ABSTRACT

CHEN, QUAN. Privacy Implications of Emerging Web Technologies. (Under the direction of Alexandros Kapravelos.)

The web of today is vastly different from when it was first introduced decades ago. While the capability for websites to link to arbitrary third-party contents remains a powerful enabling feature of the web, gone are the days when most websites are confined to serving static pages. Scripting languages such as JavaScript allow implementation of dynamic interactive pages that change their layouts in response to users' actions. As with other endeavours of software engineering, websites rarely implement common functionalities themselves but use readily available JavaScript libraries provided by third parties whose interests are not always aligned with those of the including websites. Furthermore, in order to monetize website traffic, site owners routinely include content (such as JavaScript code) from advertisement networks, which are incentivized to know as much about the users as possible. Finally, adding to this complexity is the fact that modern browsers also support extension mechanisms that allow users to customize or enhance their web browsing experiences by installing *browser extensions*, which are small JavaScript programs that are run inside the browser, often with access to special privileged extension APIs. The privacy implications of having all these JavaScript code executing inside the user's browser need to be investigated and understood.

In this dissertation, we focus on using instrumentation of the browser's JavaScript execution environment and its associated API implementations to gather behavior profiles of JavaScript code, loaded from diverse sources who often have conflicting interests, that executes in the users' browsers when they visit the web. We then use the collected logs of script behaviors to measure and understand their privacy implications. Specifically, we propose and implement two distinct sets of instrumentation to the Chromium browser: (i) we augment the V8 JavaScript engine and the WebKit/Blink layout engine to provide dynamic taint tracking capabilities, in order to understand the data flow of privacy-sensitive information, and (ii) in another set of instrumentation, we aim to record in a graph structure the runtime actions taken by JavaScript code, including DOM (Document Object Model) modifications, network requests, and API accesses, so that we can create identifying signatures of script behaviors. Leveraging these two sets of instrumentation as the basis, we present three works that systematically analyze the current web ecosystem for privacy-abusing practices.

In the first work, we present the technical details of our first set of instrumentation that implements taint tracking for JavaScript, which we refer to as Mystique. Having this tool at our disposal, we then turn our attention to understanding whether browser extensions, given their privileged APIs that have access to their users' browsing activities, exfiltrate privacy-sensitive data about their users. Using Mystique, we analyzed 178,893 Chrome extensions as well as 2,790 Opera extensions, and flagged 3,868 (2.13%) of them to be potentially leaking privacy-sensitive data. The top 10 of the

flagged Chrome extensions that we confirmed to be leaking privacy-sensitive data have more than 60 million users combined, demonstrating the extent to which user privacy is being impacted by browser extensions.

In the second work, we detail the second set of our instrumentation to the Chromium browser, with the goal of generating identifying signatures of script behaviors that we can then use to find similarly-behaving scripts in the current web. This is motivated by the observation that privacy-enhancing filter lists, such as EasyList/EasyPrivacy (EL/EP) [Ele], rely heavily on manual crowd-sourced efforts and are therefore prone to errors and omissions. More importantly, this also makes them vulnerable to trivial evasions, e.g., moving blocked code to new domains/URLs, inlining on the first-party website, and bundling blocked code with other desirable, benign functionalities. We apply our system to analyze JavaScript code found on the Alexa top 100K websites, building 2,001 identifying signatures from existing EL/EP-blocked scripts, and find an additional 3,589 unique scripts exhibiting similar privacy-harming behaviors but are not blocked by EL/EP. We provide detailed analysis of our findings, including a taxonomy of the techniques used for potential filter list evasions along with case studies.

In our third work, we apply our Mystique instrumentation to study a web tracking technique that has seen increased usage as major browsers begin to adopt stricter policies of blocking third-party cookies from known trackers. Instead of using third-party cookies, this new tracking technique uses *first-party* cookies that are set and read by third-party JavaScript code, which we refer to as *external cookies*. In this work we further enhanced the capabilities of Mystique by implementing precise byte-level taint tracking for the string type. Our results show that a majority (97.72%) of Alexa top 10K websites have external cookies. We further propose novel techniques to automatically detect if a cookie contains web tracking IDs, and find that out of the 26,632 unique external cookies found on the Alexa top 10K, 4,212 of these can be used to track users. Finally, we develop a heuristic to understand the relationships between setter and receivers of external cookies, shedding light on the complex network of entities involved in this web tracking practice.

