overcome them.

2.3.4 Client-side communication

Frames on pages are isolated based on the Same Origin Policy, limiting interaction between frames. An early workaround for client-side communication between frames was using fragment identifiers (the part of a URL after the `#` sign.) While a frame's content cannot be accessed from frames of different origin, the URL is readable. Changes to the fragment do not trigger a request, this allows a frame to update it's own fragment, while another frame can read it. As a result, creating a client-side communication channel.

This workaround became obsolete through the introduction of `postMessage`, a standard that formalizes communication between frames. Frames can listen to messages sent to them, and send messages addressing other frames. Messages are asynchronous and offer authentication based on a sender's origin.

These communication channels also introduced new vulnerabilities. Barth et. al studied security properties of both fragment identifiers and `postMessage` [21]. Furthermore, `postMessage` is prone to insecure handling of message content, or not verifying authenticity of message senders [85, 108].

2.3.5 Popular JavaScript Frameworks

As web applications migrate to the client and become more and more complex, one of the most common architecture patterns used to organize is the Model View Controller (MVC) architecture [73]. This method encourages separating distinct functions of the application into smaller, more manageable pieces. A variety of JavaScript MVC frameworks has emerged and each provides diverse features and functionality that span a wide range of development needs [59].

2.3.5.1 AngularJS

Angular, which is maintained by Google, is one of the most popular client-side web application frameworks. The premise of Angular is that HTML is not robust enough to create applications, but at the same time full-fledged frameworks tend to take charge and only allow additional code to fill in the details. Rather than taking one of these approaches, Angular

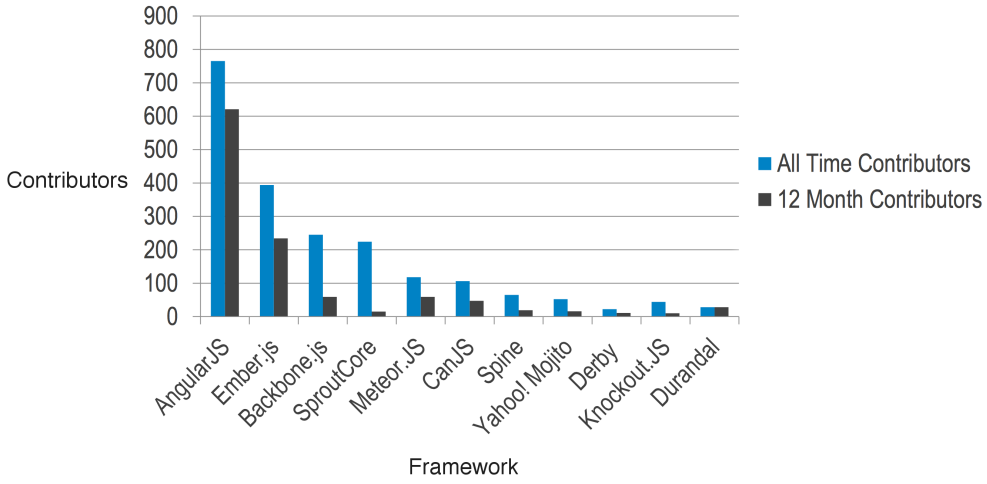

Figure 2.1: Contributors to JavaScript frameworks on GitHub [41].

instead applies markup (HTML) as if it were designed not just for building static pages, but for building apps [30]. This is accomplished by building upon HTML and expanding its syntax by teaching the browser "directives." These directives provide functionality such as data binding, DOM control, form validation, and assigning new behavior to DOM elements [17]. Rather than enforce high-level abstractions, Angular provides primitives useful for building high-level abstractions tailored to the specific application. As such, Angular is a low-level "framework" and intended to be more of a toolset for building application frameworks [16].

2.3.5.2 Backbone.js

The goal of Backbone is to create the minimal set of data structures and user interface elements that are central to building a JavaScript web application. Doing so provides freedom in designing the application rather than having to conform to a framework or library. These minimal elements in Backbone are models and collections (data structures) and views and URLs (user interface) [19]. Backbone's dependency is Underscore.js for RESTful persistence, history support via Backbone.Router, DOM manipulation with Backbone.View, and jQuery is required [36].

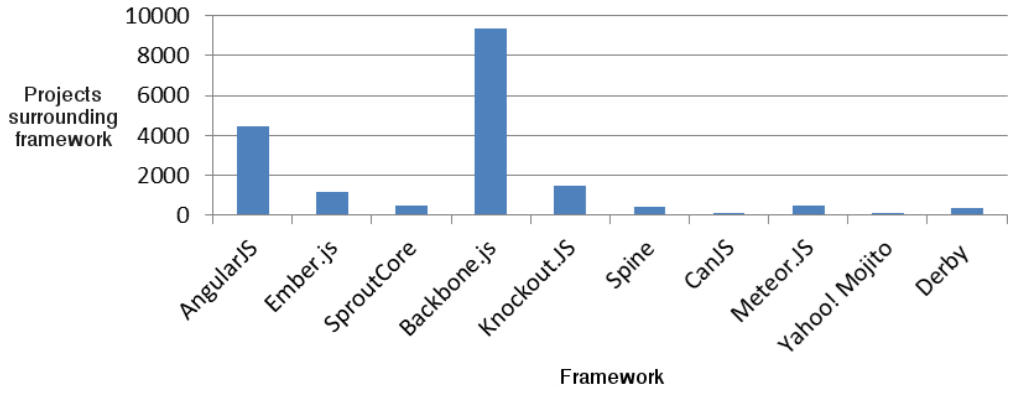

Figure 2.2: Ecosystem of frameworks, demonstrating overall momentum and usage [41] .

2.3.5.3 Ember.js

The philosophy of Ember is that the framework *should* incorporate high-level abstractions and enforce conventions for properly structuring applications. Providing a uniform structure reduces the amount of trivial design decisions that must be made and prevents the need to "reinvent the wheel." In this way Ember values convention over configuration [30]. By intentionally focusing on abstracting away the low-level implementation details, Ember also eliminates the need to write a great deal of boilerplate code, thus allowing the developer to focus on the parts that make the application unique. Ember's dependencies are Handlebars.js and jQuery.

2.3.6 Server-Side Pre-Rendering

Similar work in the area of shifting page rendering to the server exists. For example the Opera browser for mobile offers an option for slow phones to render pages server-side.[16] Furthermore, prerender is a system that uses headless Chrome to render pages server-side to optimize them for SEO or social networks[17]. The difference between these tools and SPARE is that these tools need pages to be navigatable via URLs, otherwise they are inaccessible. Conversely, SPARE exposes all states even when not accessible by URL.

[16] `https://www.opera.com/mobile`
[17] `https://github.com/prerender/prerender`

2.3.7 Vulnerability Scanners

Web application vulnerability scanners can be divided into three main categories. Black-Box scanners which operate disregarding source-code analysis, white-Box scanners that utilize source-code analysis techniques, and Gray-Box scanners which use a combination of the two. All scanners considered in this project are considered black-box vulnerability scanners.

Another category vulnerability scanners can be divided into are dynamic and static. Dynamic scanners execute a program with concrete input, and observe results. Static scanners analyze a program without executing it, or executing it with abstract values. Advantages of dynamic analysis are that results are immediately observable, however, it is challenging to find input that triggers relevant program areas. Static analyzers use abstract values that can be refined based on desired results, but this leads to problems such as path explosion that can make analysis infeasible. Scanners for the Web are typically dynamic, and the remainder of this section will focus on dynamic analysis black-box vulnerability scanners.

2.3.7.1 Black-Box Vulnerability Scanners

Black-box vulnerability scanners are often promoted as "point-and-click" pentesting tools that require little or no human interaction in evaluating the security of web applications. Previous research analyzing the effectiveness of black-box vulnerability scanners in discovering vulnerabilities in traditional client-server web applications demonstrated that these scanners are full of limitations and fall far short of the "point-and-click" tools that they aim to be. This research additionally concluded that the ability to crawl a web application is as important as the actual ability to detect vulnerabilities [31].

Black-box vulnerability scanners consist of three main modules—the crawler, the attacker, and the analyzer. In this research I focus primarily on the crawler, as it is vital for a scanner to have a capable crawler in order to perform a thorough analysis. If a black-box vulnerability scanner cannot reach a part of the web application, then it can never find a vulnerability there.

The Crawl Module The crawling module is directed to a starting URL, from which it follows all links on that page, then retrieves those web pages and follows any further links in

order to establish a list of all attainable pages in the application. If the crawler successfully discovers a page in the application, the attacker module has the potential, based on the effectiveness of its attacking and analyzing modules, to discover any vulnerabilities within the given page. However, if the crawler does not discover a page in the application, the other modules have no opportunity to discover any vulnerabilities within that page regardless of their effectiveness.

The Attack Module Once a scanner completes the crawling process and analyzes each discovered page for application input, it subsequently carries out an automated audit for vulnerabilities by emulating an attacker [13]. One common approach that these tools employ is "fuzzing," in which they bombard the application's input parameters with input data in an attempt to trigger a vulnerability. Fuzzing is a simple technique that offers a high benefit-cost ratio, especially because it accesses a web application in the same way users do and can be utilized independent of the technology that powers the web-application [31, 76]. Although more sophisticated tools use intelligent heuristics to perform their attacks, fuzzing alone cannot provide a complete picture of the overall security and must be used in conjunction with other techniques.

The Analyzing Module For every input sent by the attacking module, the analyzing module inspects the changes triggered in the web application to determine if a vulnerability was triggered. Once the scanner has completed its work, it typically presents the user with a list of vulnerabilities and the pages of the application that contain them.

2.3.8 Analysis of Web Vulnerability Scanners

Although this work focuses on Web vulnerability scanners in a client-side environment, there exists ample research evaluating these tools in other fashions. In a study of vulnerability assessment tools, Curphey and Araujo reported lackluster performance of black-box scanners [27]. Peine surveyed the functionality and interfaces of seven scanners [72]. Kals et al. designed a new vulnerability assessment tool and evaluated it against approximately 25,000 Web pages [49], but no concrete metrics can be discussed because no definitive statis-

tics are obtainable for these sites. Using metrics such as code coverage, Suto tested three scanners against three applications to ascertain their success [92]. Additionally, Suto tested seven scanners and used their run time and detection capabilities to compare them [93]. Five unnamed tools were run against a tailored benchmark by Wiegenstein et al., but the causes for the failures are not explained [109]. AnantaSec assessed three scanners against a variety of Web applications and JavaScript tests, yet no clarification is provided for the results [15]. Vieira et al. tested four scanners against 300 Web services and asserted high false positive and false negative rates [99]. Doupè et al. evaluated eleven black-box tools against a custom vulnerable Web application and reported that many vulnerabilities are overlooked by security tools, as well as that these tools lack the ability to thoroughly crawl applications [31]. Bau et al. conducted an analysis of eight black-box scanners to determine their relevance and effectiveness upon vulnerabilities in the wild, but did not draw comparisons from their results [22]. Chen maintains a website that list a high-level comparison of black-box vulnerability scanners [24].

Outside of the real of Web applications, a promising approach is presented by LAVA [29]. It is a system that generates synthetic vulnerabilities to evaluate bug finding tools. The system operates on C source code and was applied to real-world programs. The tools under evaluation found some of the bugs, but not all, highlighting shortcomings in the state-of-the-art in bug finding tools.

Chapter 3

Investigating Content Security Policy

3.1 Introduction

The Web as a platform for application development and distribution has evolved faster than it could be secured. Consequently, it has been plagued by numerous classes of security issues, but perhaps none are as serious as content injection attacks. Content injection, of which cross-site scripting (XSS) is the most well-known form, allows attackers to execute malicious code that appears to belong to trusted origins, to subvert the intended structure of documents, to exfiltrate sensitive user information, and to perform unauthorized actions on behalf of victims. In response, many client- and server-side defenses against content injection have been proposed, ranging from language-based auto-sanitization [77] to sandboxing of untrusted content [62] to whitelists of trusted content [48].

Content Security Policy (CSP) is an especially promising browser-based security framework for refining the same-origin policy (SOP), the basis of traditional Web security. CSP allows developers or administrators to explicitly define, using a declarative policy language, the origins from which different classes of content can be included into a document. Policies are sent by the server in a special security header, and a browser supporting the standard is then responsible for enforcing the policy on the client. CSP provides a principled and robust mechanism for preventing the inclusion of malicious content in security-sensitive Web applications. However, despite its promise and implementation in almost all major browsers, CSP is not widely used in practice—in fact, according to my measurements, it is deployed

in enforcement mode by only 1% of the Alexa Top 100.

In this part of my dissertation, I present the results of a long-term study to determine why this is the case. In particular, I repeatedly crawled the Alexa Top 1M to measure adoption of Web security headers, and find that CSP significantly lags behind other, more narrowly-focused headers in adoption. I also find that for the small fraction of sites that have adopted CSP, it is often deployed in a manner that does not leverage the full defensive power of CSP.

In addition to this Internet-scale study, I also quantify the feasibility of incrementally deploying CSP from the perspective of a security-conscious administrator using its report-only mode at four websites. Although this is an oft-recommended practice, I find significant barriers to this approach in practice due to interactions with browser extensions and the evolution of Web application structure over time.

Finally, I evaluate the feasibility of automatically generating CSP rules for Web applications, again from the perspective of an administrator. I find that for websites that are well-structured and do not change significantly over time, rules can indeed by generated in a black-box fashion. However, for more complex sites such as those that make use of third-party advertising libraries in their proper site context, policy generation is significantly more difficult.

To summarize, the contributions of this study are the following:

- I perform the first long-term analysis of CSP adoption in the wild, performing repeated crawls of the Alexa Top 1M over a 16 month period.
- I investigate challenges in adopting CSP, and why it is not deployed to its full extent even when it has been adopted.
- I evaluate the feasibility of both report-only incremental deployment and crawler-based rule generation, and show that each approach has fundamental problems.
- I suggest several avenues for enhancing CSP to ease its adoption.

Table 3.1: The types of directives supported in the current W3C standard CSP 1.0.

Directive	Content Sources
`default-src`	All types, if not otherwise explicitly specified
`script-src`	JavaScript, XSLT
`object-src`	Plugins, such as Flash players
`style-src`	Styles, such as CSS
`img-src`	Images
`media-src`	Video and audio (HTML5)
`frame-src`	Pages displayed inside frames
`font-src`	Font files
`connect-src`	Targets of XMLHttpRequest, WebSockets

3.2 Content Security Policy

The goal of CSP is to mitigate content injection attacks against Web applications directly within the browser [7, 87]. In the following, I describe CSP as it is currently implemented, and briefly discuss both future extensions and the classes of attacks it is intended to prevent.

3.2.0.1 Overview of CSP

Content Security Policy is fundamentally a specification for defining policies to control where content can be loaded from, granting significant power to developers to refine the default SOP. Developers or administrators can configure Web servers to include `Content-Security-Policy` headers as part of the HTTP responses issued to browsers. CSP-enabled browsers are then responsible for enforcing the policies associated with each resource.

A content security policy consists of a set of directives. Each directive corresponds to a specific type of resource, and specifies the set of origins from which resources of that type may be loaded. Table 3.1 explains the directive types supported in the current W3C standard CSP 1.0.[1] The scheme and port in source expressions are optional.

CSP also supports wildcards (∗) for subdomains and the port, and has additional special keywords: `'self'` represents the origin of the resource, while `'none'` represents an empty resource list and prevents any resource of the respective type from being loaded. The `script-src` and `style-src` directives additionally support the `'unsafe-inline'` keyword,

[1]The directive `script-src http://seclab.nu:80`, for instance, allows a protected website to load scripts from the host `seclab.nu` via HTTP on port 80, but blocks all scripts from other sources.

which allows inline script or CSS to be included in the HTML document rather than being loaded from an external resource. Finally, `'unsafe-eval'` allows JavaScript to use string evaluation methods such as `eval()` and `setTimeout()`. If not explicitly whitelisted, CSP disables these special source types because their use is considered to be particularly unsafe. However, changing websites to remove all inline scripts can be a burden on developers, and increase page load latency by introducing additional external resources.

CSP can operate in one of two modes: *enforcement* or *report-only*. In enforcement mode, compatible browsers block resources that violate a policy. In report-only mode, however, browsers do not enforce policies, but rather report violations that would be blocked on the developer console. Additionally, a special CSP directive (`report-uri`) can be used to instruct browsers to send violation reports to the given URI. This feature can be used to learn policies before enabling enforcement, or to monitor for unforeseen changes or attacks against a website. In this project, I make extensive use of the report-only mode and violation reports to explore various ways to (semi-)automatically generate policies for websites.

CSP has been widely adopted by the browser manufacturers. It is supported by the current versions of almost all major browsers, including some mobile browsers. It is, however, only partially supported by Internet Explorer.

3.2.0.2 Deploying CSP

To prevent XSS attacks, disallowing inline scripts and `eval` is the core requirement to benefit from CSP. Inline scripts should be disabled to prevent the browser from inadvertently executing scripts that have been injected into the site. Eval-constructs, often abused to parse JSON strings, can be used directly by an attacker to execute arbitrary code if she controls the data source. While the `unsafe-inline` and `unsafe-eval` options allow this behavior to be enabled, their presence marginalizes the benefit of CSP.

Therefore, for version 1.0 of CSP, inline scripts should be moved to files and `eval` replaced with a safe equivalent for the corresponding task, such as `JSON.parse()` to parse JSON. Furthermore, JavaScript should be hosted on a domain that only serves static files instead of user content. This separation makes it harder for attackers to execute code in the browser. Also, external scripts should be moved to a server controlled by the website owner, reducing

trust in third-party servers. The number of whitelisted sources should be kept to a minimum to increase the difficulty of data exfiltration for attackers.

In the current draft version 1.1, additional features have been introduced to safely support inline scripts as well as functionally replace the `X-Frame-Options` header. As these features are subject to change, I do not address them in this work.

3.2.0.3 Attacks Outside the Scope of CSP

CSP can prevent general content injection attacks, and in draft version 1.1 subsumes previous mechanisms such as the `X-XSS-Protection` header, which serves the narrow purpose of enabling browser XSS filters. However, it is not intended to address other Web attacks such as cross-site request forgery (CSRF). More fundamentally, CSP describes which content can be loaded by source, but the order of inclusion is out of scope. Hence, even with strict rules and perfect enforcement, out-of-order inclusion can lead to undesired side effects in JavaScript applications [9]. JSONP (JSON with padding) is a mechanism to bypass SOP restrictions by including a script tag from a remote server and specifying a function to be executed once a result becomes available. Hence, whitelisted JSONP sources can be used for calls to arbitrary functions—or, if input for the callback function is not filtered, arbitrary code execution.

3.2.1 Usage of HTTP Security Headers

In this section, I describe the data collection of HTTP response headers. I collected this data in an effort to understand the landscape of security headers in the wild, particularly in regards to CSP. The headers are described in detail in Section 2.1.

3.2.1.1 Methodology

To acquire a long-term overview of CSP adoption, I performed weekly crawls of the Web starting in December 2012. I crawled the front page of each site in the Alexa Top 1M most frequently visited websites. For every site `x`, I connected to `http://x`, `https://x`, `http://www.x`, and `https://www.x`. I counted a site as using a particular header if any of the

four responses served that header. However, the crawler only visited the front page of each Alexa entry. Therefore, sites that employ CSP only on subdomains or areas other than the front page were not detected in the crawl.[2] Furthermore, if the CSP rules are generated based on user agent discrimination, the collected data does not hold for all types of browsers visiting the site. I used a Firefox user agent string, updating version information over time.

3.2.1.2 Adoption of HTTP Security Headers

To measure the popularity of CSP in contrast to other security headers, I looked at the HTTP response headers in the weekly crawls, as well as a static snapshot from the end of March 2014. For the static snapshot, I used the entire Alexa Top 1M, breaking down websites by popularity. I used a snapshot of the Top 10K to track the evolution of response headers back to December 2012.

To compare the adoption of security-related headers between different levels of site popularity, I split Table 3.2 into brackets. From the data, it is apparent that websites that are less popular use CSP less frequently. For instance, among the 100 most popular sites, only two used CSP (2%), while CSP was enabled for only 775 among the 900,000 least popular sites (0.00086%).

Hence, websites that are less popular use CSP less frequently. In contrast, for CORS, header usage was more evenly spread out, with all brackets between 0.7% and 2.6%.

During the crawls, I noticed that Google enabled CSP headers only occasionally. I performed an additional test of `google.com` with 1,000 requests, finding that 0.8% of the responses included CSP headers. While Google had 18 sites in the top 100, none of them issued CSP headers in the crawl of Table 3.2.

In Figure 3.1, I track the evolution of security-related headers of the Alexa Top 10K from March 2014 backwards in time to December 2012. P3P was particularly popular; however, the P3P policies served were often invalid, providing only an explanation for why the website did not support it. I observe that CSP is only slowly gaining traction over time. The main contributing factor for the fluctuation of CSP headers in the data is due to Google.

[2]One example is Twitter, which uses CSP for parts of their site, but not the front page.

Table 3.2: Number of websites with security-related HTTP response headers, grouped by intervals of site popularity, for the Alexa Top 1M ranking.

Header / Alexa Rank	$[1-10^2]$	$(10^2-10^3]$	$(10^3-10^4]$	$(10^4-10^5]$	$(10^5-10^6]$
P3P	47	176	849	6,315	79,600
DNS Prefetch Control	1	0	3	40	461
XSS Protection	26	77	269	2,336	43,045
Content Type Options	10	27	172	1,995	42,150
Frame Options	43	165	581	2,747	21,746
HSTS	5	16	83	476	2,475
CORS	1	26	217	1,228	7,149
CSP	2	2	15	57	775
Any security header	66	304	1,623	11,491	132,347

For the hosts in the Top 10K of this crawl, I identified all servers that had sent CSP rules at any point in time during this study. I found 140 sites that did so; 110 of those belonged to Google (79%).