Privacy Implications of Emerging Web Technologies

by
Quan Chen

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2021

APPROVED BY:

William Enck	Brad Reaves
Guoliang Jin	Alexandros Kapravelos Chair of Advisory Committee

DEDICATION

To my parents, for their love and support.
And to Maomi, for eighteen plus wonderful years.

BIOGRAPHY

The author hails from the picturesque seaside city of Xiamen, China, where he lived and studied prior to coming to NC State University for his Ph.D. program (naturally, he went to Xiamen University for his undergraduate education in Computer Science). He has found two quotes which have come to shape his outlook towards his scientific endeavours, as well as life in general. The first of these two quotes is: “The important thing is not to stop questioning”, by Albert Einstein. The second is: “Ah, but a man’s reach should exceed his grasp, Or what’s a heaven for?”, by Robert Browning.

ACKNOWLEDGEMENTS

I would like to thank my academic advisors here at NC State University, both past and present: Dr. Alexandros Kapravelos, Dr. Peng Ning, and Dr. Douglas Reeves, for helping me through the years in the Ph.D. program. Many thanks also go to my collaborators, especially Panagiotis Ilia, Michalis Polychronakis, Peter Snyder, and Ahmed M. Azab, for working together on novel and impactful research projects, and for mentoring me and preventing me from going down rabbit holes. Last but not least, I would also thank my committee members Dr. William Enck, Dr. Brad Reaves, Dr. Guoliang Jin and Dr. Steffen Heber for suggesting improvements to the presentation of this thesis and for attending my defense.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

1.1 The Web Ecosystem and Its Privacy Implications

A powerful enabling feature of the web is its capability for websites to refer to or otherwise include arbitrary third-party resources. Hyperlinks on a website provide users pointers to relevant information that might be hosted on a completely different domain, while special HTML tags such as `<img>` and `<iframe>` allow third-party contents to be embedded and displayed as though it's part of the including website. Despite the fact that the HTTP protocol underlying the web is stateless, HTTP cookies allow websites to link together requests that originate from the same user. Although a necessary requirement for stateful web applications such as e-commerce that we take for granted today, HTTP cookies, in the form of *third-party cookies* which directly results from the web's capability to include third-party resources, have also enabled third-party tracking of the web user's online activities by advertising networks.

Furthermore, in order to support dynamic user-interactive pages, client-side scripting languages such as JavaScript are in almost universal use: according to W3Techs [Jav], JavaScript is used on 97.1% of all websites, and these scripts, like other contents in a webpage, are often hosted on and/or provided by multiple third parties and execute in the same context as the visited page, granting them access to sensitive information such as the browsing habits and keyboard inputs from the users. They also have access to the persistent per-website storage facilities (e.g., cookies and the `localStorage` API), and can send/receive network requests (e.g., using the `XMLHttpRequest` API,

or by injecting an element into the web page with the element's `src` attribute encoding the request). Recently JavaScript has also been used by web tracking providers to calculate unique fingerprints of the users' browser, based on device-specific properties such as supported fonts that is accessible through JavaScript APIs. As the complexity of web pages increase, in an effort to support an ever-increasing list of desirable features, so does the complexity of the relationships among the entities that provide the resources necessary for implementing these features.

Finally, adding to the above complexity, modern browsers also support extension mechanisms that allow users to customize or enhance their web browsing experiences. This is achieved by installing *browser extensions*, which are essentially small programs written in HTML, CSS and JavaScript that are run inside the browser (e.g., Google Chrome Extensions for the Chrome/Chromium browser, and Add-ons for the Firefox browser). Extensions implement their functionalities largely by injecting JavaScript code to be executed in the context of a visited page, as well as having a persistent background page that can react to specific page events. Unlike the aforementioned JavaScript code that is included as part of a web page (regardless of whether they are provided on a website by third parties or not), extension JavaScript has additional access to privileged extension APIs. Such APIs allow extensions to, among others, register callback handlers for privacy-sensitive events such as page navigation, as well as to directly query for the user's browsing history information.

Realizing the potential for privacy abuse, all modern browsers implement mitigation mechanisms to reduce the impact on users' privacy. For example: 1) the same-origin policy (SOP) forbids scripts contained in a web page to access data from another page if the two have different origins (e.g., different domains); 2) to prevent tracking based on third-party cookies, all major browsers offer a way for their users to block third-party cookies. The Firefox browser has also gone one step further with an experimental policy that automatically blocks third-party cookies from known trackers [Fira], and the Safari browser has introduced a built-in privacy protection mechanism known as Intelligent Tracking Prevention (ITP) [Web20] that relies on machine learning classifiers running directly on users' devices, ; 3) in regard to browser extensions, the intent use privileged extension APIs must be declared in the extension's manifest file. However, given the modern web's complex nature, existing efforts are either inadequate in their effectiveness, or lacking in a more global view of the extent of current privacy abuses. In this thesis we present our works that aim to bridge this gap in order to better protect the web user's privacy against increasingly prying eyes.

1.2 Thesis Statement

In this dissertation, we attempt to understand and measure the prevalence of privacy-abusing behaviors in the current web ecosystem. Specifically, given the pervasiveness of JavaScript code in the current web, as we have mentioned previously, we focus on designing systems that would allow us detailed insights into the behaviors of JavaScript loaded from diverse sources that users would

encounter when they visit the web. We then use such systems to conduct large-scale analysis of JavaScript behaviors in the wild as they relate to user privacy, and distill valuable insights. As such, we propose our thesis statement as follows:

Instrumentation of the browser's JavaScript execution environment allows capturing of detailed runtime script behaviors, and when combined with careful analysis, facilitates the understanding of privacy-abusing practices in the current web ecosystem.

Furthermore, as we will show in the rest of this thesis, besides detecting and measuring privacy-abusing behaviors, such systems can also inform and guide efforts on the defensive side to improve user privacy, e.g., by serving as a component of defense mechanisms that run in the users' browser to protect their privacy in real-time as they visit the web, or as a triage system that monitors behaviors of browser extensions hosted on online extension stores such as the Chrome Web Store.

1.3 Thesis Contributions

Our efforts at instrumenting the browser's JavaScript runtime are two-fold:

(i) We implement dynamic taint tracking for the JavaScript language, in the context of the Google Chrome browser. Our implemented taint tracking engine is generic - in that it it able to track and propagate taint across all JavaScript types, instead of being limited to specific types such as the string type ([Lek13] is an example of the latter). Our taint engine is also able to propagate taint across control flow dependencies. More technical details on the implementation of the proposed taint engine, *Mystique*, are presented in our first work, in Chapter 3, where we leverage it to detect leakage of privacy-sensitive information by browser extensions.

(ii) Apart from the above instrumentation that allows dynamic data-flow analysis on the behaviors of JavaScript code, in this thesis we also leverage another instrumentation, PageGraph [Bra20a], that records runtime JavaScript actions into a graph structure which extends the Document Object Model (DOM) tree maintained internally by browsers when rendering a web page. PageGraph is an open-source instrumentation developed by Brave Software [Bra20b]. The runtime actions recorded by PageGraph include events such as DOM interactions (e.g., element creation/deletion), network requests, and web API usage. We present in Chapter 4 how we use the actions recorded by PageGraph to generate identifying signatures of known

privacy-abusing JavaScript code, as well as how we apply the generated signatures to detect instances of filter list evasion.

By leveraging the above instrumentations to the JavaScript execution environment, in this thesis we make the following contributions:

- *Mystique: Uncovering Information Leakage from Browser Extensions* (CCS'18): In this work we perform large-scale analysis of browser extensions for their privacy practices. We begin by implementing dynamic taint tracking capabilities for JavaScript. To do this, we leverage the fact that JavaScript is a dynamic language that is compiled/interpreted just-in-time (JIT) at execution and therefore all the JavaScript code that is to be executed in the browser will be available to its JavaScript JIT compiler/interpreter. Specifically, we realize dynamic taint propagation by using information gained from static analysis of the JavaScript source code being executed. We modified the V8 JavaScript engine, the WebKit/Blink browser engine, and the extension subsystem of the Chromium browser to be taint-aware. The taint-enhanced Chromium browser is then used to analyze 178,893 Chrome extensions as well as 2,790 Opera extensions.