3.2.1.3 Detailed Analysis of CSP Headers

In this part, I describe in detail how websites use CSP, whether they use CSP's reporting feature to learn policies, whether they actively enforce policies, and how effective those policies are in mitigating attacks.

Enforcement vs. Report-Only. During this crawl at the end of March, I found 815 sites in enforcement mode, 35 sites in report-only mode, and no sites that sent both types of headers. Out of the websites in enforcement mode, only 23 collected violation reports.

In the Top 10K, I observed only one site in report-only mode that later switched to enforcement. The Norwegian financial services site `dnb.no` started collecting reports in June 2013, and enabled enforcement in February 2014. Their enforced `default-src` directive consists of 74 sources, including the schemes `chrome-extension`, `chromeinvoke`, and `chromeinvokeimmediate`. Furthermore, `unsafe-inline` and `unsafe-eval` are both enabled. Therefore, this policy appears to provide little benefit over not using CSP at all.

I noticed that several websites use CSP to test for mixed content. Mixed content is the inclusion of unencrypted content into HTTPS sessions, which reduces the benefit of encryption. Google's sampling uses the following report-only policy: `default-src https: data:;`

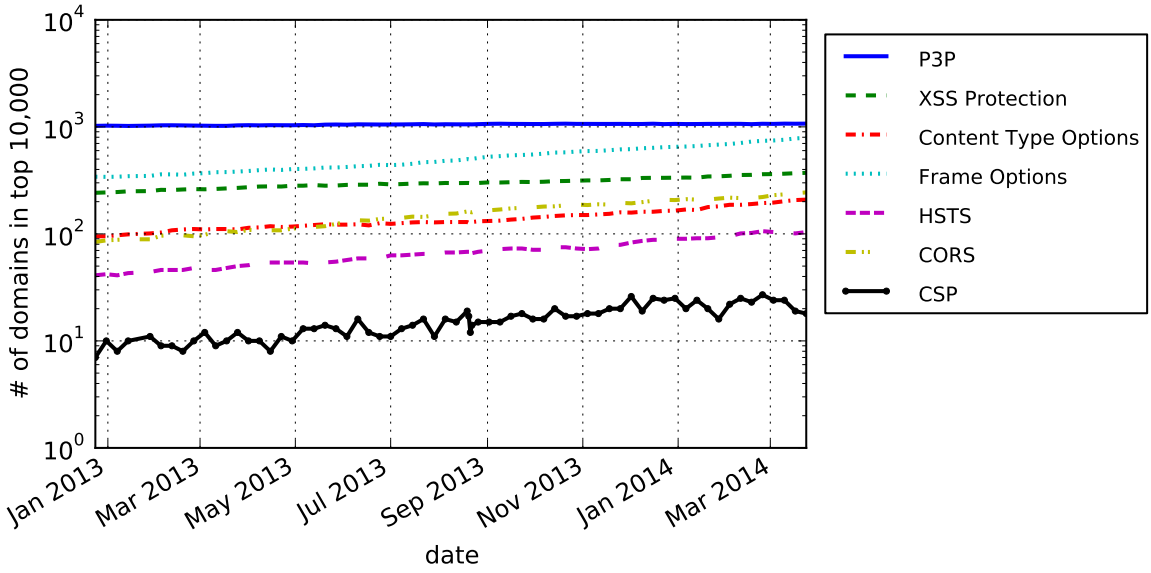

Figure 3.1: Popularity of security headers in the Alexa Top 10K.

`options eval-script inline-script; report-uri /gen_204?atyp=csp`. Etsy also samples for mixed content; I found CSP headers in nine out of 2,000 (0.45%) responses. Similarly, `hootsuite.com` tested for mixed content from April 2013 to March 2014 for all responses, but I observed no CSP headers after that.

Types of Sites Using CSP. To further understand the types of websites that use CSP, I looked for similarities in website titles. The largest portion of sites supporting CSP, 417, is due to phpMyAdmin, a PHP-based Web application used to manage MySQL databases. phpMyAdmin ships with CSP enabled by default, which allows inline scripts, `eval`, and restricts sources to `'self'`. While this policy does not prevent XSS, data exfiltration is more difficult. These rules can be deployed as the software is fairly static. However, when conducting a search for phpMyAdmin and CSP, I found users having trouble including images when modifying their installations. The general solution offered was to disable CSP in the configuration rather than updating the default policy.

Ironically, on the vendors' demo site `http://demo.phpmyadmin.net/master/`, the operators tried to include Google analytics. While the Google analytics domain is whitelisted using `default-src`, it is not in the `script-src` source list. As specific directives override the

Table 3.3: Overview of enforced policies.

Feature / Alexa Rank	$[1 - 10^4]$	$(10^4 - 10^6]$
unsafe-eval	8	700
unsafe-inline	11	728
script-src 'self'	12	789
no report-uri	10	782
#script-src > 10	2	33
*** as source**	6	230
Median #directives	6	4
Median #script sources	4	1
# CSP Policies	13	802

`default-src` directive, the script is unintentionally blocked.

I also found 170 OwnCloud instances, which uses CSP by default from version 5.

Prevalence of Unsafe Policies. I identified several patterns in CSP policies that violate deployment best practices as described in Section 3.2.0.2. In Table 3.3, I summarize the observed rules in enforcement over the Alexa Top 1M from March 24th. I split at the 10K rank to discriminate between more popular websites and lower ranking ones. '*' represents either the literal asterisk, or the entire HTTP(S) scheme is whitelisted in one or more of the source lists.

On the majority of sites, `eval` and inline is enabled: eight out of 13 and 11 out of 13 in the Top 10K bracket, 700 out of 802 and 728 out of 802 in the remaining 990,000 sites. This configuration strongly reduces the benefits of CSP for XSS mitigation. Configuring asterisk or a whole scheme as a source in a directive enables data leakage to any host. Six out of 13 and 230 out of 802 websites respectively served such directives. 10 out of 13 sites in the Top 10K bracket had no `report-uri` to collect violation reports. This is surprising as CSP could be used as a warning system.

While CSP in theory can effectively mitigate XSS and data exfiltration, in practice CSP is not deployed in a way that provides these benefits.

3.2.1.4 Conclusions

While some sites use CSP as an additional layer of protection against content injection, CSP is not yet widely adopted. Furthermore, the rules observed in the wild do not leverage the full benefits of CSP. The majority of CSP-enabled websites were installations of phpMyAdmin, which ships with a weak default policy. Other recent security headers have gained far more traction than CSP, presumably due to their relative ease of deployment. That only one site in the Alexa Top 10K switched from report-only mode to enforcement during the measurement suggests that CSP rules cannot be easily derived from collected reports. It could potentially help adoption if policies could be generated in an automated, or semi-automated, fashion.

3.3 CSP Violation Reports

Web browsers compatible with CSP can be configured to report back to the website whenever an activity, whether carried out or blocked, violates the site's policy. This is meant as a debugging mechanism for Web operators, both to develop policies from scratch, and to be informed when an existing policy needs to be updated. Starting with a "deny all" policy in report-only mode, operators can collect information about all resources that need to be whitelisted in order for the site to function, compile a corresponding policy, and eventually switch to enforcement mode. I applied this approach to four websites and analyzed the reports that I received, gaining unexpected insights into the Web ecosystem.

3.3.1 Background

CSP includes an optional `report-uri` directive that allows website operators to specify a sink for violation reports. It is supported in both report-only and enforcement mode of CSP. As an illustration, consider the following policy: `img-src 'none'; report-uri /sink.cgi`. When a user visits the URL `http://seclab.nu/test.html` and that page includes the image resource `http://seclab.nu/pic.gif`, the browser would send a report similar to the following one: `{"blocked-uri": "http://seclab.nu/pic.gif", "violated-directive":`

"img-src 'none'", "document-uri": "http://seclab.nu/test.html", …}. From this report, the developer can infer that the policy entry `img-src http://seclab.nu` should be added to the policy.

3.3.2 Methodology

I deployed CSP on four websites I had access to: two personal pages, an institutional page, and a popular analysis service. The policies I used specified empty resource lists for all supported directive types—that is, any browser activity covered by CSP was explicitly forbidden and should generate a report. I deployed the policies in report-only mode to not interfere with the normal operation of the site. Besides the additional CSP headers, the sites were not modified in any way.

During this analysis, I observed that the formats of reports sent by different browser versions varied slightly. Older Firefox versions, for instance, explicitly stated when a violation was due to the special cases `'unsafe-inline'` or `'unsafe-eval'` for script and style directives, as opposed to violations based on a resource URI. All recent versions of browsers, however, reported only an empty `blocked-uri` instead. Unfortunately, this format did not allow us to distinguish between `'unsafe-inline'` and `'unsafe-eval'` script violations.

In order to work around this issue, I leveraged the fact that recent browser versions supported multiple CSP headers in parallel. That is, in addition to the *regular* policy discussed above that captured any CSP event, I added two more policies that caused reports only for *eval* and *inline* violations, respectively:

```
   default-src *; script-src * 'unsafe-inline';
style-src * 'unsafe-inline'; report-uri /sink.cgi?type=eval
   default-src *; script-src * 'unsafe-eval';
style-src *; report-uri /sink.cgi?type=inline
```

I deployed all three policies and distinguished the reports I received using the type parameter in the report URI. I removed duplicate eval and inline violations that were reported for the *regular* policy (30 % on site D). Furthermore I removed some violations reported for the *eval* and *inline* policies that were in fact no eval or inline violations (1.8 % on site D). Those were triggered by a bug in older Firefox versions that did not properly execute

Table 3.4: Overview of the CSP violation report data sets received from partner websites in early 2014, after removing inconsistent reports.

Site	A	B	C	D
Type	personal	personal	institutional	service
# Reports	1.1 K	21.8 K	48.0 K	7.1 M
Median Reports/Day	9	671	2.1 K	348.5 K
# IP Addresses	78	1.6 K	1.2 K	14.4 K
Median Reports/Addr.	7	7	28	85
% Reports/Browser				
Chrome (mobile, derivatives)	46.6 (+5.4)	59.3 (+8.3)	54.3 (+3.7)	61.0 (+2.3)
Firefox (mobile, derivatives)	23.8 (+0.5)	22.2 (+0.6)	30.1 (+0.5)	30.3 (+0.2)
Safari (mobile)	5.7 (+2.3)	2.3 (+3.5)	4.1 (+3.7)	1.5 (+0.5)
Opera	0.5	0.3	0.6	1.9
Googlebot	15.1	3.1	2.0	2.1

multiple policies in parallel. Since newer Firefox versions were not affected, the user agent distributions of the original and the filtered data set were very similar. Table 3.4 shows the number of reports retained in the filtered data set, which is the basis for the following discussion.

From each report, I derive a policy entry that whitelists the respective violation. I extract the type, such as `img-src`, from the `violated-directive`. For *regular* violations, I append the scheme, host name and port from the `blocked-uri`, such as `http://seclab.nu`. For *inline* or *eval* violations, I append `'unsafe-inline'` or `'unsafe-eval'`. I generate a single policy per site by combining all entries and set `default-src 'none'` to block everything else.

This approach is to generate one single policy that is general enough to cover the entire protected site. Such a site-wide policy is easier to generate than individual policies, since any similarity between pages on the same site reduces the number of violation reports necessary to generate a policy. Furthermore, site-wide policies are easier to configure; a site-wide reverse proxy could insert a static policy into HTTP responses without the need to change application code.

Table 3.5: Length of policies when whitelisting all violations from the report data set (a), and with an additional filter for URL schemes of browser extensions (b). Most of the policy entries correspond to injected resources; only few are intended to be included. (In brackets, the number of unique policy entries when disregarding the protocol HTTP(S) or alternative domains, such as the www subdomain.)

Site	A	B	C	D
# Entries (a)	14	221	226	1,113
# Entries, extension filter (b)	14	212	215	1,090
Correct Subset	3 (3)	14 (9)	38 (13)	22 (9)

3.3.3 Results

Table 3.5 summarizes the policies I generated for each of these sites. I verified manually each entry in the policies and found that many of the whitelisted resources were not actually intended to be included in the websites. The policy generated for site A, for instance, is `default-src 'none'; frame-src https://sr v.mzcdn.com; img-src 'self' data: http://1.2.3.11; object-src http ://www.ajaxcdn.org; script-src 'unsafe-eval' 'unsafe-inline' http: //ajax.googleapis.com http://f.ssfiles.com http://i.bestoffersjs.i nfo http://srv.mzcdn.com http://www.superfish.com https://www.super fish.com; style-src 'unsafe-inline'`. Yet, site A was entirely static and did not contain any script at all. The correct policy for site A would have been `default-src 'none'; img-src 'self' data:; style-src 'unsafe-inline'`. In other words, only 21 % of the policy entries generated from the received reports were legitimate.

On site D, only 2 % of the policy entries were legitimate. Furthermore, many of the legitimate entries simply enumerated all the alternative domain names of the same site (e.g., with or without the `www` subdomain), or they were due to the same resource being loaded over HTTP or HTTPS. When disregarding these details to allow for a fairer comparison, as noted in brackets in the table, the percentage of legitimate policy entries drops to only 0.8 % on site D.

Table 3.6: Most frequent Chrome extensions observed at site D.

Name	# Reports
AdBlock	38 K
AdBlock Plus	29 K
Grooveshark Downloader	9.5 K
ScriptSafe	8.8 K
DoNotTrackMe	8.2 K

3.3.3.1 Reasons for invalid policy entries

I identified a number of reasons why Web browsers sent CSP violation reports for resources that did not exist in the original websites. Many of these reports appeared to be caused by browser extensions that modified the DOM of the page by injecting additional resources such as scripts or images. I observed extensions for blocking advertisements, extensions injecting advertisements, price comparison toolbars, an anti-virus scanner, a notetaking plugin, and even a BitTorrent browser extension. I could automatically identify some browser extensions based on violation reports because they attempted to load resource URIs that contained the `chrome-extension` or `safari-extension` schemes followed by the unique identifier of the extension. AdBlock and AdBlock Plus were the most frequent extensions for the Chrome browser (Table 3.6), while the most frequent Safari extension was Evernote. Yet, automatically removing these reports (and a few other unexpected schemes, such as `about` and `view-source`) accounted for fewer than 5 % of all incorrect policy entries, as shown in the second row of Table 3.5. The remaining browser extensions exhibited no such uniquely distinguishing features, often injecting libraries that are used not only in browser extensions but also in many websites, such as Ajax tools, Google Analytics, and resources from large content distribution networks.

When browsers send violation reports for modifications due to browser extensions, the reverse conclusion is that websites enforcing CSP can cause browser extensions to stop functioning. Some browser extensions thus intercept CSP headers and modify them in order to whitelist their own resources or disable CSP. I observed reports caused by one such extension, which were sent because the modification resulted in a semantic error. I cannot quantify how often such modifications were successful as they are not observable with this methodology.

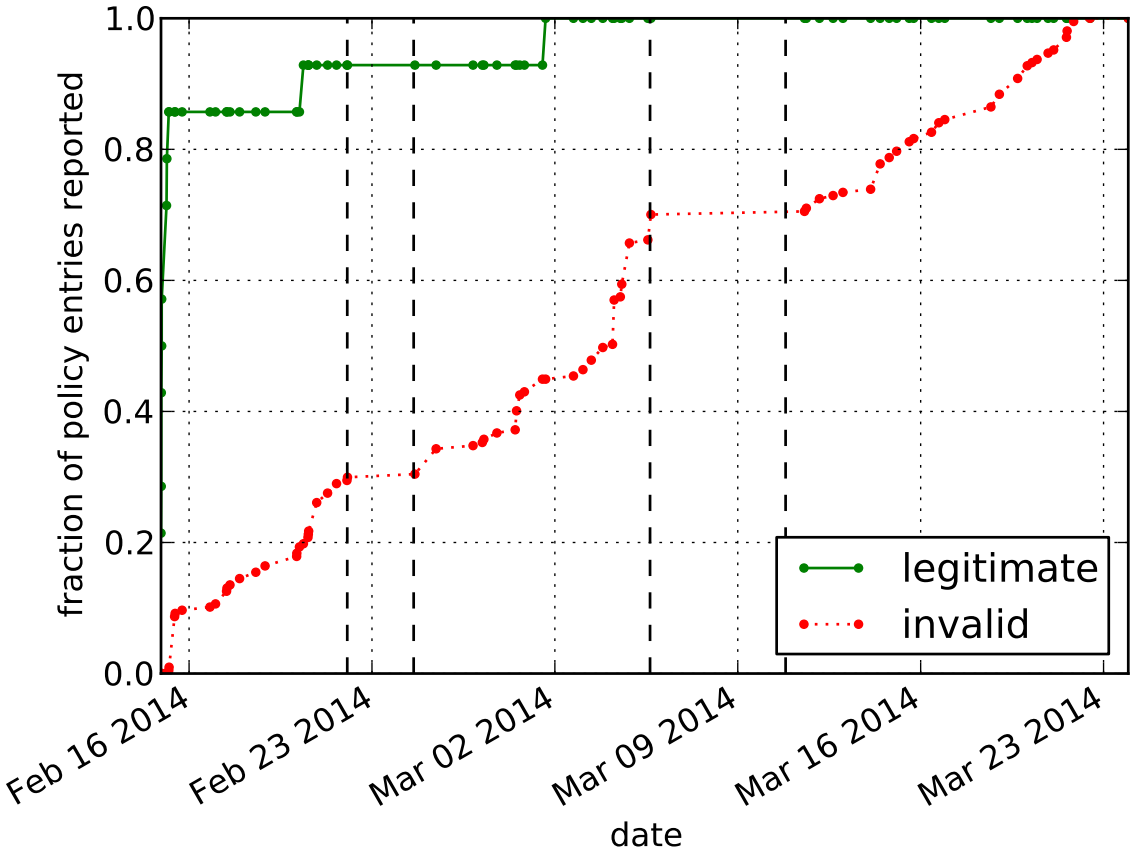

Figure 3.2: Fraction of new policy entries discovered over time on site B (measurement inactive during the dashed intervals). It can take some time until all legitimate resources have been accessed at least once; in the meantime, many injected resources are reported.

In addition to browser extensions, "in-flight" modification of pages by ISPs or Web applications such as anonymity proxies can also cause violation reports. The image loaded from `1.2.3.11` in the example above appeared to be injected by a mobile Internet provider. These examples illustrate that even when CSP violations due to browser extensions were filtered (or not reported by the browsers), other non-attack scenarios can still cause websites to receive spurious reports. Administrators who plan to generate a policy from reports submitted by their visitors' Web browsers may need to manually verify a large number of policy entries in order to avoid accidentally whitelisting resources injected by browser extensions or ISPs (let alone attackers).

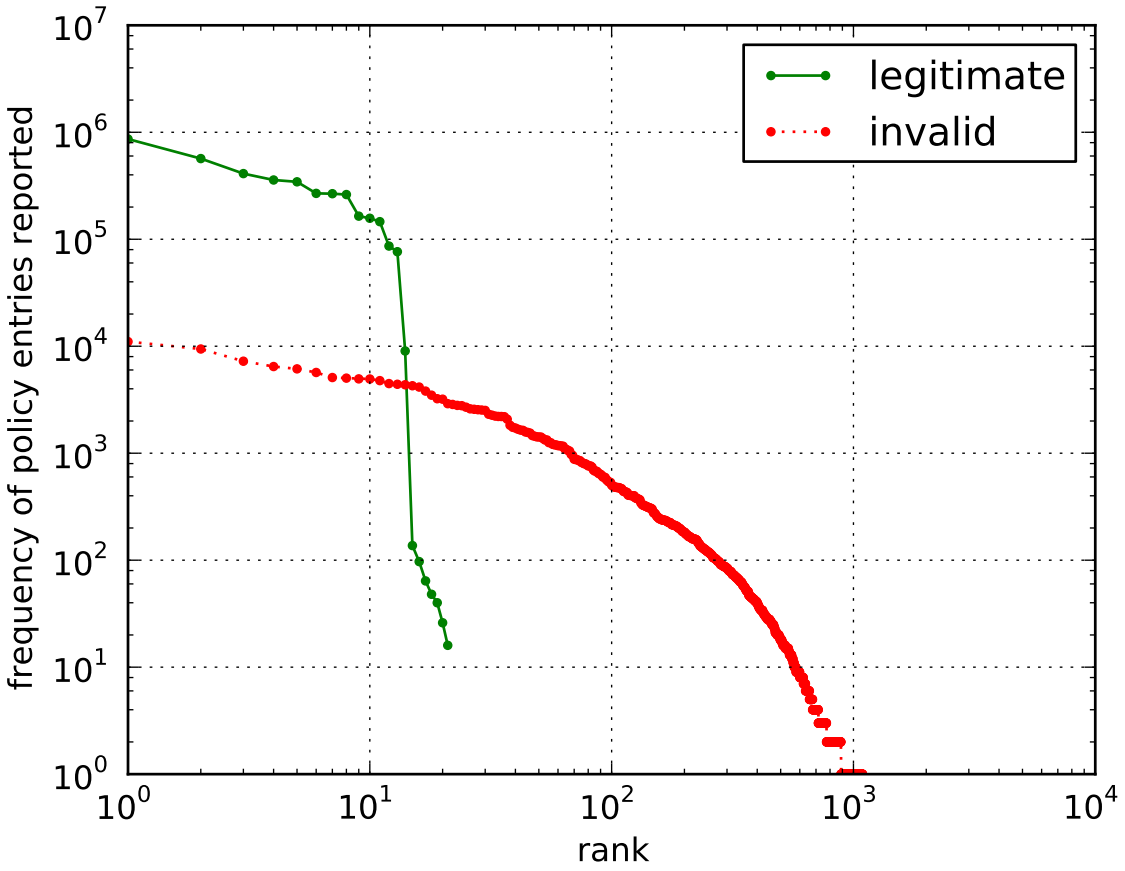

Figure 3.3: Frequency of legitimate and invalid violations being reported on site D. Some injected resources occurred orders of magnitude more often than legitimate resources.