  Highlights:

  1. Our analysis flagged 3,809 (2.13%) Chrome extensions and 59 (2.11%) Opera extensions as potentially leaking privacy-sensitive information.
  2. Manual verification of a random sample of the flagged extensions confirmed 77.70% true positive extensions and 8.82% false positive extensions. The remaining 13.48% are undecided and needs further manual scrutiny.
  3. The top 10 of our true positive extensions have more than 60 million users combined, showing the extent to which browser extensions impact user privacy.
  4. Identified cases that would have been missed by similar previous works.

- *Detecting Filter List Evasion With Event-Loop-Turn Granularity JavaScript Signatures* (S&P'21): We leverage an open-source instrumentation, PageGraph [Bra20a], that records JavaScript actions during runtime into a graph. Our work is motivated by the observation that many of the current privacy-enhancing filter lists, such as EasyList/EasyPrivacy (EL/EP) [Ele] on which the popular AdBlock Plus [Adb] is based, rely heavily on manual crowd-sourced efforts and therefore is slow to update and easily circumventable. To address this challenge, we generate identifying signatures of script actions taken by existing, known privacy-abusing scripts, and use these signatures to detect scripts exhibiting similar behaviors but are evading filter list blocking. We crawled the Alexa top 100K websites with our instrumented browser, and used our signature-based detection scheme to analyze the results.

Highlights:

1. From the data generated by our crawl of the Alexa top 100K websites, our proposed signature generation algorithm produced 2,001 signatures that correspond to 11,212 *unique* scripts that are blocked by EL/EP (i.e., EL/EP considers these scripts to be privacy-harming).
2. Using these signatures of known-harmful scripts, we found an additional 3,589 *unique* scripts exhibiting the same behaviors that EL/EP considers to be privacy-harming, but are evading blocking by EL/EP.
3. We provide detailed analysis of the results, including relationships in popularity between domains hosting the EL/EP-blocked scripts versus the evaded scripts, as well as detailed taxonomy on the methods employed for the evasion.

- *Cookie Swap Party: Abusing First-Party Cookies for Web Tracking* (WWW'21): We apply our dynamic taint analysis engine for the Chromium browser, which we proposed in *Mystique*, to measuring a different type of potential privacy abuse: web tracking that leverages the ability to execute third-party JavaScript code with the full privilege of the visited first-party web page. While third-party cookies have traditionally facilitated web tracking, due to growing awareness of users to protect their online privacy, along with ever stricter policies by browser vendors to block third-party cookies from known trackers, web trackers have increasingly been turning to using *first-party* cookies that are set by third-party JavaScript code. To investigate such practices, we further extended Mystique's taint tracking capabilities, and used it to crawl Alexa top 10K websites. We propose novel methodologies to automatically detect tracking identifiers in cookies and shed light on the relationships among entities that are involved in setting/receiving the tracking cookies.

  Highlights:

  1. We further extended Mystique to implement precise byte-level taint propagation for strings, so that *first-party* cookies that are set by third-party JavaScript code, which we refer to as *external cookies*, can be accurately tainted and their data flows tracked.
  2. Data from crawling the Alexa top 10K websites suggest that 9,772 (97.72%) that have external cookies set on them. A total of 26,632 *unique* external cookies were detected, out of which 13,323 are non-session cookies, meaning that they will persist on the user's device even after the browsing session ends and can potentially be used for web tracking.
  3. We propose a method to automatically identify external cookies that contain unique user identifiers, and use it to give a detailed taxonomy of the 13,323 non-session external

cookies that we found. In particular, 4,212 (31.61%) of them were flagged as containing tracking IDs.

4. We also propose a heuristic to correlate the domains receiving information derived from the identified UID-containing cookies, and the domains that originally set the cookies. Based on this, we give a detailed analysis of the domain relationships among entities involved in setting/receiving external cookies.

1.4 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 surveys previous research works related to the topics covered in this thesis.

Chapter 3 presents Mystique, our taint analysis engine for the Chromium browser, how we use it to uncover leakage of privacy-sensitive information by browser extensions, detailed analysis of the results, and representative case studies.

Chapter 4 presents how we use PageGraph to generate unique behavioral signatures of known privacy-harming scripts and apply the generated signatures to detect similar scripts that are evading filter list blocking.

Chapter 5 takes Mystique, our taint analysis engine for Chromium, and applies it to another setting where we shed light on the current practice of web tracking via *first-party* cookies that are set by third-party JavaScript code. We also present novel methodologies that automatically detects tracking IDs in cookies as well as uncovering the relationships among entities engaged in using this tracking method.

Chapter 6 outlines directions for future work, based on what we present in this thesis, and then concludes.

CHAPTER

2

RELATED WORK

The research topics we cover in this thesis, namely instrumentation of the browser, and measuring and understanding the behaviors of JavaScript code as they relate to user privacy, have been studied in previous research works. In this chapter we discuss these research efforts as they relate to each of the works that we present in this thesis, and how our works differ from previous research, highlighting the contributions made in this thesis.

2.1 Analyzing Information Leakage from Browser Extensions

Our work presented in Chapter 3 deals mainly with (i) instrumenting the browser's JavaScript execution environment to implement dynamic taint tracking capabilities, and (ii) large-scale analysis of browser extensions for their privacy-related behaviors. We discuss below the related works in these two broad areas.

2.1.1 Information flows in the web context

A number of previous works have investigated applying taint analysis to the web security context. Specifically, Lekies et al. [Lek13], Stock et al. [Sto14], and Melicher et al. [Mel18] detect DOM-based cross-site scripting (XSS) vulnerabilities by augmenting both the V8 JavaScript engine and WebKit to make Chrome taint-aware. As mentioned, their system only handles direct taint propagation between strings, and as such is not adequate in detecting privacy-leaking extensions. Earlier

works [Vog07; Dha09] have implemented taint tracking for the Firefox browser by extending the SpiderMonkey [Moz18] JavaScript engine. Although these were able to cover all available object types in JavaScript as well as control-flow dependencies, they relied on instrumenting all (bytecode) data flow operations emitted by SpiderMonkey. This approach is not suitable for our purposes, given the complex nature of the V8 engine as mentioned in Section 3.3.

The taint analysis technique that Mystique leverages fits in a broader category of works that deal with information flows in the web security context. Jang et al. [Jan10] use source-level rewriting to track information flows in JavaScript web applications. However, they pointed out that their approach did not cover browser built-in APIs, nor did they handle data flows through the DOM, both of which are handled by Mystique. Also, their approach introduces source-level changes (e.g., script size) that might alter program behaviors. Similar rewriting-based approach was also used by Chudnov et al. [Chu15], who propose an information flow control (IFC) monitor, but substantial additional work is needed for their approach to work with existing JavaScript applications. Another work [DG12] implements IFC for Firefox to enforce confidentiality policies between web scripts and browser APIs. Finally, Bauer et al. [Bau15] offer an approach to enforce coarse-grained information flow policies among entities in the browser (e.g., DOM elements, events, and extensions), but it lacked Mystique's ability to track the detailed data flows within JavaScript applications (they treated the V8 JavaScript engine as a black box in their analysis).

As a complement to dynamic analysis techniques, previous works (e.g., [Sax10; Ban10; Guh11]) applied static program analysis techniques in the web context to secure information flows, and/or to provide other security properties (e.g., Saxena et al. [Sax10] implemented symbolic execution for JavaScript to automatically find vulnerabilities).

2.1.2 Detecting privacy-leaking extensions

Previous research [Li07; Giu12; Sta17; Wei17] also dealt with the privacy implications of browser extensions. In particular, Starov et al. [Sta17] and Weissbacher et al. [Wei17] are closely related to Mystique. Mystique differentiates in methodology from these prior works. As we mentioned the approaches adopted by Starov et al. [Sta17] and Weissbacher et al. [Wei17] rely on monitoring and analyzing network traffic, and we already compared Mystique with them in regard to methodologies in Section 3.5.3.3. Note that unlike Mystique, Starov et al. [Sta17] and Weissbacher et al. [Wei17] only analyzed the top 10K most popular extensions on the Chrome Web Store.