3.3.3.2 Time delay until a policy can be generated

On site B, it took around two weeks to receive at least one report for each valid policy entry. The last resource that was discovered was an embedded YouTube video. Another resource that was discovered relatively late was an image loaded over HTTPS instead of HTTP; all other valid policy entries could be generated within the first two days of the measurement. For the other sites, the durations were similar. In practice I expect these numbers to vary, thus website operators will need some prior knowledge about the resources used on their website so that they can decide when it is safe to switch from report-only to enforcement mode without causing any disruption. Operators could therefore be tempted to run the observation period for as long as possible in order to minimize the risk of not

receiving reports for legitimate resources. However, as Figure 3.2 shows, the rate of newly observed, invalid policy entries remained relatively constant over time, suggesting that longer measurement periods can significantly increase the number of policy entries an operator needs to verify manually.

3.3.3.3 Report frequency as a (poor) distinguishing feature

Only about 4 % of all reports received on site D during this measurement resulted in an invalid policy entry. Hence, one might attempt to use the frequency of a report as an indicator for its validity. However, this approach would be problematic for two reasons. First, an attacker can easily influence the frequency distribution observed by the website by submitting forged reports. Second, even in the absence of attacks, resources injected into websites can be so popular that they cause reports more often than some legitimate, but infrequently accessed, resources.

Figure 3.3 visualizes this phenomenon. The most frequently injected resource (a script loaded from `superfish.com` for price comparison) was reported more than 22,000 times. In contrast, `connect-src 'self'`, which is used by a progress meter on the site, was reported only 9,000 times, and reports corresponding to alternative domain names of site D were received even less frequently.

3.3.4 Conclusions

Websites small and large observe CSP violation reports for injected resources. Even in the absence of ostensibly malicious activity, which I did not observe, the high number of injected resources complicates the process of generating a viable policy from the received reports. At the moment, this task is mostly a tedious and, from my own experience, error-prone manual process. As a semi-automated approach to filtering reports, it might be possible to generate signatures for the most common browser extensions, either manually or by leveraging the fact that an installed browser extension usually causes several violations to co-occur (based on time, IP address, and user agent signature). These signatures could be shared with the community and could be used to reduce the number of reports that need to be verified

manually.

3.4 Semi-Automated Policy Generation

An alternative approach to generating a policy from appropriately filtered and verified reports submitted by visitors is to make use of trustworthy reports only. In order to explore this approach further, I developed a proof-of-concept Web crawler that generates violation reports in a controlled environment.

3.4.1 Methodology

The crawler is implemented as an extension for the Chromium browser based on Site Spider, Mark II. The crawler follows at most 500 internal links on the main domain of the crawled site in a non-randomized breadth-first search. After navigating to a page, the crawler pauses for 2.5 s to load all resources of a document such as images, scripts, and external pages displayed in frames. The browser accesses the Web through an instance of the Squid Web proxy with an ICAP module. The proxy inserts the CSP report-only headers described in Section 3.3.2 and collects the resulting reports. The proxy also intercepts encrypted SSL traffic.

After crawling a site, I discarded all reports that did not match the site's main domain. These reports referred to external documents loaded in a frame and were not necessary to generate a policy for the main document. (In CSP, a document's policy does not transitively apply to nested documents loaded inside a frame.) From the remaining reports, I generated a policy as in Section 3.3.2.

The crawler should be considered a proof-of-concept to explore the feasibility of automatically generating policies for websites. By following only hypertext links, the crawler cannot detect violations that conditionally occur after load-time, such as clicking the "play" button in a Flash movie, or triggering JavaScript-related events. I leave ways to increase the crawler's coverage to future work.

As a potentially more targeted alternative to automated crawling, I also manually browsed websites in a fresh browser instance and used the proxy to collect reports. This process in-

cluded no feedback. The goal was to cover all areas of the site and trigger as many different violations as possible by specifically exercising functionality implemented in JavaScript or browser plugins.

3.4.2 Evaluation

The question of whether semi-automated policy generation for websites is a suitable approach—without requiring modifications to the sites—depends on two opposing goals. First, the generated policy must not break the site. A policy generation mechanism must discover all resources being included by a site, or a superset thereof. Second, the generated policy should be as narrow as possible in order to provide the maximum safety gain. Unnecessary resources should not be allowed by the policy, and unsafe mechanisms should not be used. In the first part of this evaluation, I compare methods of collecting reports for policy generation on sites where I know that a sound policy exists. In the second part, I explore how well different site architectures work with CSP; that is, whether a sensible policy can be deployed without changing the sites.

3.4.2.1 Crawling and manual browsing of partner sites

From the reports submitted by visitors' Web browsers in Section 3.3.3, I know that stable policies exist for these four sites. Indeed, the sets of policy entries generated by crawling and manual browsing as shown in the upper part of Table 3.7 overlap, and only a few entries were found by only one method. Especially when disregarding differences due to alternative domain names and HTTP(S), both methods performed similarly. However, neither method was perfect. The crawler discovered resources in a rather hidden portion of site B that manual browsing did not uncover. On site D, in turn, manual browsing discovered a resource inclusion that the crawler was not able to find, which was due to exercising JavaScript code when submitting content to the site. The policy entries generated from valid user-submitted reports were always a strict superset of those derived from crawling and browsing (as shown in the lower two-thirds of the table), except for site B where I found that a technical mistake had prevented CSP headers from being sent to users in a small portion of the site. I conclude

Table 3.7: Overlap between the sets of policy entries generated by the crawler, through manual browsing and from user-submitted reports. (In brackets, the number of common/different policy entries when disregarding alternative domain names or HTTP(S).) No method was fully reliable.

Site	A	B	C	D
crawler only	0 (0)	8 (8)	0 (0)	0 (0)
both	3 (3)	12 (9)	12 (10)	8 (7)
manual only	0 (0)	2 (0)	1 (0)	9 (2)
crawler only	0 (0)	9 (9)	0 (0)	0 (0)
both	3 (3)	11 (8)	12 (10)	8 (7)
valid user reports only	0 (0)	3 (1)	26 (3)	14 (2)
manual only	0 (0)	3 (2)	0 (0)	2 (0)
both	3 (3)	11 (7)	13 (10)	15 (9)
valid user reports only	0 (0)	3 (2)	25 (3)	7 (0)

that the crawler and manual browsing techniques need more refinement before they can fully replace user-submitted reports. Since both techniques are complementary, combining them could prove useful to increase coverage.

3.4.2.2 Crawling and manual browsing of CSP-enabled sites

In order to compare the crawler-generated policies to real-world policies, I generated policies for large public websites that deployed CSP in enforcement mode. As a case study, I provide more detail for Facebook and GitHub.

The crawl included the public portion of Facebook as well as authenticated sessions. The policy generated by the crawler was a subset of Facebook's actual policy. It listed the specific subdomains of Content Distribution Networks (CDNs) observed during the crawl, whereas Facebook whitelisted all CDN subdomains with a wildcard. Furthermore, while Facebook's policy restricted only `script-src` and `connect-src`, the crawler also generated entries for `img-src`, for instance. Both issues could cause unobserved (but legitimate) behavior to be blocked and illustrate that automatically generated policies are likely to require fine-tuning using domain knowledge before they can be deployed.

On GitHub, the crawler discovered all whitelisted resources of the original policy (which did not use any wildcards, and restricted only `script-src`, `style-src`, and `object-src`). The crawler generated additional entries that were not part of GitHub's policy. Upon manual

verification, I found that some resources included in GitHub's blog were not loaded due to missing policy entries. This finding illustrates the importance of monitoring enforced policies when websites evolve; regular crawls of a website could be a useful tool to help detect such changes.

3.4.2.3 Influence of design choices on CSP

Architectural features of a site can influence whether it is possible to deploy a meaningful policy without changing the site. The crawls of Twitter, for instance, found a small, stable set of policy entries, while additional manual browsing discovered only one additional policy entry. Most of the resources were internal. Multimedia content included in tweets, for instance, was loaded from internal subdomains with constant names. Such an architecture makes it relatively convenient to deploy CSP without major changes. Indeed, Twitter used CSP in some subdirectories and subdomains.

Other sites such as Amazon, Google, and YouTube dynamically used explicitly named subdomains of CDNs such as `mt{2,3}.google.com`, similarly to Facebook. These subdomains appeared to be used for load balancing and could therefore be considered equivalent from a security point of view. The crawler was not able to enumerate all these subdomains, but post-processing of the policy such as using a wildcard `*.google.com` could address the issue. A drawback of this approach is that sites such as Amazon that use external CDNs would also be whitelisting other customers' subdomains. A cleaner approach would be to use static domain names at the Web application layer and address load balancing transparently at lower layers, as appears to be done by Twitter.

In the examples above, it was possible to compensate for some degree of variability in the sites by broadening the generated policy because the variability was systematic. On certain types of sites such as blogs where users are allowed to include externally hosted content, this may not be possible. The policy used by GitHub shows a possible compromise in such situations: the site allowed images to be loaded from any source and restricted only more sensitive resource types such as scripts and plugins.

3.4.2.4 Stability of policies

A requirement to successfully deploy an enforceable policy is to predict at policy generation time the external resources that will be included when a page is rendered in a browser. A particularly unpredictable type of external content is advertising. The exact advertisement shown to a user is typically determined dynamically while the page is loading. Dynamic advertising can involve techniques such as Real-Time Bidding (RTB), where the opportunity to display an advertisement to a visitor is auctioned off in real-time, and further dynamic activity such as cookie matching between the host website and the winner of the auction. There are routinely tens to hundreds of potential bidders in RTB [68], each of whom represent a large number of actual advertisers.

In order to better understand how this dynamic activity can be reconciled with the more static requirements of CSP, I performed repeated crawls of two large websites with dynamic advertisements and counted how many new policy entries I discovered in each subsequent crawl (Table 3.8). Twitter, which I crawled as a control data point, remained stable and resulted in exactly the same policy in all crawls. On the BBC, the crawler discovered between 13 and 61 new policy entries in each of the follow-up crawls; the vast majority of them were scripts or other content related to advertising. On CNN, the follow-up crawls discovered only between one and four new policy entries, and only one was unambiguously related to advertising. Since both sites displayed comparable types and amounts of advertisements, the differences must be due to the way advertising was implemented. Indeed, the BBC loaded all advertisement-related resources, including RTB scripts, tracking code, and the final image being displayed, directly into the body of the main document. It would be very challenging to deploy CSP in such a scenario because it seems unfeasible to proactively determine any resource that could potentially be loaded. In contrast, CNN isolated advertisements from the main document by loading them as a separate document displayed inside an embedded frame.

This decoupling significantly eases the deployment of CSP because the main document's policy does not transitively apply to the document inside the frame. In such a deployment, it would be possible to enforce a rather strict policy for the main document and a much more

Table 3.8: Additional policy entries discovered in repeated crawls. The high variability due to advertising on the BBC precludes CSP from being used effectively. CNN's way of including advertisement results in a relatively stable (and enforceable) policy.

Crawl number	1	2	3	4	5
BBC	285	+34	+61	+13	+53
CNN	116	+4	+2	+ 1	+1
Twitter	20	+0	+0		

permissive policy for the embedded advertisement document (or none at all). The SOP as well as the HTML5 frame sandboxing mechanism can be used to ensure that untrustworthy scripts in the frame cannot access or modify the main document.

3.4.2.5 Safety of policies

To assess whether policies generated for a site represent any significant reduction in exposure to attacks, I checked whether the policies included "unsafe" CSP features—that is, inline script or style and calls to `eval`. Among the sites I partnered with that included JavaScript, only site B did not require `eval` privileges. Amazon, the BBC, CNN, Facebook, Google, the Huffington Post, and YouTube required all three privileges; Twitter needed inline script and style, and GitHub only inline style. These requirements may be due to code on the sites or in external libraries they include. Even though allowing inline script and `eval` reduces the effectiveness of CSP against XSS attacks, by restricting where external resources may be loaded from, CSP could still make it more difficult for attackers to include custom content such as images or to exfiltrate stolen data.

3.4.3 Conclusions

Neither naïve crawling nor manual browsing alone are sufficient methods to generate a content security policy for a website. With this approach, a certain amount of fine-tuning of generated policies is required for all but the simplest sites. Advanced crawling, or applying machine learning to the generated policies, could reduce the importance of manual tweaks. More complex sites may be able to use only a subset of CSP unless they adjust their architecture. Once a policy has been deployed, an additional challenge is to ensure that it is

always up to date.

3.5 Discussion

I saw that only few websites use CSP, and those that do use it do not leverage its full benefits. For this section, I reached out to security engineers behind larger CSP deployments and summarize key points. Furthermore, I suggest several ways in which CSP adoption could be improved.

3.5.1 Discussions with Security Engineers

To understand implementation decisions behind real-world CSP deployments, I talked to security engineers responsible for three of the measured websites. Out of these sites, two were in the Alexa Top 200, and one in the Top 5,000. The websites used CSP in enforcement mode or report-only for testing. I summarize the key observations in an anonymized fashion.

Websites prefer not to remove inline script. While inline script can be completely removed from websites, this represents significant effort and can lead to more roundtrips when loading the page. Engineers hope to address this issue with the nonce and hash features of CSP draft version 1.1. Hash might be more promising because documents can be distributed over CDNs more easily, whereas for nonce a new document would need to be generated for each response.

Risk of breaking functionality. This was manifested by disabling CSP for browser versions with problematic CSP implementations, including Chrome and Firefox. A website that is secure but not usable can harm business more than occasional XSS. For the future, reliable implementations of CSP in browsers are anticipated.

Enforcement over extensions is considered a bug. CSP rule enforcement can break the functionality of browser extensions. A workaround is to whitelist popular sources. However, extensions could still be unintentionally restricted. A modification of browser implementations or the standard to not enforce rules over extensions could solve this.

3.5.2 Suggested Improvements

I briefly summarize approaches that could help the adoption of CSP and increase its security benefits when deployed.

Ads should be integrated into iframes instead of the main site. Instead of whitelisting all possible ad networks or developing a mechanism for recursive policy adoption, ads should be moved into sandboxed iframes. This allows the main site to be protected with an effective policy, while the iframe can be more permissive, but isolated. Conflating both the site proper and ads in the same context is not necessary, since information required by ads can be passed via `postMessage` cross-window communication. However, while not widely available, alternatives such as Security Style Sheets [66] have been proposed that would allow for such separation without moving content to iframes.

More Web applications and frameworks should adopt CSP. Introducing CSP to programs that are deployed widely can have a higher impact on the overall security of the Web as compared to individual websites adopting CSP. As examples, phpMyAdmin and OwnCloud have adopted CSP, and Django can be configured with CSP. Most desirable would be the introduction of CSP to Web frameworks, which could drastically improve adoption of CSP and the safety of the Web.

Browsers should not enforce CSP on extensions. As discussed in Section 3.3, enforcing policies on browser extensions generates many unexpected reports for websites. Websites should not be forced to whitelist extensions since the number of extensions and third-party resources included by those extensions is theoretically unbounded and cannot be predicted by application developers. Furthermore, CSP in its current form is not an adequate mechanism for websites to block potentially undesired extensions and should not be used as such.

3.6 Chapter Summary

In this work, I have presented the results of a long-term study on CSP as it is deployed on the Web. I have found that CSP adoption significantly lags other Web security mechanisms,

and that even when it has been adopted by a site, it is often deployed in a way that negates its theoretical benefits for preventing content injection and data exfiltration attacks.

In addition, by enabling CSP at four sites, I observed that it is difficult for third parties to deploy CSP, either through incremental deployment using report-only mode or through Web application crawling to semi-automatically generate policies.

CSP clearly holds great promise as a Web security standard, but I can only conclude that it is difficult for most sites to deploy it to its full potential in its current form. It is my hope that the improvements I suggest here, as well as upcoming features of the 1.1 draft, will allow site operators and developers to make effective use of content security policies and result in a safer Web ecosystem.

3.7 Future Work

This work lines out several avenues for continuing research where the CSP standard had shortcomings at time of writing. Since the publication of my work, CSP has evolved and addressed for example dynamically generated scripts with nonce and hash features. Recursively enforcing CSP on included documents is supported by embedded enforcement.[3] CSP Level 2 has been adopted by all major browsers and currently Level 3 is an editor's draft.[4]

However, deploying CSP is still not straight-forward for website developers, and leaves websites vulnerable to attacks which were not considered in the design of CSP. For example, even with strict rules and perfect enforcement, out-of-order inclusion can lead to undesired side effects in JavaScript applications. The idea was outlined in the 2013 essay by Zalewski, "Postcards from the post-XSS world" [9] and shown to be practical by Lekies et al. [51]. These attacks highlight shortcomings in the state of defensive security of the Web, not just CSP per se. Future research should explore avenues towards a Web where building secure websites does not require specialized knowledge, but is an included feature.

[3] `https://w3c.github.io/webappsec-csp/embedded/`

[4] `https://w3c.github.io/webappsec-csp/`

Chapter 4

Identifying History Leaking Browser Extensions

4.1 Introduction

Browsers offer a software interface to access and modify their content: browser extensions. The downside of this powerful interface is that malicious actions at the extension level can lead to problems across all online activities for a user. Extensions can be considered as the "most dangerous code in the browser" [44]. Previous research found extensions to inject or replace ads [18,50,111], causing monetary damage to content creators and, in turn, consumers. To detect privacy-invasive extensions, previous work used dynamic taint analysis to find spyware in Internet Explorer Browser Helper Objects (BHOs) [34]. With previous research in mind, browser vendors can work to restrict malicious extensions.

Google Chrome is considered the state of the art in secure browsing. Chrome extensions can only be installed through a centralized store, and before being admitted they have to pass a review process. Users are prompted for permissions that an extension requests, and can use that information to decide whether they want to install the extension or not. Furthermore, if an extension is considered malicious after admission to the store, it can be remotely removed from clients. With all these security features in mind, privacy in Chrome extensions is still an issue.

This work aims to understand to what extent browser extensions violate user privacy

expectations. In preliminary experiments, I found suspicious activity in popular browser extensions and confirmed that data is not only leaked, but furthermore is processed by third parties. By presenting unique URLs to multiple extensions, I was able to link incoming connections on a honeypot to the particular extension responsible for leaking user data.

Inspired by these findings, I introduce Ex-Ray, a system that can automatically detect history stealing browser extensions without depending on the specific protocol used or leaking methodology. This automated approach is based on analysis of network traffic generated by dynamically exercising unmodified extensions. Extensions under test are executed within an instrumented browser test multiple times, and all network traffic generated during execution is recorded. I decided to focus on the network activity generated by browser extensions because, while their code and logic can change, they ultimately need to send the acquired information to their controller, and this will be observable from network traffic. Thus, this approach builds on a fundamental invariant of tracking and user privacy violation. Furthermore, long term studies of malware have highlighted network activity as a particularly effective medium for detecting malicious activity [53]. I model features that are intrinsic to the network traffic generated by trackers to distinguish malicious from benign traffic. I create complementary detection systems in unsupervised and supervised fashion, and a triage system which can classify the likelihood of a leak, easing the burden on security analysts to identify misbehaving extensions. After identifying a set of extensions that leak private information by looking at their network traffic, I develop a complementary component that can infer if an extension is leaking sensitive information by analyzing the API calls that it makes.

Ex-Ray automatically flagged 212 potential trackers in the top 10,691 extensions with a false positive rate of 0.27%. My system has found two tracking extensions which were not detected by previous systems because they were leaking information using a different channel than was expected by those tools. More precisely, one extension made use of strong encryption to obfuscate its behavior, and the other used WebSockets to exfiltrate user information as opposed to HTTP(S).

In summary, the contributions of this work are as follows:

- I developed the first unsupervised system to detect history stealing browser extensions

based on network traffic alone that is also robust against obfuscation.

- I quantify the magnitude of user data leakage and introduce a scoring system that is used to triage extensions. Prioritized extensions are manually vetted and the resulting labeled dataset is made available to the research community.

- I created a machine learning approach to classify extensions that I use on API call traces generated by an instrumented browser. This approach reaches 96.43% F-Measure value and the Recall value is constantly over 99%.

4.2 Motivation

This work focuses on tracking data collected from browsing behavior that is sent to third parties. As opposed to previous work on history leaking browser extensions [88], I aim for a system that will detect leaks regardless of how they are transmitted or collected. I target tracking either through background scripts or modification to pages. The main difference between these two approaches is that in the Web such trackers are only present on websites that opt-in to use them. From a user's perspective, tools that remove these trackers are available and well understood. Conversely, tracking in extensions can cover all websites a user visits, and there is no opt-in mechanism. Furthermore, no tools are readily available that would warn a user of such behavior or block it.

Transferring the current host or URL can be a necessary part of the functionality of an extension – for example, to check against an online blacklist such as an adult content filter. However, I found that often extensions also transfer URLs if no such checking is necessary, or could be expected by the extension's description, exposing all browsing habits of a user and creating a breach of privacy. Furthermore, the specification of such functionality is often buried deep in an extension's description, if present at all. Web users are concerned about how their privacy is impacted [25, 56], but are often unaware of what a privacy policy is.[1]

To provide additional context behind this sort of systemic privacy-violating behavior on the part of browser extensions, I present a detailed case study on a large actor in the history

[1] http://www.pewresearch.org/fact-tank/2014/12/04/half-of-americans-dont-know-what-a-privacy-policy-is/

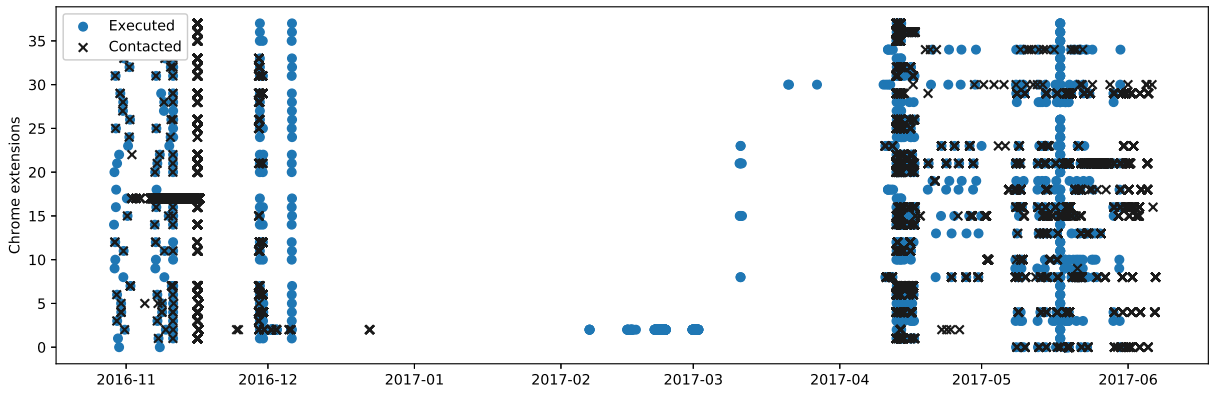

Figure 4.1: Extension execution with unique URLs vs. incoming connections to those URLs from the public Internet. These connections confirm that leaked browsing history is used by the receivers, often immediately upon execution.

data collection market in Section 4.3. In it, I demonstrate how a single library was tied to browsing data exfiltration in 42 extensions with over 8 million installations. The extensions were deleted from the Chrome Web Store within 24 hours of reporting.

4.2.1 HTTP URL Honeypot

To gain insight into the environment in which trackers operate, I configured a honeypot. To test whether leaked URLs are accessed after being received by trackers, I exercised extensions with domain names into which I encode their unique extension ID. While executing in the container, extensions only interact with local Web and DNS servers. However, I operate a web server on the public Internet to monitor client connections for such URLs. As these domains are used uniquely for these experiments, HTTP connections indicate leaks linkable to extensions. The connection and execution times are displayed in Figure 4.1, and discussed in more detail in Section 4.7. The confirmation that trackers are acting on leaked data motivated further steps in this work. After excluding VPN and proxy extensions, I received incoming connections from 38 extensions out of all Chrome extensions with more than 1,000 users.

4.2.2 Types of Trackers

Chrome offers a powerful interface to extensions, and while it can be used for useful tools it can also be misused to violate user privacy. There are multiple ways to collect and exfiltrate browsing history.

Much like trackers that are added to web pages by their authors, extensions can leak history by adding trackers to the body of web pages. An example of third-party tracking is the Facebook "Like" button. These can be blocked by extensions such as Ghostery. A more robust solution is sending collected history data via requests of extension background scripts. Such requests are not subject to interception by other extensions, and cannot be blocked as tracker objects. Compared to tracking via inserting trackers into pages, better coverage can be achieved.

To acquire browsing data, extensions can intercept requests made by websites via the `chrome.webRequest` API, or poll tabs for the URL using `chrome.tabs`. For past browsing behavior, the `chrome.history` API can be used. Diverse options to collect data render finding a unified way to identify tracking extensions challenging.

4.2.3 Threat Model

Based on the honeypot results, I assume the following attacker model. In this scenario the attacker is the owner of, or someone who controls the content of, browser extensions. I assume many users will install these extensions with a cursory reading of the extension's description. While permissions can restrict the behavior of browser extensions, capturing and exfiltrating history can be performed with modest permissions that would not raise suspicion. For instance, the browsing history permission is categorized as *low alert* by Google.

The goal of the attacker is to indiscriminately capture URLs of pages visited by the user while the extension is executed. Furthermore, I assume the adversary collects data with the purpose of analysis or monetization. As the value of traffic patterns decreases over time, I assume the attacker to be inclined to leak sooner rather than later, which seems to be confirmed by the honeypot experiments. A successful attacker would decrease the user's privacy as compared to using a browser without the extension in question.

I exclude from the threat model extensions that openly require the sharing of browsing history as part of their functionality, such as VPNs or online blacklists. Also, I consider leaks purposeful and supposedly accidental as equal, as I cannot reason well about developer intent. As detecting and hiding malicious behavior is an arms race, I prefer to be conservative and assume the attacker could escalate the sophistication of their evasion techniques in the future.

4.3 Case study of a large history data collector

As case study I look into SimilarWeb, one of the actors in data collection in browser extensions. I conducted this study before developing Ex-Ray, as the findings turned out to be symptomatic for a wider range of extensions, the findings motivated this very work [105].

SimilarWeb is a company that offers insights into third-party web analytics. To the end user the functionality is similar to Google Analytics, except that visitors can see traffic details of websites neither they or SimilarWeb are affiliated with. This is useful for analysis of competitors, or to explore new markets for a product.

Using the free version of their service, the presented information includes information on visitors, search, and advertising. The data is detailed, including number of visitors, average visit duration, search keywords used, countries of origin, referring sites, destination sites that the visitors leave through, and others.

4.3.1 Origins of Data

As the company does not have direct access to these data sources, the displayed data must be extrapolated from data which is accessible to them. This high resolution of data without direct access motivated further investigations. Their website suggest use of four types of data sources including *A panel of monitored devices, currently the largest in the industry.*

4.3.2 SimilarWeb Chrome Extension

As first step I analyzed the extension offered on their website. The offered main functionality consists of suggesting sites similar to the one currently seen. After reviewing their code and analyzing network behavior, I noticed suspicious behavior. The extension intercepts requests for all websites and reports any URL or search queries to SimilarWeb in real time, including metadata such as referrers. I noticed that the JavaScript library used for tracking was developed by another company, Upalytics [2]. The purpose of this library is to track user behavior in Chrome extensions, other platforms are advertised on their website as well, including mobile and desktop. Since this was an external library, I suspected it might be used in other extensions as well for similar purposes.

4.3.3 Finding More Extensions

After crawling the Chrome extension store I found 42 suspicious extensions by searching for code similarities. To verify malicious behavior I manually analyzed each extension under the aspect of four questions:

- Does the extension have the capability to exfiltrate private data?
- Does tracking happen "out of the box", or does the user have to opt-in?
- Is this behavior mentioned in the terms of service?
- If not, is there a link in the terms of service that explains the behavior of the extension?

All suspicious extensions were able to collect history, all but one were tracking out of the box. The only extension that offered opt-in was *SpeakIt!*, however, they only switched to that model after a user complained about the included spyware on an issue tracker [3]. Of these 42 extensions 19 explain their data collection practices in the terms of service, while 23 do not. Furthermore, out of these 23 extensions 12 have no URL where this would be explained. One URL that is used across 13 extensions to explain the privacy ramifications is

[2] `http://www.upalytics.com`

[3] `https://github.com/skechboy/SpeakIt/issues/12`

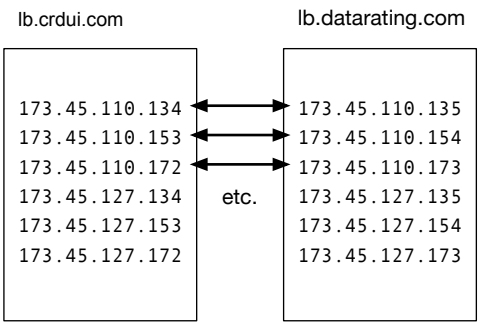

Figure 4.2: Neighboring relationships of IPs between seemingly unrelated domains used for monitoring.

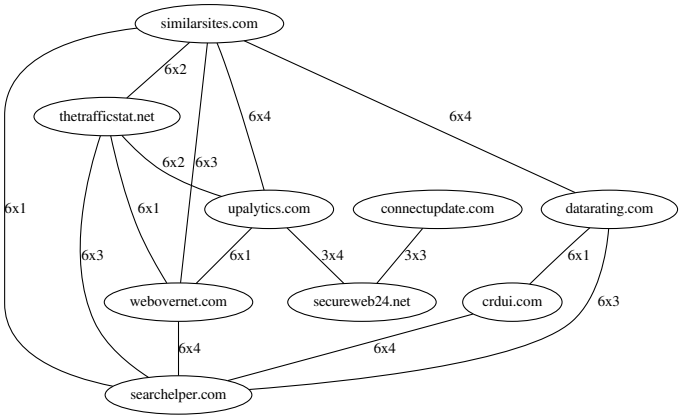

Figure 4.3: Graph linking domain names by IP relationships used in 42 extensions to covertly collect browsing history.

Figure 4.4: Domains using `upalytics.com` library reported to a network of domains that can be linked by IP neighborship.

`http://addons-privacy.com`. The text is a copy of the `upalytics.com` privacy policy rendered into a PNG image. The content explains that personal information including browsing history and IP data will be collected. Throughout the document instead of specific company names only general language such as "our product" or "company" is used. It can be used as a template for any extension using such tracking. While the URL is shared between extensions, the developers have no obvious connection. Six of the remaining domains point to the same IP address. Some versions of the privacy policy reference California Civil Code Section 1798.83 [4], which allows for inquiry about usage of personal information for direct marketing purposes. I sent emails to two of the email addresses, I received responses after less than a month.

4.3.4 Network Information

The extensions used nine different hardcoded hostnames to receive tracking information, I found relations linking all 42 extensions. All endpoint domains, `addons-privacy.com` and `upalytics.com` were registered by Domains by Proxy [5], a service used to obfuscate ownership

[4] `https://epic.org/privacy/profiling/sb27.html`
[5] `https://www.domainsbyproxy.com`

of domain names by hiding WHOIS records. All extensions were reporting to subdomains `http://lb.*`. Some of the names of the domains appear to be misleading, suggesting updates or being a searchhelper. Two of the domains (`connectupdate.com`, `secureweb24.net`) were registered 13 seconds apart. Also, the `robots.txt` file used in all cases is the same.

Furthermore, all these IPs belong to the same hoster, XLHost. Eight out of nine of these hosts have all addresses in a /18 network, half of the IPs of the `upalytics.com` endpoint are in another XLHost network. All IPs in use are unique, however, this involves consecutive IP addresses and other neighborhood relationships.

All hosts used round robin DNS, using multiple IPs for each domain name. To examine this closer I compared the distance of IP addresses used by these extensions for tracking. In Figure 4.3, the nodes are the nine domain names in use, edges are the grade of distance. By taking into account distances of up to four, I can link together all hostnames used in all 42 extensions. For example: IPs `1.1.1.1` and `1.1.1.3` have a distance of two. As for the labels, the edge between `similarsites.com` and `thetrafficstat.net` reads `6x2`. This means that the domains share six IP addresses with a distance of two. Figure 4.2 visualizes the distance relationship between `lb.crdui.com` and `lb.datarating.com`.

4.3.5 Reported Extensions

After reporting the findings, all extensions were removed from the Chrome store within 24 hours, including the official SimilarWeb and SimilarSites extensions - a partner site. I hope that loss of installations from the Chrome store will deter developers from bundling malicious libraries in the future.

4.4 Information Leaks in High-Profile Extensions

In this section I manually analyze extensions which are immune to state of the art privacy detection in extensions.

4.4.1 WOT: Web of Trust, Website Reputation Ratings

Web of Trust (WOT) is a widely used extension with 1.2 M installations. The offered functionality gives users a ranking of trustworthiness of visited websites. WOT came under scrutiny in March 2016 by selling browsing data [6]. A feature that distinguishes WOT from other extensions is usage of strong encryption on extension level. It comes with a cryptographic library (`crypto.js`) that encrypts tracking payloads with RC4 additionally to HTTPS transfer, hiding contents from analysis systems such as data leakage prevention. This extension was automatically flagged by Ex-Ray with a triage score of 61,598 (outstanding, > 1 is considered suspicious) and is undetectable to currently available systems.

The requested permissions allow to access all sites, modify requests, and access tabs. This library will track every visited website, including websites on internal networks. POST data or keystrokes are not monitored.

4.4.2 CouponMate: Coupon Codes & Deals

CouponMate is a shopping application and offers to help searching for applicable coupons. This extension collects and leaks browsing data via WebSockets, a protocol not analyzed by prior work, but rising in popularity. In this dataset I found that 103 (0.96%) of extensions use WebSockets. Ex-Ray is oblivious to protocols and flagged this extension with a triage score of 20.1, which is a high alert for a human analyst.

4.5 Detection Approach

In this section I describe the design of the approach underlying Ex-Ray. To identify privacy-violating extensions, I exercise them in multiple stages, varying the amount of private data supplied to the browser, and in turn to the extension under test. Based on the type of extension, the traffic usage can change depending on the number of visited sites. However, the underlying assumption is that benign extension traffic should not be influenced by the size of the browsing history.

[6] `https://www.heise.de/newsticker/meldung/Abgegriffene-Browserdaten-Mozilla-entfernt-Web-of-Trust-3455990.html` (Link in German)

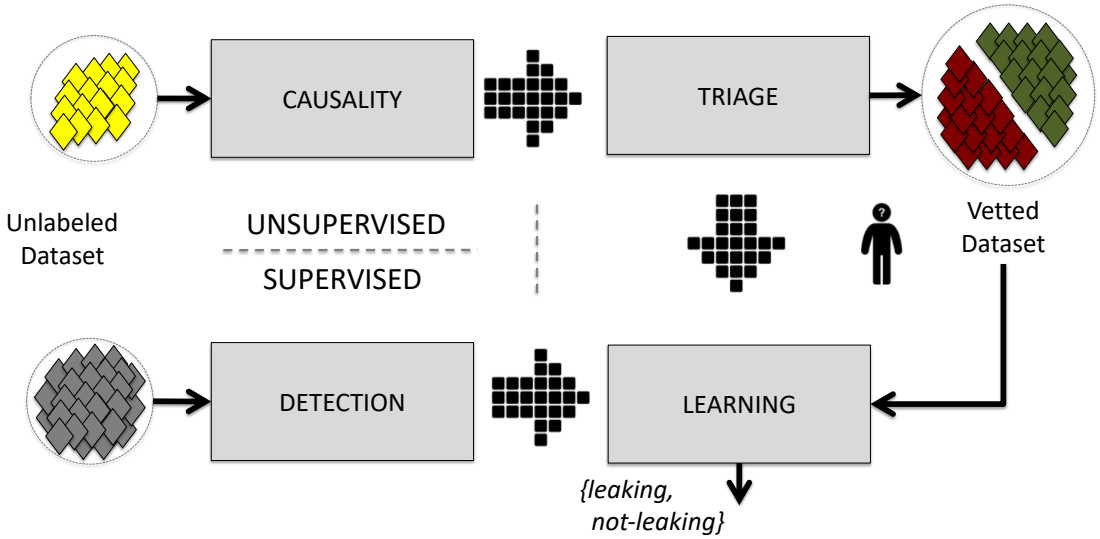

Figure 4.5: Ex-Ray architectural overview. A classification system combines unsupervised and supervised methods. After triaging unsupervised results, a vetted dataset is used to classify extensions based on n-grams of API calls.

4.5.1 Overview

A top level view of Ex-Ray is shown in Figure 4.5. The three main components of the system are summarized as follows:

1) **Unsupervised learning**: I use counterfactual analysis to detect history stealing extensions based on network traffic. This component is fully unsupervised and, by definition, prone to misinterpretations.

2) **Triage-based analysis**: I manually vet the output of my unsupervised system, i.e., I verify which extensions are factually leaking and which are not. As the manual verification is costly, I rely on a scoring system that ranks extensions based on how likely they are to be leaking information to aid the process.

3) **Supervised learning**: I systematize the identification of suspicious extensions using supervised learning over the resulting labeled dataset. This component takes into account behavior of the extension and builds a model that detects history leaks (i.e., it looks at the API calls made by the browser extension when executed).

I see different types of tracking used in browser extensions. Some intercept requests and issue additional requests to trackers. Others transfer aggregate data periodically, while still others insert trackers into every visited page. An integral part of all trackers is transferring data to an external server—simply put, this crucial step is what enables trackers to track.

This work focuses on indiscriminate tracking across all pages. To track, a history item (h_i) generated by the browser will be reported either in isolation or in aggregate. In either case, the size of history items affects network behavior. I argue that network data generated by an effective tracker, independently of protocol and whether plain, encrypted, or otherwise obfuscated has to grow as a function of history.

I execute extensions in multiple stages with increasing amounts of private information. Each h_i should contain less information than the following stage, $h_i < h_{i+1}$. I increase the size of h_i in each stage, extending the length of the testing URLs. For example, `example.com/example/index.html` in stage 0, and `example.com/example/<500characters>/index.html` in stage 10. The expected growth in traffic is h_Δ. This intuition is confirmed from Figures 4.6 and 4.7 where the boxplots clearly show that trackers usually send more data when there is more history to leak while the amount of data is constant across the different stages for benign extensions.

For deterministic tracking, the traffic deltas of adjacent measurements should project an ascending slope. However, the browser history may be sent compressed in order to send as few bytes as possible and avoid the leak being visible as plain text in the payload. This operation would reduce the number of bytes sent while retaining the same amount of information (entropy). Per information theory, message entropy has an upper bound that cannot be exceeded. As consequence, the size of compressed messages has a lower bound as a function of the message entropy. For these experiments, I used compression tools (bzip2, 7zip, xz) to establish a practical lower bound of sent data for each stage as 289 Bytes, 6.9 KB, 14 KB and 30 KB.

Extensions that use trackers establish connections with each execution. Consequently, any group of hosts that results in less measurements than the number of executions will not be considered for further analysis. Examples of hosts that extensions only connect to occasionally are ads.

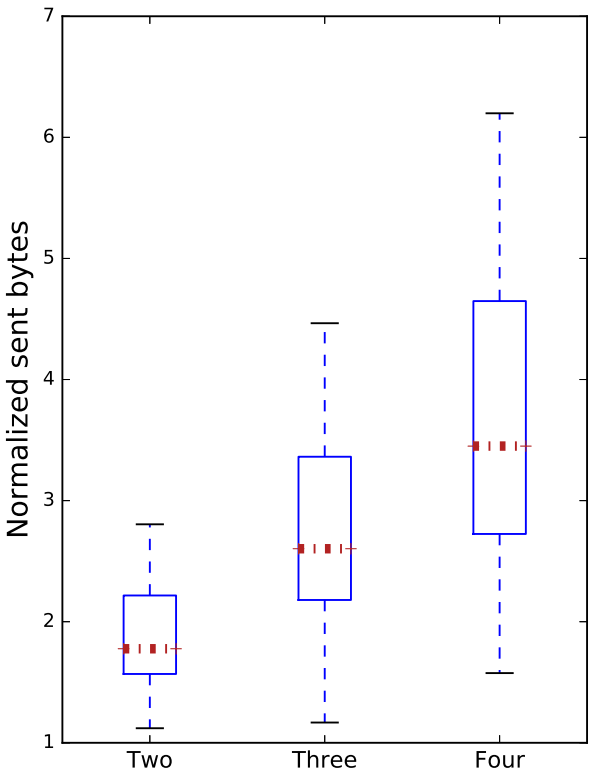

Figure 4.6: Tracking extension.

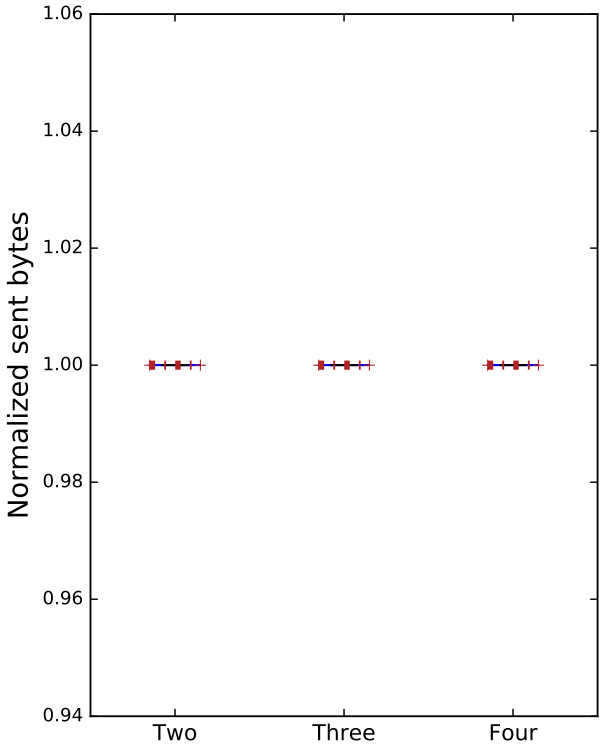

Figure 4.7: Benign extensions.

Figure 4.8: Comparison in change of traffic between executions leaking history and benign extensions. Each bar displays the change of traffic sent relative to executions with increased history. Sent data projects an ascending slope based on size of history. Received data did not reflect this trend.

4.5.2 Network Counterfactual Analysis

The goal of this phase is to model the way in which modifications to the browsing history influence observed network traffic. Figure 4.8 shows that there is a monotonic increase in sent traffic between successive stages of privacy-violating extension. Extensions that, on the contrary, are privacy-respecting show no significant difference. One key finding observed during the analysis of the traffic behavior is that privacy-violating extensions might exhibit non-leaking behavior when connecting to certain domains. Thus, it is important to consider individual flows when building this model. Additionally, I observed that variations exhibited by privacy-violating extensions are well-fit by linear regression.