2.1.3 Large-scale studies of browser extensions

Apart from some of the works mentioned above (e.g., [Ban10; Sta17]) and Mystique, researchers have also proposed a number of dynamic analysis frameworks aimed at automatically analyzing large numbers of browser extensions for malicious behaviors. Kapravelos [Kap14] actively elicit

malicious behaviors from browser extensions by developing HoneyPages, which we incorporated into our evaluation of Mystique. Another work [Wan12] uses an instrumented Firefox browser to automatically analyze extensions for dangerous behaviors.

2.2 Blocking Privacy-Harming Web Trackers

In Chapter 4, we analyze the current web for attempts by user tracking providers to evade filter list blocking. Filter lists, such as EasyList/EasyPrivacy (EL/EP) [Ele], are essentially crowd-sourced blacklists that block privacy-harming scripts from executing in the user's browser, and as such, there are financial incentives for tracking providers to evade them, especially since their bottom lines are affected by such enterprises. We survey previous attempts at blocking web trackers as well as the more general areas of browser instrumentation and code similarity, and how our work in Chapter 4 compares and constrats with them.

2.2.1 Blocking trackers

The current line of defense that most users have against web tracking is via browser extensions [Adb; Hil; Gho; Disa]. These extensions work by leveraging hand-crafted filter lists of HTML elements and URLs that are connected with advertisers and trackers [Ele]. There are also dynamic approaches for blocking, like Privacy Badger from EFF [EFF], which tracks images, scripts and advertising from third parties in the visited pages and blocks them if it detects any tracking techniques. The future of browser extensions as web tracking prevention tools is currently threaten by the transition to the newest version of WebExtensions Manifest v3 [Blo], which limits the capabilities of dynamically making decisions to block content.

Previous research has also focused on automated approaches to improve content blocking. Gugelmann et al., built a classifier for identifying privacy-intrusive Web services in HTTP traffic [Gug15]. NoMoAds leverages the network interface of a mobile device to extract features and uses a classifier to detect ad requests [Shu18].

2.2.2 Instrumenting the browser

Extracting information from the browser is mandatory step into understanding web tracking. Previous approaches, like OpenWPM, have focused on leveraging a browser extension to monitor the events of a visited page [Eng16]. In-band approaches like OpenWPM inject JS into the visited page in order to capture all events, which can affect their accuracy, as they are running at the same level as the monitored code. Recently, we have observed a shift in pushing more browser instrumentation out-of-band (in-browser) [Iqb20; Li18; Jue19a]. In this chapter, we follow a similar out-of-band

approach, where we build signatures of tracking scripts based on the dynamic code execution by instrumenting Blink and V8 in the Chromium browser.

2.2.3 Code Similarity

Code similarity is a well-established research field in the software engineering community [Roy07]. From a security perspective, finding code similarities with malicious samples has been explored in the past. Revolver [Kap13] performed large-scale clustering of JavaScript samples in order to find similarities in cases where the classification is different, automatically detecting this way evasive samples.

Ikram et al. [Ikr17], suggested the use of features from JavaScript programs using syntactic and structural models to build a classifier that detects scripts with tracking [Ikr17]. Instead of relying on syntactic and structural similarity, in our work we identify tracking scripts based on the tracking properties of their execution in the browser, defeating this way techniques like obfuscation [Sko19] and manipulation of ASTs [Fas19].

2.2.4 Other Content Blocking Strategies

Another approach to block content is via perceptual detection of advertisements [Sto17; Sen]. This approach is based on the identifying advertisements based on known visual patterns that they have, such as the AdChoices standard [(DA09]. Although this is an exciting new avenue of blocking content on the web, there is already work that aims to create adversarial attacks against perceptual ad blocking [Tra19].

2.3 Cookie-Based Web Tracking

We survey in Chapter 5 the current web for instances where first-party cookies are set by third-party JavaScript code that runs in the first-party context, and the privacy abuses that arise from this capability. This work is thus related to previous research on the topics of (i) measurement of, and defense against, third-party web tracking, and (ii) JavaScript sandboxing.