Thus, I use linear regression on each set of flows to estimate the optimal set of parameters that support the identification of history-leaking extensions. I aim to establish a causality

relation between two variables: (i) the amount of raw data sent through the network, and (ii) the amount of history leaked to a given domain. For this, I rely on the counterfactual analysis model by Lewis [54], where:

> The model establishes that, in a fully controlled environment, if we have tests in which we change only one input variable, and we observe a change in the output, then the variable and the output are linked by a relation of *causality*.

In this case, the *input* variable is the amount of history, the *output* is the number of bytes sent in the different flows, and the *tests* are run with both goodware and malware. My framework allows to evaluate this relationship by means of different statistical tests, such as Bayesian inference. This is ideal for situations were there is no deterministic relationship between the variables, such as in targeted advertisement tracking. Although my framework is designed to model these scenarios, in practice, I observed that leaking extensions behave in a deterministic fashion.

In order to systematically identify the conditions under which the *causality* link is established, I run three steps. The first step is performed before applying linear regression, while the second and third steps are based on the linear regression parameters.

1. **Minimum Intercept**. While the extension might communicate to a domain in all given stages, the content transmitted may not contain a privacy leak. This step verifies whether the amount of data sent exceeds a certain threshold. This threshold is set based on the size of the history compressed as described in Section 4.5.1.

2. **Minimum Slope**. In this work I am primarily interested in extensions that actively track users. This type of extensions is expected to leak as much history data as possible from the user. This implies that the relationship between stages is expected to be linear and have a constant variance, modulo any sort of attempt at obfuscation. Based on this, I set a threshold to the slope in order to exclude all those extensions that do not fully meet these two criteria.

3. **Level of Confidence**. Depending on the extension, the fitted regression model might not always be strictly linear. I can choose to apply certain bounds (lower, upper, or both) from a fitted model to adjust the precision of the output. Choosing bounds

that are very close to the fitted model will give a higher level of confidence in the decision. On the contrary, a very relaxed model will capture boundary cases at the cost of introducing false positives.

I define the term *flagging policy* as the set of parameters used for these checks. A *strict policy* is a policy in which parameters select a restricted area and flag less flows than a *relaxed policy* which flags many more flows as suspected of leaking browsing history.

The notion of confidence described above and the use of the different policies is precisely what motivates the triage system described next.

4.5.3 Extension Triage

After determining which extensions could potentially be leaking history in an unsupervised way, I manually vet those results. The goal of this phase is to design a score that quantifies history leakages and prioritize the manual analysis. For that, I first define a function L that estimates the number of URLs leaked between two controlled experiments such that

$$\mathsf{L}(\mathsf{s_i}, \mathsf{s_j}) = \frac{|\mathsf{s_j}| - |\mathsf{s_i}|}{\tau}. \tag{4.1}$$

Here, $|s_j|$ and $|s_i|$ are the number of bytes sent to a given domain in stages i and j respectively, while τ is a threshold that estimates the expected growth h_δ between i and j. This threshold is based on the size of the URLs used in the experiment described in Section 4.5.1. I abuse notation for the sake of simplicity and denote s when I refer to a transition between two consecutive stages with increasing exercised browsing history.

Given an extension x, I then obtain a score by multiplying the number of URLs leaked between two consecutive stages s:

$$\mathsf{score}(x) = \prod_s e^{L(s)}. \tag{4.2}$$

Note that I aim at obtaining a rough understanding of which extensions could be leaking URLs. Thus, I am mainly interested in high values of $L(s)$. I use an exponential function of

$L(s)$ to prioritize extensions that are leaking in all stages but also to find a trade-off between extensions that only show a leak in some of the stages. Positive and large values will then output a high score.

Negative values of $L(s)$ mean that the total amount of data sent in the earliest stage is higher than the amount sent in the latest. Intuitively this means that data sent in each stage does not depend on the size of URLs in the browser history. The score then treats these cases as an exponential decay function, giving less weight to those stages and outputting values closer to $score(x) \simeq 0$. Likewise, when the amount of data exchanged between stages is exactly the same (i.e., $L(s) \simeq 0$), the scoring function will then output $score(x) \simeq \prod e^0 \simeq 1$. One could obtain a probability of the likelihood of a leak by scaling the score to the interval $[0, 1]$. However, as my system currently aims only at prioritizing extensions, giving a rough notion of risk (without scaling the output) suffices. Thus, I consider the following thresholds as a general rule of thumb when triaging extensions:

$$leak(x) = \begin{cases} \texttt{not-leaking} & \text{if } score(x) \leq 1 \\ \texttt{possibly-leaking} & \text{if } 1 < score(x) \leq 100 \\ \texttt{likely-leaking} & \text{otherwise,} \end{cases} \tag{4.3}$$

where very large values of $score(x)$ show a high confidence that the extension is aggressively tracking users.

It is important to highlight that the triage stage is optional. An ideal setting with endless human resources would obviate the need to prioritize extensions and, thus, render the triage stage merely informative. In practice, extension markets are very large and human workers tend to be constrained time-wise, which can be a bottleneck for the verification process. In this case, having a triage system would be valuable. For the purposes of this work, when labeling the outputs given by the unsupervised module, I have invested human effort in manually vetting extensions that are primarily ranked as `likely-leaking` (for positive samples) and `not-leaking` (for negative samples). I also look at a fraction of `possibly-leaking` extensions in a best-effort fashion.

4.5.4 History Leakage Detection

The last component of this system aims at systematizing the identification of unwanted extensions from a behavioral point of view. For that, I instrument the browser to monitor dynamically relevant behaviors of the extension during runtime. This component operates in a fully supervised manner and is composed of the following two phases:

- **Learning**: The system is trained using the dataset labeled in the previous phase, building a model of the most discriminatory runtime behaviors.
- **Detection**: The system is deployed to detect previously unknown privacy-violating extensions.

Predictions can be then used (i) to obtain a better understanding of how extensions (mis)use the user's private information; and, (ii) to discover previously unknown privacy-violating extensions than can then be analyzed by the triage component. As a result, the list of labeled samples together with the model can be extended.

I implemented this classification algorithm using *Extra Randomized Trees*. I choose this classifier due to its efficiency over several types of classification problems [35]. However, my framework accepts a wide range of classifiers. Likewise, the system can learn from several types of features. For the purposes of this work, I limit my analysis to behaviors related to history leaking. In particular, I profile the way in which extensions interact with certain components of the Application Programming Interface (API) exported to extensions by Chrome.

From all API traces that are extracted, I model the way in which consecutive calls are invoked using n-grams. This detection method has been widely explored for the identification of malicious software. As explained in prior work [94, 110], n-grams are particularly useful to model sequences of elements. The number associated to the "n" is the length of each examined sequence; the system receives labeled sequences and uses them to train a classifier in order to recognize from the sequences of an unknown sample to which label the sample should be assigned. In this system I tried different depths of the n-grams, namely 1, 2, and 6, and the sequences are the sequences of consecutive API calls invoked by the extension.

4.6 Ex-Ray Implementation

In this section I describe technical aspects of the experiments, how I exercise extensions and collect data. The overview of the experimental setup is depicted in Figure 4.10.

4.6.1 Extension Containers

As part of the test environment, I created websites that allow scaling the size of a web client's browsing history without otherwise changing the behavior of the websites. I used local web and DNS servers so that the browser could connect to the website without sending information to the public Internet. For each execution, I started the experiment from an empty cache in a Docker container using an instrumented Chromium binary. I exercised each extension four times for five minutes each, capturing all generated network traffic. Capturing traffic on the container level provides a full picture of each extension's network interactions.

To reduce measurement noise, I blocked traffic to Google update services and CRLsets[7] via DNS configuration. I also disabled browser features such as SafeBrowsing and account synchronization.

Considering the maximum URL length of 2,083 characters, I increased the length of URLs by 500 characters between stages. Other than changing the length of URLs used, the pages served to the instrumented browsers did not change between stages. The maximum length of URLs generated by us is below 1,600, leaving sufficient space for trackers that submit URLs as GET parameters. For each execution I open 20 pages; thus, if all URLs were transmitted uncompressed I would expect an increase of 10 KB per stage. I stored DNS information to group IP traffic by hostname.

4.6.2 Browser Instrumentation

To collect detailed data of extension behavior I implemented a Clang LibTooling program to rewrite Chromium source code. I instrumented the source folders `extensions`, `chrome/browser/extensions`, and `chrome/browser/extensions`. Excluding files such as unit tests, I

[7] gvt1.com, redirector.gvt1.com, clients1.google.com, clients4.google.com

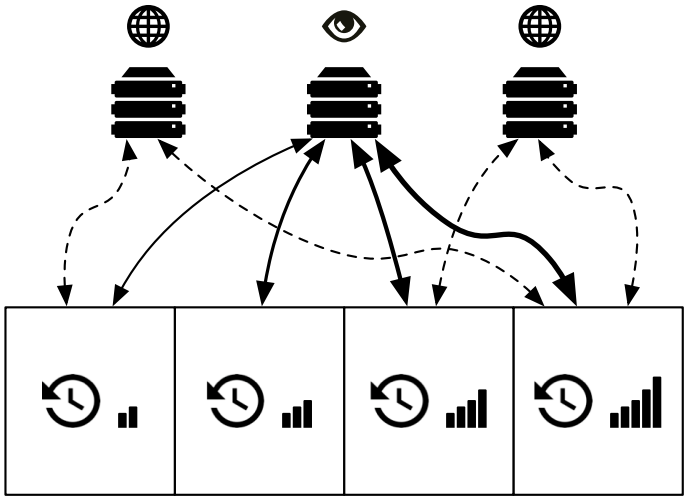

Figure 4.9: Extensions interact with multiple servers on the Internet, sending and receiving data. Trackers that receive browsing history behave differently than other servers. By varying browsing history over repeated executions, patterns of trackers become apparent.

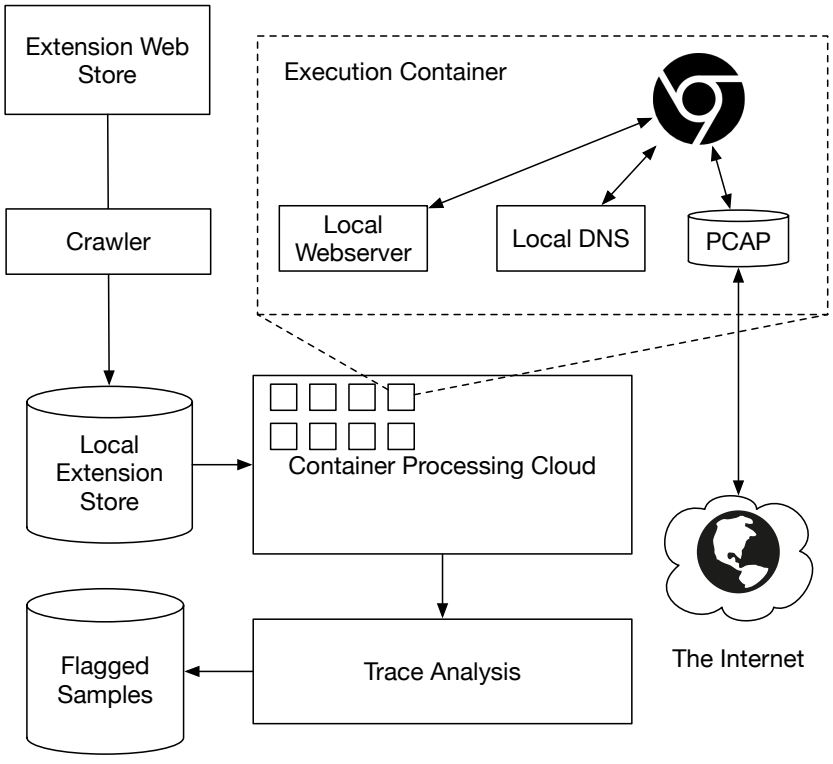

Figure 4.10: Ex-Ray extension execution overview.

Source Folder	Files	Functions	Parameters
chrome/browser/extensions	385	4,967	2,548
chrome/common/extensions	34	218	176
extensions	504	5,947	3,401

Table 4.1: Instrumented files, functions, and collected parameters after running the LibTooling program on the Chromium source code.

inserted 11,132 trace points in 923 files collecting 6,125 function parameters. I collect all primitive parameters or objects that I can convert into strings.

4.7 Evaluation

In this section, I describe my evaluation of Ex-Ray: the experimental methodology, results, and discussion of findings.

4.7.1 Experimental Setting

An overview of the experimental setup is depicted in Figure 4.10.

4.7.1.1 Extension Dataset

I crawled the Chrome Web Store and downloaded extensions with 1,000 or more installations. For this analysis, I only consider extensions that can be loaded without crashing. Examples of extensions that could not be loaded are those with manifest files that cannot be parsed or referencing files that are missing from the extension packages. This left 10,691 extensions for Ex-Ray to analyze.

To establish baseline ground truth, I searched for different types of tracking extensions. I did not use the extensions mentioned in Appendix 4.3 as they were not available in the store at the time of these experiments, and the future behavior of tracker endpoints was unclear.

I mainly relied on two approaches to discover extensions:

- **Heuristic search**. I looked for suspicious hostnames, keywords in network traffic, and applied heuristics to traffic patterns. Through manual verification I confirmed 100

benign extensions and 53 privacy-violating extensions. The dataset contains different types of samples, including aggregate data collection and delivery over HTTP(S) and HTTP2.

- **Honeypot probe**. I registered extensions interacting with the honeypot and verified 38 as connecting back from the public Internet. Figure 4.1 shows a map of all incoming connections with respect to the time I exercised the extension with unique URLs in the history. Table 4.2 shows the most installed five malicious extensions with domains connecting to the honeypot. Connections often appear immediately after running the extension, but I detected deferred crawls as well.

I excluded VPN and proxy extensions that redirect traffic through a remote address as these are not part of the threat model. The connecting clients performed no malicious activities I could identify in the log files. The hostnames of clients that connected to us varied widely. The most popular one was `kontera.com` with 704 connections, followed by AWS endpoints. Surprisingly, I received many requests from home broadband connections, such as `*.netbynet.ru`, often connecting only once. However, I connected four graphs of extensions that were contacted from the same hosts. The biggest graph connected eight extensions with two hosts. The other graphs connected five, two, and two extensions.

12 of these extensions were removed from the Chrome Web Store before the experiments concluded.

4.7.2 Ex-Ray Results

4.7.2.1 Tuning of the Unsupervised Component

The first step of Ex-Ray consists of applying linear regression for counterfactual analysis. The linear regression test flags flows if they respect the three parameters explained in Section 4.5.2. To find the best configuration of these parameters, it is necessary to evaluate the results on a labeled dataset. I used F-Measure as a comparison metric. The strictest policy checks for a minimum of five URLs leaked, a 2% minimum slope, and 90% accuracy. This policy results in an F-Measure of 96.9% and no false positives.

Extension Name	installations	Connecting from
Stylish - Custom themes	1,671,326	∗.bb.netbynet.ru, ∗.moscow.rt.ru, ∗.spb.ertelecom.ru
Pop Up Blocker for Chrome	1,151,178	∗.aws.kontera.com, 176.15.177.229, ∗.bb.netbynet.ru
Desprotetor de Links	251,016	∗.aws.kontera.com, ∗.moscow.rt.ru, ∗.bb.netbynet.ru
Открытые вкладки (Open Tabs)	97,204	∗.dnepro.net, 109.166.71.185, ∗.k-telecom.org
Similar Sites	45,053	∗.aws.kontera.com, ∗.moscow.rt.ru, ∗.netbynet.ru

Table 4.2: Top five extensions connecting to the honeypot with highest installation numbers which are still available in the Chrome Web Store.

To obtain better results, in the final configuration I used two less strict configurations and flagged as suspicious all flows flagged in both engines. Both configurations check for a minimum of two URLs leaked and 2% minimum slope. However, there is a difference in the last check: while one used 90% accuracy in checking only the lower bound, the other one used 80% accuracy checking both the upper and lower bound. As such, the first and the last checks are less strict, but the F-Measure did not decrease even if a larger area of the feature space can be flagged. The system correctly flagged more flows as with the stricter configuration, but the flows belonged to the same extensions already flagged by the previous system.

4.7.2.2 Labeling Performance

Ex-Ray flagged 212 extensions out of 10,691 as history leaking using the linear regression on the traffic sent by the extensions. By checking manually, I noticed that not all the extensions flagged were history leaking. Out of 212 samples, 184 were leaking, two were goodware, and 26 were unclear. It has not been possible to determine if among those 26 extensions there were ones leaking or not. Therefore, to provide a conservative evaluation, I consider Ex-Ray to have 28 benign extensions wrongly identified as history leaking.

As mentioned earlier, detection systems can be prone to false negatives. To measure this for Ex-Ray, I spot-checked a representative sample of extensions reported as benign.

To establish baseline false negatives I scanned the pcap files for leaks and reimplemented another system used for brute-force searching extension traffic for obfuscated strings with a fixed set of algorithms [88]. This system flagged 367 extensions which I used for the dataset. The false negative samples I subjected to examination numbered 178. These results lead to a precision of 87%, a recall value equal to 50.13%, and an F1-Measure value equal to 63.66%. The overall accuracy value is 98.03%. These values are reached using only the first step of Ex-Ray that is a completely unsupervised algorithm. As I show below, these results are further improved by the next phases of the system.

Among the extensions flagged by Ex-Ray, there are some noteworthy case studies (such as the Web of Trust extension) which are discussed thoroughly in Section 4.4.

4.7.2.3 Prioritizing Extensions

Extensions processed in the previous step are then ranked using the score function defined in Section 4.5.3. Ideally, a triage system should prioritize extensions that are more likely to be privacy-violating than others. This way, the analyst can invest most of his efforts on specimens that are likely to be worth exploring. Together with a ranking of the extensions, Ex-Ray provide a report with an informative breakdown of the contribution of each network flow to the overall score. This is also useful to the analyst for further manual investigation. I next show snippets of a triage report for three extensions that are ranked high, medium, and low:

```
QR Code Generator (cicimfkkbejhggfjaabggafffgdnjgjp)
    4e+18  connectionstrenth.com
   394.88  a.pnamic.com
    28.22  eluxer.net
     4.48  rules.similardeals.net
     1.16  code.jquery.com

Kizi - Free Fun Games (pmmbokildidpgafchfmebmhpoeiganhj)
    89.22   static-opt1.kizi.com
    89.22   cdn-opt0.kizi.com
    89.29   cdn-opt1.kizi.com
     6.12   tpc.googlesyndication.com
     3.15   securepubads.g.doubleclick.net

Bible Quote of the Day (pogchimbndbckepmhaagnapfmlfgnala)
     1.00   www.gstatic.com
     1.00   chromium-i18n.appspot.com
     1.00   ssl.gstatic.com
     1.00   localhost
     0.67   www.google.com
```

The snippet first displays the name and unique identifier of the extension, followed by the score given to each of the network flows used to allegedly leak the history. As mentioned before, I group network flows by hostname using DNS information captured during the execution. When considering the thresholds introduced in Equation 4.3, the recommendation given by the triage system for these three extensions is `likely-leaking`, `possibly-leaking`, and `not-leaking` respectively.

As mentioned, these recommendations are manually verified. The analysis starts with review of the source code, looking for access to `chrome.tabs`, `chrome.webRequest` interception, and other methods of history access. Next, an analyst checks for elements inserted into the DOM that leak the referrer. Finally, requests generated by the background scripts and other recorded network traffic is checked.

To quantify the performance of the triage system, I first study how the triage system ranks extensions given by the unsupervised system with respect to the baseline ground truth described above (see Section 4.6). 73 extensions are flagged as `likely-leaking`. Out of those, all but one were manually verified to leak (99%).

Next, the analyst is tasked with verifying 121 additional extensions from the `possibly-leaking` category. Out of those, only one is confirmed to be benign, 24 are marked as "unclear," and the rest (80%) are confirmed to leak. When ordering the triage score from the bottom, it is easy to find extensions that behave legitimately. For the purpose of this work, the analyst vetted approximately 100 extensions as `not-leaking`.

I emphasize that the purpose of this phase is not to exhaustively label all leaking extensions, nor to obtain a comprehensive understanding of non-leaking extensions. Instead, the aim is to obtain a slice of those extensions where the quality of the ground truth is enough to apply supervised learning. As a byproduct of this manual verification, I have gained a number of insights into the ecosystem of unwanted extensions which is discussed later in the Section 3.5.

4.7.2.4 Classification Results

The last experiment evaluates the effectiveness of the supervised system introduced in Section 4.5.4. I aim at understanding the performance of Ex-Ray in classifying leaking extensions using API call traces.

I rely on the dataset labeled and vetted in previous stages of this experimentation. I split the dataset between training and testing set using a k-fold cross-validation approach, which has been widely applied in the past [74].

To collect behavioral data from extensions, I implemented a Clang LibTooling program that instruments Chromium's source code. In particular, I instrumented the following components of the Chromium framework: *extensions*, *chrome/browser/extensions*, and *chrome/browser/extensions*.[8]

To evaluate the results, I refer to precision (or positive predictive value) and recall (or sensitivity). With reference to detecting leaking extensions, I judge the performance by

[8]Excluding unit tests files, I inserted 11,132 trace points in 923 files collecting 6,125 function parameters.

Type	Prec.	Recall	ACC	F1
n-gram=1	87.36%	98.19%	87.30%	92.46%
n-gram=2	93.56%	99.49%	94.14%	**96.43%**
n-gram=6	92.18%	99.23%	92.70%	95.58%

Table 4.3: n-gram classification results for varying n.

the F1-score, as it represents the harmonic mean of precision and recall. For the sake of completeness I also report the proportion of correct predictions (accuracy).

Table 4.3 shows results over a 5-fold split using random sampling. Classification results indicate that I can accurately identify when extensions are leaking by examining at their behavior. After evaluating different sizes of n-grams, I obtained the best results with $n = 2$ at 96.43% F1 followed by $n = 6$. When looking at the histogram of APIs executed by the extension ($n = 1$), the performance drops about 7%.

Among the most informative features in the best setting, I can observe calls to different API packages related to the manipulation of URLs such as *extensions.common.url_pattern*, as well as the manipulation of runtime code (JavaScript) associated with the preferences of an extension. In particular, the following two API calls are predominantly seen together in leaking extensions: *extensions.browser.extension_prefs.GetExtensionPref()* $\rightarrow$ *chrome.browser.extensions.shared_user_script_master.GetScriptsMetadata()*.

4.7.2.5 Comparison to DNS Blacklist Approaches

While tracking in extensions is bypassing adblockers in browsers, other approaches are possible. Blocking of ads and trackers via DNS is an option, examples are AdAway[9] or PI Hole[10]. To use such blockers client devices (desktop computers or mobile devices) are configured to use such a DNS server. Queries to hosts which are considered ads or trackers are resolved to `127.0.0.1`, otherwise the name will be resolved correctly. Such DNS-based blocking has advantages such as not requiring installation of client-side software or preventing tracking on a lower level. Disadvantages are that domains might be used for multiple purposes, and if flagged as tracker the domain becomes inaccessible for benign requests too.

[9] `https://adaway.org/`
[10] `https://pi-hole.net`

I compare results of the unsupervised component to such DNS blacklists. 212 extensions were flagged by the unsupervised component, out of which 184 were manually confirmed to leak. These extensions leaked browsing history to 209 different hosts. Furthermore 35 hosts were wrongly flagged as history-leaking.

Out of the 209 flagged hosts only 28 were blocked via DNS blocking. And out of the mislabeled 35 hosts 12 matched the blocked list.

The takeaways are two-fold:

1. Few of the verified leaking hosts (13%) can be flagged via DNS blocking. However, such blocking could still reduce the impact of history leaking browser extensions.

2. A third of the mislabeled hosts (34%) were identified as ads or trackers. This suggests that although these hosts were not in scope for the used threat model, the component identified potentially suspicious activity.

To summarize, DNS blocking would not be sufficient to prevent history leaking through browser extensions, but could reduce the impact. Due to ease of configuration blocking ads and trackers via DNS is a promising blocking technology.

4.8 Discussion

In this section, I describe and discuss a number of findings resulting from this work. I present the most prevalent types of trackers and discuss their fundamental differences together with the issues due to invasive tracking. Finally, I discuss evasion strategies.

4.8.1 Browser-enabled Tracking

Trackers are popular on websites and well-studied. However, they are fundamentally different from tracking in browser extensions. Websites need to opt-in to use a tracker, and their scope is limited to their own website unless purposefully shared. Furthermore, visitors can use tracker-blockers to opt-out of tracking with extensions such as Ghostery. Conversely, in browser extensions the scope of tracking is not limited to a single website, but collects

information on all websites the extension has permission to access. Furthermore, no tools exist to reduce the impact of privacy on the user.

Mozilla Firefox Using a prototype I developed for Firefox extensions, I scanned the most popular available ones in the store. I found five extensions with over 400,000 total installations which were tracking user behavior outside of extensions, and reported them to Mozilla. Out of these, three were removed from the store because they did not disclose tracking in their privacy statement. However, this type of tracking is generally tolerated for Firefox, and as a result I have not further pursued notifications on that platform.

4.8.2 Foundations Towards Solutions

A combination of these suggested solutions would palliate the problem of invasive tracking in browser extensions.

- Analyze extensions submitted to extension stores with tools that check for tracking behavior, such as the proposed Ex-Ray system. Users can then be warned that their browsing history will potentially be leaked.
- Implement a new browser extension API to inspect and block traffic to trackers generated by other extensions in background scripts. No such API currently exists. Filtering approaches have proven effective for tracking on websites, and Ex-Ray could be integrated into this model and extended to filter background traffic.
- Consider invasive tracking as a violation of the single purpose rule in extension stores, analogously to ad injection.

4.8.3 Evasion

Malware evasion is a well-explored area and is part of the arms race between attackers and defenders. Examples of this include fingerprinting analysis environments or creating more stealthy programs. While no ultimate solutions exist for these problems, Ex-Ray addresses tracking at a fundamental level.

Another approach would be to lay dormant and only leak at a later point in time. However, I have seen with the honeypot experiments that if leaks are utilized, this often happens immediately. Furthermore, there is an economic incentive on the part of attackers to obtain and monetize leaked history as quickly as possible before its value begins to degrade.

Extensions could also pad sent history to show stable traffic behavior or create noise. However, this would either limit the leakage capacity or be easy to detect from simple checks applied by current defense systems if extensions regularly send large amounts of data to mask leakage.

4.9 Future Work

Ex-Ray's goal is to flag extensions that collect private data such as browsing history and exfiltrate it to third parties. An actor with the goal to collect user data is interested in collecting data in real time, which is supported by the samples I analyzed.

It is possible that extensions only exfiltrate data after waiting for a period longer than my tests. However, this is at odds with economic incentives due to the decreasing value of stolen data over time, and is thus unlikely from the perspective of the malicious actor.

Extensions that are narrow in scope, e.g., that collect data for a specific website, would not be flagged by my system. I consider stealing of private information on a wider scale. To enhance this system, an approach similar to honeypages in Hulk [50] could be used.

Browsing data could be aggregated by extensions and only transmitted at a later point. However, tracking for the purpose of analysis of large-scale user behavior requires timely data on all websites. The scope of this work is identifying wholesale tracking through extensions.

Malicious software that only triggers on narrow conditions can be impossible to exercise. For example, authors could assemble code based on environmental parameters unknown to analysts. A famous example is the Gauss malware.[11] This malware will only trigger on computers that have a specific configuration and is otherwise not decryptable. Global efforts to analyze this malware have failed to date.

[11] http://arstechnica.com/security/2013/03/the-worlds-most-mysterious-potentially-destructive-malware-is-not-stuxnet/

Knowing the specifics of my tool, malicious developers could apply evasion techniques, for example transferring a constant amount of data per visited website by padding URLs or captured keystrokes. Evasion is a general concern for any detection system and there exist several avenues to address this [55].

4.10 Chapter Summary

With this work I introduce new methods of detecting privacy-violating browser extensions independently of their protocol. I use a combination of supervised and unsupervised methods to find features characteristic to tracking in extensions. I implement Ex-Ray, a prototype implementation of my approach for the Chrome browser, and find two extensions in the official Chrome Web Store which leak private information in previously undetectable ways. Privacy leaks in browser extensions are in an arms race, with extensions evading known methods of detection of previous work. I suggest that extensions should be both tested more rigorously when admitted to the store, as well as monitored while they execute within browsers.

Chapter 5

SPA Rewriting to Enhance Vulnerability Discovery

5.1 Introduction

Developers introduce programming mistakes through oversight, which can lead to exploitable vulnerabilities. Integration of bug-finding tools into the build process is considered best-practice. One example of such techniques is fuzzing, which is part of the Microsoft Security Development Lifecycle[1]. Such penetration testing tools play a critical role in finding vulnerabilities in programs. Tools as these can help identify vulnerabilities before an application is deployed and can be exploited.

Unfortunately, most of these Web pentesting tools consider Web applications only as server-side programs, following a paradigm of request and response transitions. As more code is deployed on the client-side, introspection of these programs presents a roadblock for finding vulnerabilities. Transitions in client-side program states are not represented through a request/response pair, but interactions with the application that are purely client-side. As applications can be implemented in a variety of different frameworks, singular interaction strategies are difficult. Exploration of JavaScript programs is a feature that is not widely supported by penetration testing tools.

[1] `https://www.microsoft.com/en-us/sdl`

This chapter consists of two mains parts. First, I measure how well adept penetration testing tools are to modern Single Page Applications (SPA). Second, based on the state of the art and shortcomings of these tools in regards of SPAs, I introduce a prototype, SPARE, that rewrites SPAs to the request/response paradigm.

The results of this study demonstrate that the current offerings of black-box vulnerability scanners lack the capabilities vital to keep up with progressive technologies. The shift of program code from the server to the client has not been followed by these penetration testing tools. The scanners need to overcome a number of challenges when testing modern web applications, and even though some tools have begun undertaking this task, even these are lacking. Of the analyzed scanners only three could execute JavaScript: Acunetix, IBM Security AppScan, and Google Cloud Security Scanner. However, limitations in JavaScript capabilities prevents penetration testing tools from discovering vulnerabilities in SPAs.

Although a tool may be sophisticated, shortcomings in one area can lead to failure as a whole. Acunetix was fairly sophisticated, yet it is unable to handle Backbone's event binding. These tools must develop an ability to better interface with client-side applications to explore them correctly to uncover security flaws.

The SPARE prototype can push client-side code to be rendered on the server, shifting the paradigm. With this approach SPAs can be lead into the traditional request/response paradigm and enhance access scanners have to applications.

The prototype moves the program state as it would be represented in the client-side application back to a server-side DOM, and offers interactions in a format that traditional penetration testing tools can interact with. These tools are often closed source an cannot be modified, therefore rewriting SPAs is necessary to become compatible. It enables us to leverage the benefits of existing scanners without modifying them.

The contributions of this work are as follows:

- I evaluate 13 state-of-the-art black-box web application vulnerability scanners against three client-side web application frameworks.
- By analyzing the evaluation results, I identify a number of challenges that Web application vulnerability scanners must overcome in order to properly crawl and fuzz

client-side web applications.

- I show, by the results of this analysis, that client-side Web application are fundamentally different from server-side web applications, and, therefore, require novel techniques to perform automated vulnerability analysis.

- I introduce a prototype, SPARE, that overcomes some of the limitations imposed on Web penetration testing tools, by rewriting SPAs into the request/response paradigm.

5.2 Motivation

This work aims to both evaluate the state of the art of client-side web penetration testing scanners with regards of SPAs, and advance it. Black-box penetration testing tools are often used to find vulnerabilities in web applications before they are widely deployed. These point-and-click solutions provide an important baseline for vulnerability discovery before manual search for vulnerabilities.

These scanners take multiple steps to identify a vulnerability, crawling is the first one. Deficiencies in crawling will prevent detection of vulnerabilities in later steps. Prior work has evaluated server-side black-box scanners [31], concluding that vulnerabilities are often overlooked. However, the request / response paradigm is well-understood for such tools, and yet it is challenging for these tools. How interacting with SPAs compared to traditional web application scanning was unclear. SPAs provide a paradigm shift in interaction, as JavaScript became an essential part of web applications. As no comparable work on SPAs exists, it is a pressing topic to explore.

5.2.1 Threat Model

For the scope of this chapter I assume that both the browser and underlying operating system are trusted and free of vulnerabilities. As the client-side web application is executed in the browser, malware or rootkits installed on the same computer could compromise applications running in the browser, or any data it operates on. The target of black-box

vulnerability scanners is to find vulnerabilities in the application itself, rather than in the underlying layers.

For an attacker the threat model of SPAs is comparable to those of server-side web applications. Examples of vulnerabilities that a black-box vulnerability scanner finds are when an attacker can manipulate input to the client-side web application to perform operations on behalf of a legitimate user. Such attacks would compromise security aspects as either the confidentiality, integrity, or availability of the application.

An example of a client-side vulnerability is DOM-based XSS where an attacker is able to execute data as code, leading to execution of untrusted code on behalf of the program. Consequences of DOM-based XSS are similar to XSS in a traditional server-side web application.

5.3 System Overview

This section is divided into tools developed for analysis of effectiveness of black-box testing tools, and rewriting SPAs to become easier to scan by such tools. To assess functionality I used a capabilities test with JavaScript code instrumentation, and a tool to evaluate the generated traces. For rewriting SPAs to become compatible with scanners, I created a NodeJS application that uses a server-side DOM to keep state, acting as a proxy to pre-render JavaScript execution.

5.3.1 JavaScript Instrumentation

The instrumentation is built on top of Google Closure Compiler [6]. Instrumentation is applied on a function level, collecting traces on function invocation and exit. The process is based on code used for the ZigZag [108] project. A patch that allows configuration of instrumentation externally was accepted upstream.

With each function invocation execution trace data is collected. The collected data includes the filename, unique function id, function name (as string), invocation type (entry or exit), URL of invocation (href), and arguments used. For each collection event a POST

```
1    function x(a, b) {
2        // function body
3        ...
4        return a+b;
5    }
```

Figure 5.1: Function before performing instrumentation.

```
1    function x(a, b) {
2        __calltrace(functionid);
3        // function body
4        ...
5        return __exittrace(functionid, a+b);
6    }
```

Figure 5.2: Function after instrumentation

request to the data collection server is sent. Examples of a function before and after instrumentation are displayed in Figures 5.1 and 5.2, trace collection code is shown in Figure 5.3.

5.3.2 Vulnerability Scanner Capability Test and Log Analysis

The test application contains challenges for crawling and finding of exploits. While the application is executed, the instrumented code is collecting trace data. The log analysis component consumes traces to interpret the capability test by the log analysis component. The result of this test is an assessment of how well a penetration testing tool can explore the application. Relevant metrics are code coverage but also callgraphs can be reconstructed by matching invocation ids.

Shortcomings of scanner capabilities can be used as motivation for rewriting SPAs to become compatible. E.g., for a scanner that does not support event binding, an SPA can be rewritten to display `a` elements instead.

Various crawling challenges were used to test capabilities. Some of these difficulties reflect variations in the designs of the implemented frameworks. It should be noted that each of these challenges requires the crawling module to execute JavaScript, as this is a fundamental task in order to evaluate client-side web applications. However, the challenges test to what extent the scanners are able to interact with these JavaScript applications.

```
1    function __calltrace(functionid) {
2      var a = {};
3      a["seq"] = ++globalseq;
4      a["filename"] = "app.js";
5      a["fun_id"] = functionid;
6      a["type"] = "entry";
7      a["function_name"] = get_function_name(arguments.callee.caller);
8      a["href"] = document.location.href;
9      a["args"] = _flatten_args(arguments.callee.caller.arguments);
10     a["unique_id"] = uniqueid;
11     $.ajax({method:"POST", url:"http://datadumper.example.com",
12             data:JSON.stringify(a)});
13   }
```

Figure 5.3: Data-collection function which is invoked on each function entry. The call sends all relevant tracing information to a trace collection server. The exit-function operates similarly, except that it is invoked with the return value, which is passed back.

Furthermore, I consider CAPTCHAs [100] as out of scope the analysis. CAPTCHAs are a technique used to prevent unauthorized, automated access to websites. In the use-case of performing an authorized pentest on website they can be disabled.

Event binding: In traditional web applications, a button or link that corresponded to an action would load a new page from the server based on the associated `href` or `action` attribute. In one of the biggest changes brought about by client-side applications, rather than generate a new page based on an action, event binding enables the application to run JavaScript code in response to the action without the need to change the page. This difference presents a difficulty to the traditional crawling tactic of parsing a page for `href`s and `action`s and then requesting the resulting pages. Now an application can run code and optionally change pages through JavaScript instead, as done throughout the test application.

Data binding: Another distinction between client-side web applications and traditional web applications is that client-side applications dynamically modify the web page using JavaScript. While traditional applications reload the entire page sent from the server, client-side apps can run code in the web browser to modify portions of the page without reloading it entirely. The crawling module must be able to register changes that occur on the same page

and subsequently handle the new elements that were introduced.

Delayed redirect: A delayed redirect is used for one of the pages in the test app. When the corresponding button is clicked, the application presents a loading page and then switches to another page after three seconds. This functionality tests if the crawling module can handle a real-world flow rather than only load a page, quickly extract its links, and move on.

Required/non-empty text field: When creating a new entry or searching for entries, the test application only continues if text has been entered in the text field or search query field. This is done by both setting the `required` HTML attribute (an HTML5 feature) and enforcing it with JavaScript. Doing so tests if the vulnerability scanner correctly fills in input fields.

Checkbox: Included on the one of the pages is a checkbox that, if checked, will perform the search when the corresponding button is clicked. If not checked, the application will navigate back to the start page. This serves to test if the crawler handles control elements and evaluates scenarios with and without them.

5.4 SPARE–SPA Rewriting for Exploitability

SPAs include elements which vulnerability testing tools only partially can interact with, or cannot interact with at all. After receiving the HTML document JavaScript is loaded, adding interaction elements such as event binding. While an interactive client can change views or manipulate data, vulnerability scanners that will not execute JavaScript cannot reach this functionality. For penetration testing tools that have no capability to support JavaScript this restricts vulnerability discovery capabilities. Furthermore, even if JavaScript is partially supported through a limited execution environment, this can also limit scanners' efforts of exploring the application.

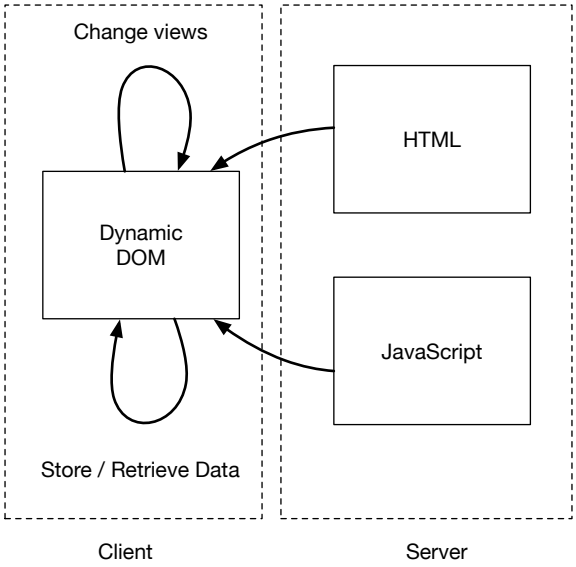

Figure 5.4: SPAs, consisting of HTML and JavaScript code are transferred to the client. Users interact with the SPA locally, state changes are not visible to the server. Except, if data is explicitly sent to the server.

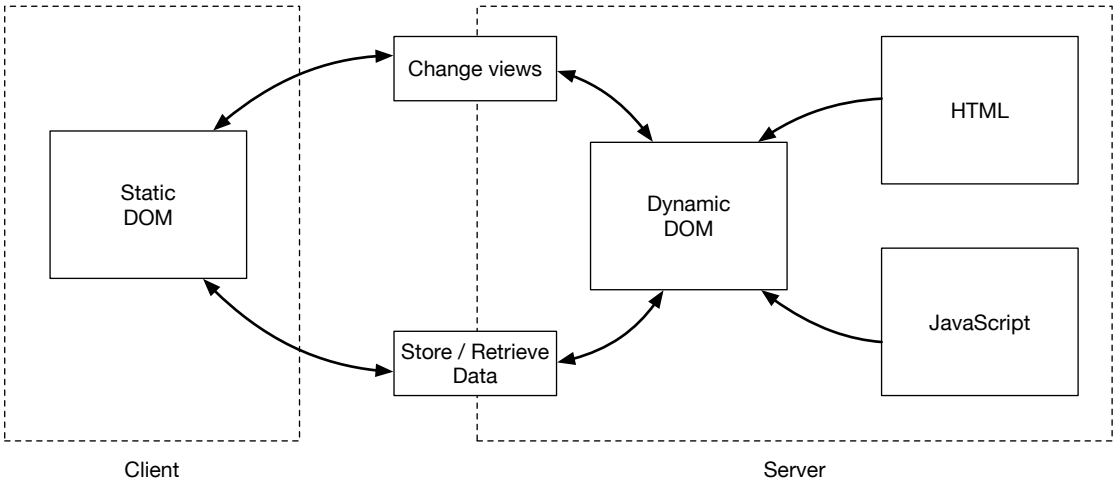

Figure 5.5: After rewriting a SPA, the DOM state is stored on the server-side, encapsulating client-side features. Users (or penetration testing tools) can interact with it in the request/response paradigm. The DOM is rendered server-side and the result returned as a static document. The state becomes visible to the server.

To this end, I wrote SPARE (SPA Rewriting for Exploitability), a prototype NodeJS server that shifts the execution of program logic from the client to the server. Vulnerability scanners are presented with a compatible version of the program that allows for penetration testing. The server acts as proxy between a scanner and an application it would not be able to test otherwise. The system is a prototype operating in semi-automated fashion, and was implemented to rewrite Angular applications, one of the most popular frameworks in use for SPAs.

SPAs often use the localStorage API [2] to store data between sessions. As jsdom does not support localStorage, to overcome this shortcoming I use a wrapper around localStorage methods to transparently write to and read from cookies.

The system uses a virtual DOM on the server-side that was implemented on top of jsdom[3]. The jsdom package implements a subset of the browser that is sufficient for rendering pages for vulnerability scanners. The rewriting includes translation of views into routes, and management of program state via cookies. An overview of regular SPAs is depicted in Figure 5.4, and after rewriting in Figure 5.5. When the server receives a request, the following actions are performed:

1. Create DOM based on request parameters. The DOM includes all resources which would be used in the regular environment for that SPA.

2. Restore state from cookies. If cookies are included in the request, SPARE will restore them before creating a new DOM. Cookies are accessed through a transparent localStorage wrapper as jsdom does not support localStorage.

3. Apply action based on request and state by injecting generated JavaScript code. This can include changing views or adding data to the application.

4. Remove all JavaScript elements: script files and inline code. The DOM is reduced to a static version by removing all dynamic elements.

[2] https://www.w3.org/TR/webstorage/
[3] https://github.com/jsdom/jsdom

5. Iterate through possible view changes, add hyperlink elements and register them as new routes in the server. The added hyperlinks make the application more accessible to scanners and crawlers.