Third-party tracking. The web's power comes from its ability to link to third-party contents, but this also enables third-party tracking. Krishnamurthy et al. [Kri09] examined the prevalence of third-party tracking by carrying out a longitudinal study. Mayer et al. [May12] proposed a web measurement platform, FourthParty, to survey the policy and technology issues involved in third-party tracking. Englehardt et al. [Eng16] conducted a large-scale measurement of Alexa top one million websites. Other than pure measurements, previous work also proposed defenses against third-party tracking [Pan15; Roe12]. Pan et al. [Pan15] proposed an anti-tracking browser that isolates unique identifiers into different browser principals so that the accuracy of those identifiers

is significantly reduced. Roesner et al. [Roe12] explored various techniques employed by trackers. Although the authors acknowledged that external cookies can be used to track repeat visitors to the same website (i.e., case of web analytics), they underestimated their potential in bypassing third-party cookie blocking and facilitating cross-domain exchange of tracking IDs, a capability which we examine at length in this paper. Franken et al. [Fra18] evaluates third-party cookie policies on the current web. A recent work, by Fouad et al. [Fou18] explored different techniques employed by trackers, and among them found that the values of first-party cookies can be leaked to third parties. Our work adds another perspective to this ongoing line of research by focusing on first-party cookies that are set by third-party code, and explore how they relate to third-party tracking on the web.

Lastly, Sanchez-Rola et al. [SR21] is a closely related work that is concurrent to ours, which studies "ghost cookies" that is conceptually the same as external cookies. In that work the authors propose the notion of cookie trees to systematically explore the relationships among entities engaged in web tracking, but they lack the detailed JavaScript data flow information that is afforded by an analysis system such as Mystique, which we leverage in this work. Combining the strengths of their methodology and ours is a promising direction for further bettering the understanding of web tracking.

JavaScript sandboxing: The abuses of first-party cookies set by third-party JS code, which we focus on in this paper, can be eliminated or otherwise mitigated, if browsers implement sandboxing for scripts loaded from different origins. The Chrome browser already enforces a form of sandboxing where the content scripts injected by browser extensions cannot access any variables or functions created by the normal JavaScript code running on the page, or by other content scripts [Con]. Previous research in sandboxing JavaScript by Agten et al. [Agt12] proposes an approach to securely integrate third-party JavaScript code that achieves complete mediation. They improved on existing works that rely on instrumenting untrusted code on the server side to a safe subset of JavaScript (e.g., [Pol11]), or implementing a reference monitor inside the browser (e.g., [Mey10; VA11]). Adapting these research efforts to mitigate the privacy impact of first-party cookie tracking is a direction for future work.

CHAPTER

3

MYSTIQUE: UNCOVERING INFORMATION LEAKAGE FROM BROWSER EXTENSIONS

Browser extensions are small JavaScript, CSS and HTML programs that run inside the browser with special privileges. These programs, often written by third parties, operate on the pages that the browser is visiting, giving the user a programmatic way to configure the browser. The privacy implications that arise by allowing privileged third-party code to execute inside the users' browser are not well understood.

In this chapter, we develop a taint analysis framework for browser extensions and use it to perform a large scale study of extensions in regard to their privacy practices. We first present a hybrid approach to traditional taint analysis: by leveraging the fact that extension source code is available to the runtime JavaScript engine, we implement as well as enhance traditional taint analysis using information gathered from static data flow and control-flow analysis of the JavaScript source code. Based on this, we further modify the Chromium browser to support taint tracking for extensions. We analyzed 178,893 extensions crawled from the Chrome Web Store between September 2016 and March 2018, as well as a separate set of all available extensions (2,790 in total) for the Opera browser at the time of analysis. From these, our analysis flagged 3,868 (2.13%) extensions as potentially leaking privacy-sensitive information. The top 10 most popular Chrome extensions

that we confirmed to be leaking privacy-sensitive information have more than 60 million users combined. We ran the analysis on a local Kubernetes cluster and were able to finish within a month, demonstrating the feasibility of our approach for large-scale analysis of browser extensions. At the same time, our results emphasize the threat browser extensions pose to user privacy, and the need for countermeasures to safeguard against misbehaving extensions that abuse their privileges.