6. Rewrite forms to routes to allow saving data. SPA forms become POST routes that store data via code injection.

7. Return rendered DOM as static HTML document, carrying state in cookie.

5.4.1 State Management and Manipulation

The prototype implementation of SPARE creates a new DOM for every request, applying state from cookies. The server-side DOM is stateless, as state is carried exclusively in the cookie and DOM is discarded after the request. I receive cookies from the client-browser, apply them to the server-side DOM, and respond with an updated cookie. SPAs can store data in localStorage, a feature that is not supported by jsdom, the prototype redirects access to localStorage to cookies.

Cookies are limited in size (4093 Byte) and represent a limitation of the prototype. The system could be extended through a server-side database holding state, and only using the cookie to store a reference to state in a database. As speed optimization, DOM instances could be cached between sessions.

Forms are rewritten to POST routes, where the submitted data is applied to the program state by injecting generated code, accessing internal Angular calls directly. An example for a template triggering is displayed in Figure 5.7. The program is executed right after injecting it as a script tag. Before sending the rendered DOM as a response, all program code is removed, including such injected code.

5.4.2 Accessibility of views

Penetration testing tools have shortcomings in exploring applications (crawling). A suggested solution is to make all views accessible, by rewriting views as routes, and adding links to these to the main page. As such, views which are otherwise inaccessible become directly

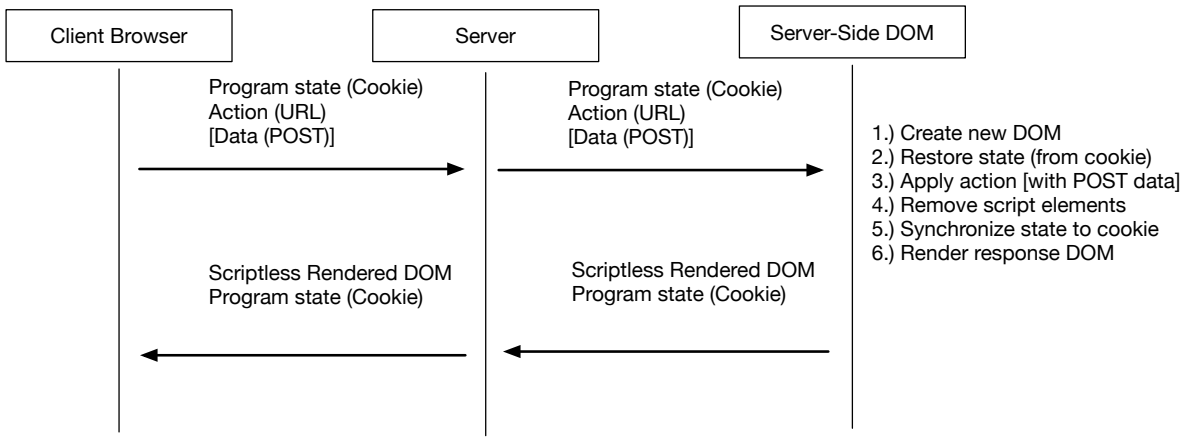

Figure 5.6: Overview of data transferred and actions taken with each state change for a rewritten SPA.

```
1    (function (){
2        $("div[ng-controller='%CONTROLLER%']").scope().%ACTION%(%POSTDATA%)
3    })();
```

Figure 5.7: Route and action-specific code injection into JSDOM object. The code is added to the DOM as a new `script` element, executed immediately, and removed from the DOM before being rendered for a response.

reachable. A limitation of this approach is that views could require specific actions to be taken earlier to render correctly. As all views are offered, this could lead to errors while executing the program.

5.5 Evaluation

I tested 13 black-box vulnerability scanners and one crawler (Crawljax [26]) by running them against each version of the test application. To decide on the scanners to evaluate, I used a website maintained by Chen [24], which lists current black-box web vulnerability scanners. For the commercial scanners, I used all those which offered trial versions, and I used as many of the open-source tools as possible, however I was unable to evaluate them all, due to time constraints.

The scanners were composed of 6 open source tools (Grabber [37], Skipfish [39], Vega [90],

Name	Version	License
Acunetix	9.5 Build 20140903	Commercial
Crawljax	3.5.1	ASFv2
Google Cloud Security Scanner	N/A	Commercial
Grabber	0.1	BSD
IBM Security AppScan	9.0.2	Commercial
ParosPro	1.9.12	Commercial
N-Stalker	10.14.1.5	Commercial
Skipfish	2.10b	ASFv2
Tinfoil Security	N/A	Commercial
Vega	1.0	EPL1
w3af	1.6.0.5	GPLv2
Wapiti	2.3.0	GPLv2
WebInspect	10.30.507.10	Commercial
ZED Attack Proxy	2.3.1	ASFv2

Table 5.1: Characteristics of the scanners evaluated

w3af [101], Wapiti [91], and ZED Attack Proxy [69]) and 7 commercial tools (Acunetix [11], Google Cloud Security Scanner [40, 57], IBM Security AppScan [47], ParosPro [63], N-Stalker [64], Tinfoil Security [97], and WebInspect [45]). Although I used evaluation versions of the commercial tools, each made available the functionality necessary for these tests. A summary of these tools is presented in Table 5.1, which contains the name of the scanner, the version used, and the license information. The cloud-based scanners have N/A as their version, as they have no externally visible version information.

5.5.1 Setup

Each scanner was run against the Angular, Backbone, and Ember implementations of the test application with both default and manual configuration (for a total of 84 runs). The manual configuration consisted of enabling any useful options pertaining to crawling, JavaScript evaluation, and *cross-site scripting (XSS)* [12]. Beyond this, the scanner was simply pointed to the web page and instructed to start scanning.

Google Cloud Security Scanner deserves special mention, as it is a cloud-based black-box web vulnerability scanner that only scans applications on Google App Engine. To test Google Cloud Security Scanner, I ported each implementation of the test application to Google App Engine. Using the Google developer console, separate projects were created for each version of the app. I created a configuration file for each version called app.yaml. This file had directives to Google App Engine, describing how the files should be deployed. No other code

changes were required, and the functionality of the Google App Engine application instances were the same.

5.5.2 Evaluation of Existing Black-box Systems

When evaluating scanners against a traditional server-side web application, the HTTP requests and responses made by the scanner give insight into how the tool works. In the case of client-side web applications, there is no HTTP requests and responses, as all changes to the application state take place within the DOM. Therefore, I devised a method to try to understand how the scanner was crawling and interacting with the client-side web application, and therefore the capabilities of the scanner.

I determined the capabilities of the scanners in a black-box manner, so that the technique would be applicable to the commercial as well as the open-source tools. To determine if the scanner has the ability to execute JavaScript, after the `DOM` [102] of each page of the application loaded, using JavaScript I inserted an image reference (`img` HTML tag). The `src` attribute of the tag would be the server, so that by looking at the log of the server I could determine which pages of the application were visited and executed (each `img` tag contained a unique `src` attribute). To ensure that these `src` attributes would not be discovered by parsing, they were obfuscated in the JavaScript code.

As an additional assurance of the capabilities of the scanner, I included a form containing a text field, and the `action` attribute of the form contained a URL with an intentional XSS vulnerability. In this way, I could use the vulnerability reports of the scanner to determine how much of the application the scanner was able to reach.

5.5.3 Crawling Results

Based on the tests, only three of the tools (Acunetix, Google Cloud Security Scanner, and Crawljax) were able to execute JavaScript and interact with the application. Because Crawljax is solely a crawler and uses a web browser to navigate, Acunetix and Google Cloud Security Scanner are the only black-box scanners tested that were able to execute JavaScript code. ZED Attack Proxy provides a traditional crawler and has an `AJAX` crawler plugin avail-

	About	About-Load	Created	Edit Note	Home	New Note	Search	Search Tags	Search Results	View Note	%
Acunetix											
-Angular	✓	✓			✓	✓					40%
-Backbone	✓	✓			✓						30%
-Ember		✓			✓	✓	✓	✓			50%
Crawljax											
-Angular	✓	✓			✓	✓	✓				50%
-Backbone	✓	✓			✓	✓	✓	✓			60%
-Ember	✓	✓			✓	✓	✓				50%
Google CSS											
-Angular	✓			✓	✓	✓					40%
-Backbone	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	100%
-Ember	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	100%

Table 5.2: Pages reached without any modifications to the testing application

able to download, but because this plugin uses Crawljax and is not included by default, I did not consider ZAP itself able to execute JavaScript. WebInspect was able to locate all of the `XSS` vulnerabilities, but because this was done through parsing the intentional `href` and `action` attributes it is not an indication of the completion of the crawling challenges. IBM Security AppScan includes a JavaScript Security Analyzer extension which is supposed to detect client-side vulnerabilities, however IBM Security AppScan was not able to interact with the application. It reached the index page, which has a form containing an XSS vulnerability, but it did not report the vulnerability. The remaining scanners did not report any of the `XSS` vulnerabilities, which constitutes a failure of the most basic test and suggests a possible bug in their HTML parsing or JavaScript execution abilities.

Considering that only Acunetix, Google Cloud Security Scanner, and Crawljax execute JavaScript, I will focus mainly on their performance against the crawling challenges. These three tools had a varying degree of success with both the challenges and the different JavaScript frameworks, as seen in Table 5.2.

An important detail to note here is that Acunetix was unable to handle Backbone's event binding. As a result I was unable to fully assess Acunetix against many of the other crawling challenges implemented with Backbone, however I can still make inferences about Acunetix's overall capabilities based on its performance against the other two frameworks, along with a version of the test application that was specifically modified to test Acunetix.

Event binding: While Crawljax successfully navigated the test application via the but-

tons bound to events, Acunetix had a mixed degree of success. It was able to successfully handle both `{{action}}` and `{{link-to}}` in Ember but could not handle any of Backbone's view events. With Angular it executed only some events and not others, even when the triggering buttons were directly next to one another and implemented in the same fashion. For example, there are two buttons which are side by side on the main page, yet only one of the paths was followed and the other was not visited. I was unable to determine the reason for this or the crawling methods employed by Acunetix. Google Cloud Security Scanner was able to navigate and reach only some of the pages in Angular, however it was able to reach all pages in both Backbone and Ember.

Data binding: Because the pages of the application were crawled in the order that they were found, the tools evaluated the Home page first and then moved on to the others. The table of existing notes is updated dynamically when the internal model changes, thus the tools must be able to detect and consider this change to a page that they previously evaluated, as this is the only way to view a newly created note. Crawljax was unsuccessful in viewing a created note in all of the frameworks while Acunetix succeeded with Angular and Ember. Acunetix "failed" with Backbone due to its inability to handle the event generated by clicking on a note in the table, but it succeeded when the event was changed to an `href`. Therefore, Acunetix*can* handle data binding in all three frameworks, but its shortcoming in another area prevented it from doing so without modification. Google Cloud Security Scanner was able to create, view, and delete notes in the Angular, Backbone, and Ember versions of the app.

Delayed redirect: Without any execution parameters, Crawljax was unable to handle the delayed redirect because it would move on from a page too quickly. Only once the `-waitAfterEvent` argument was specified, which told it to pause longer than the 3 second "loading," could it handle the redirect in all three of the frameworks. Acunetix was able to handle the delayed redirect in Angular and Backbone but not in Ember, which could be due to the more complex timeout functionality of Ember's *run-loops*. Google Cloud Security Scanner was unable to handle delay in Angular but it was able to handle it in Backbone and

Ember.

Required/non-empty text field: Navigating from the Search page to the Add Search Tags page saw mixed results. The only stipulation is for text to be entered in the search query field, yet this proved to be more of a hindrance than expected. Crawljax only succeeded with Backbone while Acunetix was only successful with Angular. When the event binding was changed to an `href`, Acunetix was then successful with Ember as well, although still not with Backbone. Google Cloud Security Scanner was able to create notes by filling out the mandatory fields in Angular, Backbone, and Ember.

Checkbox: Although the only requirement to transition from the Add Search Tags page to the Search Results page is having the checkbox marked, neither Acunetix nor Crawljax was able to do this in any scenario. This is surprising due to the fact that many web applications include more complex controls. Google Cloud Security Scanner was unable to pass the checkbox test in the Angular implementation but it did succeed in the Backbone and Ember implementations.

Dealing with side-effects: An additional difficulty for crawling modules that presented itself is the consequence of actions. The test application provides a deletion button, if a data was entered or the application was seeded with data, but the tool invoked this button before crawling, it would never reach pages related to viewing or editing that data. This occurred with Crawljax, as it sequentially evaluated the Home page and reached the delete button before the data display table. This was also the case with Google Cloud Security Scanner, because it was able to create a new note but it was unable to reach many of the other pages because it deleted the note before it could proceed any further.

Framework Support: Google Cloud Security Scanner had little success in crawling the app built using Angular framework. This is surprising, as it did the best in the tests, as it was able to successfully crawl and reach all pages of the Backbone and Ember applications. It is noteworthy that Google maintains the Angular framework and yet, their security scanner

	Event Binding	Data Binding	Delayed Redirect	Required Field	Checkbox	%
Acunetix						
-Angular	✓	✓	✓	✓		67%
-Backbone		✓	✓			33%
-Ember	✓	✓		✓		50%
Crawljax						
-Angular	✓		✓			33%
-Backbone	✓		✓	✓		50%
-Ember	✓		✓			33%
Google Cloud Security Scanner						
-Angular		✓		✓		50%
-Backbone	✓	✓	✓	✓	✓	100%
-Ember	✓	✓	✓	✓	✓	100%

Table 5.3: Crawling challenges successfully completed

was unable to handle this framework. Furthermore, as this is a cloud scanner that will only scan Google App Engine web applications, it is difficult for us to determine what exactly about Angular caused Google Cloud Security Scanner trouble.

5.5.4 Measuring and Comparing Capabilities

While Table 5.2 details the pages reached *without* modification to the test application, it provides a baseline evaluation of each tool's abilities, Table 5.3 depicts the crawling challenges completed *with* modifications necessary to test individual functionality. Any alterations to the test application were made only if a tool's shortcoming in one area prevented testing another. These modifications included exchanging event binding for `href`s and seeding the application with existing notes. Whenever this was necessary for testing, I ensured that any modifications did not compromise the integrity of the specific challenge that required further individual testing.

As previously stated, 10 of the 13 tools I tested were unable to execute JavaScript in any capacity. Between the three remaining tools, Google Cloud Security Scanner performed the best in the unmodified test, reaching 80% (24/30) of the pages followed by Crawljax which reached 53% (16/30) of the pages across the three frameworks while Acunetix reached 40% (12/30). In the individual challenges, Google Cloud Security Scanner was successful in 72% (13/18) across the three frameworks while Acunetix was successful in 50% (9/18) followed by Crawljax, which completed 39% (7/18). Because Crawljax is solely a crawler, Google Cloud Security Scanner was the top performing black-box scanner that I tested, yet even it

	# reported	# detected	(# reported - # detected)	% redundancy
Angular	32	4	28	87.5%
Backbone	162	10	152	93.82%
Ember	136	10	126	92.64%

Table 5.4: Redundant reports and redundancy rate of Google Cloud Security Scanner

had shortcomings and issues.

5.5.5 Security Effectiveness

The focus of this work is on understand the crawling capabilities of black-box web vulnerability scanners, however, because I used a simple XSS vulnerability to measure the crawling progress of the scanners, I was able to gain some insights into the security functionality of the tools.

Typically, the effectiveness of vulnerability analysis tools depend on the number of true positives (reports that are actually vulnerable), false positives (reports that are not actually vulnerable), and false negatives (vulnerabilities that are not reported). However, in this study I identified a new type of metric, the *redundant reports*. This occurs when a scanner reports the same vulnerability multiple times. This behavior was observed in Google Cloud Security Scanner. I define the *redundancy rate* (similar to the false positive calculation) as

$$\frac{\#\ vulnerabilities\ reported - \#\ vulnerabilities\ detected}{\#\ vulnerabilities\ reported}$$

This rate describes what percentage of the vulnerability reports were redundant. The results for Google Cloud Security Scanner, which was the only scanner to demonstrate this behavior, are shown in Table 5.4. For the Angular implementation, out of the 10 vulnerable pages, 4 were detected and each of these were reported 8 times, hence the total number of vulnerabilities reported was 32. Though all vulnerabilities were detected in Backbone and Ember, the number of redundant reports was even higher: 162 vulnerabilities were reported in Backbone with only 10 real vulnerabilities. As for Ember, Google Cloud Security Scanner reported 136 vulnerabilities from 10 real vulnerabilities. The issue here is that even though there were 10 pages with the same XSS vulnerability, Google Cloud Security Scanner used different types of injection vectors for each detected page, and reported each successful re-

sult, thus increasing the count of detected vulnerability. Similar to false positives, redundant reports erode the developer's confidence in the tool, as the total number of reported vulnerabilities is larger than the actual number of vulnerabilities, and it is up to the developer to wade through and understand all the reports. However, redundant reports are distinct from false positives, as false positives are vulnerability reports that are not vulnerable, while redundant reports are vulnerability reports that are actually vulnerable, but reported multiple times.

5.5.6 SPARE Results

The goal of rewriting SPAs is to make them more accessible to vulnerability scanners which otherwise cannot explore them or find vulnerabilities. The SPARE prototype is designed for Angular and I evaluated it on the the test application used for capability evaluation with Acunetix.

The project was originally measuring how well penetration testing tools can explore SPAs, rather than find vulnerabilities. Acunetix was previously only able to access 40% of all views before rewriting 5.2. Views it could not explore include *Created, Edit Note, Search, Search Tags, Search Results,* and *View Note*. After rewriting the application through SPARE, all view changes and forms are rewritten to provide `hrefs` and HTML forms. This step allows the crawling module to bypass tests such as checkboxes and required data input fields, and data dependencies as requiring prior input. Before identifying vulnerabilities, tools are required to access the relevant pages. Making these views accessible is a step towards measuring increased vulnerability finding, and should be analyzed in future work.

5.6 Discussion

This work found that not only is it difficult for existing black-box vulnerability scanners to explore client-side web applications, but also that the complexity of and incongruity among various JavaScript frameworks further complicates comprehensive analysis.

Two of the key reasons that traditional crawlers are unsuccessful in this context are that they fail to fully replicate browser behavior and do not effectively discover application

resources [79]. To handle client-side applications, crawlers must have the ability to execute or analyze JavaScript in a similar manner to a browser's JavaScript engine, which 10 of the 13 tested did not. They can no longer simply parse HTML for a few keywords; they must encompass multiple frameworks' implementations of event binding and navigating pages. Tools such as SPARE can be used to enhance the state of the art, alternatively scanners could be enhanced to provide full JavaScript execution.

Client-side frameworks have become popular and pervasive, and thus support for these technologies must be enhanced. Even among scanners that have been improved there is a need to consistently adapt to the diversity of emerging frameworks and technologies. For example, although Google Cloud Security Scanner performed the best of the tested tools, it faced considerable challenge in the Angular version. Although a tool may be very sophisticated, having a shortcoming in one area can prohibit it from succeeding as a whole.

The main challenge of client-side web applications, from the perspective of a black-box vulnerability scanner, is understanding how to get input to the application, in order to fuzz the input and test for vulnerabilities. Each client-side web application has significant freedom to choose the way that input is sent to the application. Server-side web applications have, for the most part, standardized on sending input to the application through the query or path of the URL, and black-box vulnerability scanners take advantage of this standardization to fuzz accordingly. However, client-side web applications have significantly more freedom, both in how the input is encoded (name/value pairs separated by the traditional `=` or any other delimiter) and the location in the URL of the parameters (in the URL fragment or elsewhere).

These *custom interfaces* are a fundamental challenge that is unique to client-side web applications. For black-box vulnerability analysis tools to be effective at finding vulnerabilities in client-side web application, they must (1) crawl the application, (2) understand the interface, and (3) use this understanding to properly fuzz the application. The new step here is understanding the interface of the client-side web application, and this is a research challenge that must be tackled in order to extend the benefits of black-box vulnerability scanners to client-side web applications. Tools such as SPARE offer an option to enhance capabilities of such scanners and should be further explored.

5.7 Future Work

This project benchmarks black-box penetration testing tools and suggests a prototype to overcome their shortcomings. There are multiple avenues for engineering improvement, such as performance, flexibility, and adoption for other frameworks.

One area of possible improvement is management of state. The prototype creates a new DOM object for every request and state is written to, and restored from cookies. The system could be extended to expose the state in a direct manner, such as generate forms for objects. Vulnerability testing tools with advanced fuzzing capabilities could directly mutate state then.

As JSDOM is simulating a browser with limited capabilities, SPARE could be ported to a real browser, such as headless Chrome. This would close the gap and present the penetration testing tools with correct rendering of the page. This would require a proxy system translating pentesting events to actions in the browser. However, this would still require for the application to be rewritten, as the tools would not be able to interact with elements, even if they are rendered correctly.

Furthermore, not all events can be shifted from the client to the server. For example mouseover events cannot be represented as server-side effect due to lack of pointer object.

SPARE allows to access all views immediately, this can lead to states not intended by the original application. For example, a scanner could call *Delete* before creating elements, this could otherwise be prevented by program logic.

A solution that would improve penetration testing tools is to make them aware of JavaScript and use the program code to their benefit. This would make rewriting obsolete.

5.8 Chapter Summary

This chapter evaluates how black-box penetration testing tools interact with SPAs, and enhances access through a SPA-rewriting prototype, SPARE. The results of the evaluation clearly demonstrate that the current offerings of black-box vulnerability scanners lack the capabilities vital to keep up with progressive technologies. The paradigm shift of server-side

code execution to more code on the client-side has not been followed by these penetration testing tools. I have identified a number of challenges that scanners need to overcome when testing modern web applications, and even though some tools have begun undertaking this task, even these are far from complete.

Additionally, although the primary focus is on client-side frameworks, it is also clear that support for new client-side web applications must be improved. Both Acunetix and Crawljax had marginal success with a required text field and no success with a required checkbox. These are commonplace scenarios and the failure to handle them is a significant shortcoming.

I found that, although a tool may be sophisticated, a deficiency in one area can lead to failure as a whole. Acunetix was fairly sophisticated, yet its inability to handle Backbone's event binding rendered it useless against the unmodified testing application.

As such, these tools must develop the ability to *understand* the interface of client-side web applications, in order to fuzz the correct input. Only by doing so will vulnerability scanners maintain the ability to evaluate security flaws in the ever-evolving world of web applications.

With the SPARE prototype I pushed client-side code back to the server. With this approach SPAs can be lead into the traditional request/response paradigm and enhance access scanners have to applications.

Chapter 6

Papers

This chapter gives an overview of publications this thesis is built upon, and publications that are not part of it. The former are mentioned briefly as they are discussed throughout the thesis, while the latter are discussed in more detail.

6.1 Thesis Publications

Parts of this thesis are based on the following papers. Two are peer-reviewed and published [106, 107], the third one is in preparation for submission.

M. Weissbacher, T. Lauinger, and W. Robertson. Why is CSP Failing? Trends and Challenges in CSP Adoption. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2014.

M. Weissbacher, E. Mariconti, G. Suarez-Tangil, G. Stringhini, W. Robertson, and E. Kirda. Ex-ray: Detection of history-leaking browser extensions. In *Annual Computer Security Applications Conference (ACSAC)*, 2017.

The material Chapter 5 is based off is not yet published or submitted for publication.

6.2 Other Work

This section summarizes my work outside of the scope of the main thesis. All listed projects are completed and published.

ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities [108]

Client-side Validation (CSV) Vulnerabilities can occur when JavaScript-based Web applications use modern client-side messaging primitives. ZigZag is a system for hardening JavaScript-based Web applications against such CSV attacks. The system transparently instruments client-side code to perform dynamic invariant detection on security-sensitive code, generating models that describe how – and with whom – client-side components interact. Learned invariants are then enforced through a subsequent instrumentation step. ZigZag is capable of automatically hardening client-side code against both known and previously-unknown vulnerabilities. I used static analysis to target important functions when recompiling, and dynamic analysis to prevent attacks.

To make dynamic JavaScript analysis more accessible to other researchers, I published a patch for Google Closure Compiler that enables program instrumentation via templates. It was accepted into the mainline branch in September 2015.

Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance [81]

Software permeates every aspect of our world, from our homes to the infrastructure that provides mission-critical services.

As the size and complexity of software systems increase, the number and sophistication of software security flaws increase as well. The analysis of these flaws began as a manual approach, but it soon became apparent that a manual approach alone cannot scale, and that tools were necessary to assist human experts in this task, resulting in a number of techniques and approaches that automated certain aspects of the vulnerability analysis process.

Recently, DARPA carried out the Cyber Grand Challenge, a competition among autonomous vulnerability analysis systems designed to push the tool-assisted human-centered paradigm into the territory of complete automation, with the hope that, by removing the human factor, the analysis would be able to scale to new heights. However, when the autonomous systems were pitted against human experts it became clear that certain tasks, albeit simple, could not be carried out by an autonomous system, as they require an understanding of the logic of the application under analysis.

Based on this observation, I propose a shift in the vulnerability analysis paradigm, from tool-assisted human-centered to human-assisted tool-centered. In this paradigm, the automated system orchestrates the vulnerability analysis process, and leverages humans to perform well-defined sub-tasks, whose results are integrated in the analysis. As a result, it is possible to scale the analysis to a larger number of programs, and, at the same time, optimize the use of expensive human resources.

This paper covers the design for a human-assisted automated vulnerability analysis system, describes its implementation atop an open-sourced autonomous vulnerability analysis system that participated in the Cyber Grand Challenge, and evaluates and discuss the significant improvements that non-expert human assistance can offer to automated analysis approaches.

BabelCrypt: The Universal Encryption Layer for Mobile Messaging Applications [70]

Internet-based mobile messaging applications have become a ubiquitous means of communication, and have quickly gained popularity over cellular short messages (SMS). Unfortunately, from a security point of view, free messaging services do not guarantee the privacy of users. For example, free messaging providers can record and store exchanged messages indefinitely to collect information about specific users. Moreover, these messages can be accessed by criminals who gain access to social media accounts. In this paper, I introduce BabelCrypt, a system that addresses the problem of automatically retrofitting arbitrary mobile chat applications with end-to-end encryption. My system works by transparently interfacing

with the original client applications supplied by the respective service providers. It does not require any modification to the individual applications, nor does it require any knowledge or customization for specific chat applications. Babelcrypt is able to automatically inject control messages in-band, using the underlying application's message exchange mechanism, and thus supports running arbitrarily complex encryption protocols such as OTR. I successfully used BabelCrypt with a number of popular messaging applications including Facebook Messenger, WhatsApp, and Skype. The evaluation shows that BabelCrypt provides end-to-end security for arbitrary messaging applications while satisfactorily preserving the original user experience of the messaging application.

Chapter 7

Conclusion

The increased use of the Web platform, combined with the shift of Web program complexity from the server to the client, presents new challenges which this thesis is addressing. The growth in complexity of client-side Web software opens new attack vectors where vulnerabilities can be exploited and hidden functionality can act against assumed privacy.

In particular, the **thesis statement**:

Exploitable software vulnerabilities and hidden functionality permeate software. Analysis of client-side software in search of vulnerabilities and unintended behavior becomes increasingly important as the Web program logic shifts from the server to the client. In this thesis I develop novel methods and automated mechanisms to reduce the impact of client-side vulnerabilities and hidden privacy invasions. I show that such an approach is both feasible and effective. I investigate shortcomings and possible avenues for enhancement of CSP, Web vulnerability scanners, and privacy of browser extensions.

I back this claim as follows:

- The need for such systems has been demonstrated by identifying shortcomings, measuring and evaluating them in the three areas. I show that CSP is underutilized, too complex, and often misconfigured. Browser extensions often leak private brows-

ing data, and black-box Web vulnerability scanners underperform in exploring SPAs leaving vulnerabilities to be exploited in production.

- The resulting systems reduce the impact on client-side security in the three areas as follows: Automatically generating CSP rules enables website owners to employ CSP, reducing exposure to content injection. Detection systems for history leaks in browser extensions can expose hidden functionality, reducing the impact on users. In fact, during the course of this thesis extensions were removed from an extension store, effectively reducing the privacy impact on 8 million users. For black-box vulnerability tools, I highlighted shortcomings that can be used to extend these systems. However, I also wrote a prototype that exposes potentially unexplored pages to vulnerability scanners, making them more accessible to vulnerability search.

Given the above hypothesis, this thesis presents novel methods and automated mechanisms to reduce the impact of client-side vulnerabilities and hidden privacy invasions. I show that such an approach is both feasible and effective, by investigating shortcomings and possible avenues for enhancement of CSP, Web vulnerability scanners, and privacy of browser extensions. In more detail:

Content Security Policy. I present a long-term study on CSP as it is deployed on the Web. I have found that CSP adoption significantly lags other Web security mechanisms, and that even when it has been adopted by a site, it is often deployed in a way that negates its theoretical benefits for preventing content injection and data exfiltration attacks.

In addition, by enabling CSP at four sites, I observed that it is difficult for third parties to deploy CSP, either through incremental deployment using report-only mode or through Web application crawling to semi-automatically generate policies.

CSP clearly holds great promise as a Web security standard, but I can only conclude that it is difficult for most sites to deploy it to its full potential in its current form. It is my hope that the improvements I suggest here, as well as upcoming features of the 1.1 draft, will allow site operators and developers to make effective use of content security policies and result in a safer Web ecosystem.

History Leaking Browser Extensions. I show that history leaks in browser exten-

sions are a prevalent problem, and I introduce new methods of detecting privacy-violating browser extensions independently of their protocol. I use a combination of supervised and unsupervised methods to find features characteristic to tracking in extensions. I introduce Ex-Ray, a prototype implementation of my approach for the Chrome browser, and find two extensions in the official Chrome Web Store which leak private information in previously undetectable ways.

SPA Rewriting to Enhance Vulnerability Discovery. I demonstrate that the current offerings of black-box vulnerability scanners lack the capabilities vital to keep up with progressive technologies. I have identified a number of challenges that scanners need to overcome when testing modern Web applications, and even though some tools have begun undertaking this task, even these are far from complete. As such, these tools must develop the ability to *understand* the interface of client-side web applications, in order to fuzz the correct input. Only by doing so will vulnerability scanners maintain the ability to evaluate security flaws in the ever-evolving world of web applications. With the SPARE prototype I pushed client-side code back to the server. With this approach SPAs can be lead into the traditional request/response paradigm and enhance access scanners have to applications.

To conclude, I presented systems developed for measurement and detection of security properties of client-side Web applications. I demonstrate both the importance of measurement of such properties, as well as the effectiveness and feasibility of systems used to detect such shortcomings and find hidden privacy invasions.

7.1 Future Work

There are several open questions as results of the presented work. For CSP and Web vulnerabilities in general the ideal solution would be to allow Web applications to be developed securely out of the box, free of vulnerabilities. This would make both CSP and black-box Web vulnerability testing tools obsolete. However, as such a goal seems elusive, more tangible work directions include development of Web frameworks that automatically generate CSP directives, and development of applications scanners that are able to interact

with JavaScript directly.

While I demonstrated technological measures for detection of hidden privacy leaks, technology is lacking understanding of program intent. Extensions may be purposefully leaking data as part of their functionality and the user is well aware of it. A promising research direction could be exploring options of merging measurement of program behavior with perceived intent. Furthermore, exploring paths of discouraging extension developers from packaging such illicit functionality could be fruitful.

Bibliography

[1] DNS Prefetching - The Chromium Projects.

[2] The Platform for Privacy Preferences 1.0 (P3P1.0) Specification, 2002.

[3] IE8 Security Part IV: The XSS Filter, 2008.

[4] IE8 Security Part V: Comprehensive Protection, 2008.

[5] RFC 6797 - HTTP Strict Transport Security (HSTS), 2012.

[6] Closure Compiler, 2013.

[7] Content Security Policy 1.1, 2013.

[8] Cross-Origin Resource Sharing, W3C Candidate Recommendation 29 January 2013, 2013.

[9] Postcards from the post-XSS world, 2013.

[10] RFC 7034 - HTTP Header Field X-Frame-Options, 2013.

[11] Acunetix. Acunetix Web Vulnerability Scanner. `http://www.acunetix.com/`.

[12] Acunetix. Cross-site scripting (xss) attack. `http://www.acunetix.com/websitesecurity/cross-site-scripting`.

[13] Acunetix. The role and function of black box scanners. `http://www.acunetix.com/websitesecurity/blackbox-scanners`.

[14] A. Aggarwal, B. Viswanath, L. Zhang, S. Kumar, A. Shah, and P. Kumaraguru. I spy with my little eye: Analysis and detection of spying browser extensions. *arXiv preprint arXiv:1612.00766*, 2016.

[15] AnantaSec. Web Vulnerability Scanners Evaluation. `http://anantasec.blogspot.com/2009/01/web-vulnerability-scanners-comparison.html`, 2009.

[16] AngularJS. Angularjs. `https://angularjs.org`.

[17] AngularJS. What is angular? `https://docs.angularjs.org/guide/introduction`.

[18] S. Arshad, A. Kharraz, and W. Robertson. Identifying extension-based ad injection via fine-grained web content provenance. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Paris, FR, 2016.

[19] Backbone.js. Backbone.js. `http://backbonejs.org`.

[20] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. 2010.

[21] A. Barth, C. Jackson, and J. C. Mitchell. Securing Frame Communication in Browsers. *Communications of the ACM*, 2009.

[22] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[23] S. Calzavara, A. Rabitti, and M. Bugliesi. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[24] S. Chen. The prices vs. features of web application vulnerability scanners. `http://sectoolmarket.com/price-and-feature-comparison-of-web-application-scanners-unified-list.html`, 2015.

[25] L. F. Cranor, J. Reagle, and M. S. Ackerman. Beyond concern: Understanding net users' attitudes about online privacy. *The Internet upheaval: raising questions, seeking answers in communications policy*, pages 47–70, 2000.

[26] Crawljax. Crawljax: Crawling ajax-based web applications. `http://crawljax.com`.

[27] M. Curphey and R. Araujo. Web Application Security Assessment Tools. *IEEE Security and Privacy*, 4(4), 2006.

[28] detectify labs. Chrome extensions - aka total absence of privacy. `https://labs.detectify.com/2015/11/19/chrome-extensions-aka-total-absence-of-privacy/`, 2015.

[29] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.

[30] E. Dorn and H. Howard. Ember.js and angularjs: Two architectures compared. `http://www.slideshare.net/lrdesign/architecture-emberjs-and-angularjs`, 2012.

[31] A. Doupé, M. Cova, and G. Vigna. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Bonn, Germany, 2010.

[32] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna. deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[33] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2011.

[34] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. X. Song. Dynamic spyware analysis. In *USENIX annual technical conference (ATC)*, 2007.

[35] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we need hundreds of classifiers to solve real world classification problems? *The Journal of Machine Learning Research (JMLR)*, 15(1):3133–3181, Jan. 2014.

[36] FusionBrew. Angularjs vs backbone.js vs ember.js - choosing a javascript framework. `http://blog.fusioncharts.com/2014/08/angularjs-vs-backbone-js-vs-ember-js`, 2014.

[37] R. Gaucher. Grabber. `http://rgaucher.info/beta/grabber`.

[38] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing (TRUST)*. Springer, 2012.

[39] Google. Skipfish. `https://code.google.com/p/skipfish`.

[40] Google. Google cloud security scanner. `https://cloud.google.com/tools/security-scanner/`, 2015.

[41] D. Gruber. The top 10 javascript frameworks, and the communities behind them. Technical report, Black Duck, 2014.

[42] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy (Oakland)*, 2011.

[43] D. Hausknecht, J. Magazinius, and A. Sabelfeld. May i?-content security policy endorsement for browser extensions. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.

[44] S. Heule, D. Rifkin, A. Russo, and D. Stefan. The most dangerous code in the browser. In *USENIX Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, 2015.

[45] Hewlett-Packard. Webinspect. `http://www8.hp.com/us/en/software-solutions/webinspect-dynamic-analysis-dast`.

[46] J. Howell, C. Jackson, H. J. Wang, and X. Fan. Mashupos: Operating system abstractions for client mashups. In *HotOS*, 2007.

[47] IBM. Ibm security appscan. `http://www-03.ibm.com/software/products/en/appscan-standard`.

[48] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *International Conference on World Wide Web (WWW)*, 2007.

[49] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: A Web Vulnerability Scanner. In *Proceedings of the International World Wide Web Conference*, 2006.

[50] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security Symposium*, San Diego, CA, 2014.

[51] S. Lekies, K. Kotowicz, S. Groß, E. A. V. Nava, and M. Johns. Code-reuse aacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[52] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *USENIX Security Symposium*, Austin, TX, 2016.

[53] C. Lever, P. Kotzias, D. Balzarotti, J. Caballero, and M. Antonakakis. A Lustrum of Malware Network Communication: Evolution and Insights. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2017.

[54] D. Lewis. Counterfactuals and comparative possibility. *Journal of Philosophical Logic*, 2:2161–2173, 1973.

[55] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti. Detecting Environment-Sensitive Malware. In *Recent Advances in Intrusion Detection (RAID)*, 2011.

[56] N. K. Malhotra, S. S. Kim, and J. Agarwal. Internet users' information privacy concerns (iuipc): The construct, the scale, and a causal model. *Information systems research*, 15(4):336–355, 2004.

[57] R. Mann. Using google cloud platform for security scanning. `http://googlecloudplatform.blogspot.com/2015/02/using-google-cloud-platform-for.html`, 2015.

[58] E. Mariconti, J. Onaolapo, G. Ross, and G. Stringhini. The cause of all evils: Assessing causality between user actions and malware activity. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2017.

[59] A. Matthews. Flame on! a beginner's guide to ember.js. `http://www.adobe.com/devnet/archive/html5/articles/flame-on-a-beginners-guide-to-emberjs.html`, 2012.

[60] J. R. Mayer and J. C. Mitchell. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[61] S. McConnel. Code complete, 1993.

[62] L. A. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[63] MileScan. Parospro. `http://www.milescan.com`.

[64] N-Stalker. N-stalker: The web security specialists. `http://www.nstalker.com`.

[65] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[66] T. Oda and A. Somayaji. *Enhancing Web Page Security with Security Style Sheets*. Carleton University, 2011.

[67] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji. SOMA: Mutual Approval for Included Content in Web Pages. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[68] L. Olejnik, M.-D. Tran, and C. Castelluccia. Selling Off Privacy at Auction. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[69] OWASP. Owasp zed attack proxy project. `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`.

[70] A. T. Ozcan, C. Gemicioglu, K. Onarlioglu, M. Weissbacher, C. Mulliner, W. Robertson, and E. Kirda. BabelCrypt: The Universal Encryption Layer for Mobile Messaging Applications. In *Financial Cryptography and Data Security (FC)*, 2015.

[71] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[72] H. Peine. Security test tools for web applications. Technical Report 048.06, Fraunhofer IESE, 2006.

[73] T. Reenskaug. Models - views - controllers. Technical report, Xerox Parc, 1979.

[74] P. Refaeilzadeh, L. Tang, and H. Liu. Cross-validation. In *Encyclopedia of database systems*, pages 532–538. Springer, 2009.

[75] F. Roesner, T. Kohno, and D. Wetherall. Detecting and defending against third-party tracking on the web. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, 2012.

[76] M. Rouse. Fuzz testing (fuzzing). `http://searchsecurity.techtarget.com/definition/fuzz-testing`, 2010.

[77] M. Samuel, P. Saxena, and D. Song. Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[78] G. A. Seber and A. J. Lee. *Linear regression analysis*. John Wiley & Sons, 2012.

[79] S. Shah. Hacking web 2.0 applications with firefox. Technical report, Symantec, 2010.

[80] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 2011.

[81] Y. Shoshitaishvili, M. Weissbacher, L. Dresel, C. Salls, R. Wang, C. Kruegel, and G. Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[82] A. J. Smola and B. Sch lkopf. A tutorial on support vector regression. *Statistics and computing*, 14(3), 2004.

[83] P. Snyder, L. Ansari, C. Taylor, and C. Kanich. Browser feature usage on the modern web. In *Proceedings of the 2016 Internet Measurement Conference*, 2016.

[84] P. Snyder, C. Taylor, and C. Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 179–194. ACM, 2017.

[85] S. Son and V. Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013.

[86] S. Sousa, F. G. Martin, M. C. M. Alvim-Ferraz, and M. C. Pereira. Multiple linear regression and artificial neural networks based on principal components to predict ozone concentrations. *Environmental Modelling & Software*, 22(1):97–103, 2007.

[87] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In *International Conference on World Wide Web (WWW)*, 2010.

[88] O. Starov and N. Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *Proceedings of the 26th International Conference on World Wide Web*, 2017.

[89] B. Stock, M. Johns, M. Steffens, and M. Backes. How the web tangled itself: Uncovering the history of client-side web (in)security. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security '17)*, Vancouver, BC, 2017. USENIX Association.

[90] Subgraph. Vega. `https://subgraph.com/vega`.

[91] N. Surribas. Wapiti. `http://wapiti.sourceforge.net`.

[92] L. Suto. Analyzing the Effectiveness and Coverage of Web Application Security Scanners. Case Study, 2007.

[93] L. Suto. Analyzing the Accuracy and Time Costs of Web Application Security Scanners, 2010.

[94] K. M. Tan and R. A. Maxion. " why 6?" defining the operational limits of stide, an anomaly-based intrusion detector. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 188–201. IEEE, 2002.

[95] M. Ter Louw and V. Venkatakrishnan. BLUEPRINT: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *IEEE Symposium on Security and Privacy (Oakland)*, 2009.

[96] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. A. Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[97] Tinfoil. Tinfoil security. `https://www.tinfoilsecurity.com/`.

[98] S. Van Acker, D. Hausknecht, and A. Sabelfeld. Data exfiltration in the face of csp. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016.

[99] M. Vieira, N. Antunes, and H. Madeira. Using Web Security Scanners to Detect Vulnerabilities in Web Services. In *Proceedings of the Conference on Dependable Systems and Networks*, 2009.

[100] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. In *Communications of the ACM*, volume 47, 2004.

[101] Web Application Attack and Audit Framework. http://w3af.sourceforge.net/.

[102] W3Schools. Xml dom introduction. http://www.w3schools.com/dom/dom_intro.asp.

[103] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[104] J. Weinberger, A. Barth, and D. Song. Towards Client-side HTML Security Policies. In *Workshop on Hot Topics on Security (HotSec)*, 2011.

[105] M. Weissbacher. These chrome extensions spy on 8 million users. http://mweissbacher.com/blog/2016/03/31/these-chrome-extensions-spy-on-8-million-users/, 2016.

[106] M. Weissbacher, T. Lauinger, and W. Robertson. Why is CSP Failing? Trends and Challenges in CSP Adoption. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2014.

[107] M. Weissbacher, E. Mariconti, G. Suarez-Tangil, G. Stringhini, W. Robertson, and E. Kirda. Ex-ray: Detection of history-leaking browser extensions. In *Annual Computer Security Applications Conference (ACSAC)*, 2017.

[108] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities. In *USENIX Security Symposium*, 2015.

[109] A. Wiegenstein, F. Weidemann, M. Schumacher, and S. Schinzel. Web Application Vulnerability Scanners—a Benchmark. Technical report, Virtual Forge GmbH, 2006.

[110] C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck. A close look on n-grams in intrusion detection: anomaly detection vs. classification. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 67–76. ACM, 2013.

[111] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee. Understanding malvertising through ad-injecting browser extensions. In *International Conference on World Wide Web (WWW)*. ACM, 2015.