3.1 Introduction

All popular web browsers today offer extension mechanisms that allow users to customize or enrich their web browsing experiences by modifying the browser's behavior, enhancing its functionalities or integrating with popular web services. To support interaction with the visited web pages, such as modifying their the contents or UI layouts, extension frameworks provide mechanisms to inject custom JavaScript code into a web page and execute in the page's context (e.g., [Con; Moz18]). This capability allows extensions to inject code that retrieves private information from a web page (e.g., page URL, cookies, form inputs, etc). Moreover, browser extensions have access to privileged extension APIs that are out of reach from the normal JavaScript code executing as part of the web pages. For example, Chrome extensions can use the JavaScript extension API `chrome.history` to directly query any past browsing history information [Chrg].

This unique vantage point enjoyed by browser extensions provide them opportunities to gain intimate knowledge of the browsing habits of their users; when this knowledge is abused, it puts users' privacy and personal information at risk. Although the potential for abuse is high, the privacy implications posed by browser extensions have only recently caught the attention of the security community. Several reports and blog posts shed light on the scope of the issue by manually analyzing a few extensions [Wei18; Det18; How18]. Recent works [Sta17] and [Wei17] investigated the privacy practices of browser extensions by analyzing the network traffic generated by extensions. Specifically, [Sta17] applied heuristics to attempt decoding of common encoding/obfuscation techniques, while [Wei17] used machine learning to identify traffic patterns that indicate possible privacy leaks. However, these previous efforts lack either the scale or the depth to examine the full scope of the privacy implications introduced by third-party extensions. For example, the approach proposed by [Sta17] cannot handle customized encoding algorithms or encryption; traffic pattern analysis employed by [Wei17] is prone to evasion whereby attackers mask their network traffic with noise. Indeed, addressing the potential privacy abuse posed by browser extensions requires not only an automatic analysis framework, but also a mechanism that tracks the detailed data flows inside browser extensions.

Requirements: Privacy-intrusive extensions abuse their privileges to leak sensitive information. To avoid detection at the network level, they can arbitrarily generate decoy traffic patterns or otherwise obfuscate/encrypt such information before it is sent on the wire. Thus, to be generic an analysis

framework must be able to label any sensitive information accessed by third-party extensions, as well as to track their usage throughout the lifetime of the extensions. That is, it must implement dynamic taint tracking (e.g., [Enc10]). Such an analysis framework must be able to track data flows across all available JavaScript object types, handle control-flow dependencies, and address any browser-specific data flows paths that are introduced, for example, by the DOM interface or the extension APIs. Additionally, to detect extensions that utilize local storage to persist privacy-sensitive information for later exfiltration only when certain conditions are met (e.g., a threshold number of events), the analysis framework must also identify such extensions and flag them for further scrutiny.

Previous research in the direction of applying dynamic taint tracking to the browser context include [Vog07; Dha09; Dje10], which relied on instrumenting the bytecode instructions emitted by the Firefox browser's JavaScript engine SpiderMonkey [Moz18]. However, similar research efforts is lacking for the Google Chrome browser, which currently holds over 57.64% worldwide market share as of April 2018 [Sta19]. Previous works [Lek13; Mel18] implemented taint tracking for Chrome, albeit only for the string type and did not handle the extension API. There is currently no *complete* dynamic taint tracking implementation for Chrome browser's V8 JavaScript engine [Goo18] that satisfies all the requirements of detecting privacy-intrusive extensions. Furthermore, given the highly optimized nature of the V8 engine, previous approaches that applied to Firefox cannot be straightforwardly adapted to Chrome.

Introducing Mystique: To help bridge this gap, we propose Mystique, an extension analysis framework that serves as the first effort at a complete implementation of dynamic taint tracking for the Google Chrome browser. We augment multiple components of the browser, particularly its V8 JavaScript engine, with taint tracking capabilities. To analyze extensions, Mystique automatically loads them in a monitored environment. Our primary goal in this chapter is to identify third-party browser extensions that leak privacy-sensitive information. To this end, Mystique automatically taints values obtained from extension-accessible data sources that divulge users' private information. To overcome complexities of the JavaScript language as well as the V8 engine, Mystique implements runtime taint propagation by leveraging information obtained from a static data flow and control-flow dependency analysis of the JavaScript source code. Mystique logs extensions that triggered taint sinks with tainted values during the analysis. To aid in post-analysis understanding, Mystique also logs how tainted values are propagated and used by the extension JavaScript code.

We applied Mystique to analyze 178,893 extensions that were crawled from the Chrome Web Store between September 2016 and March 2018, plus a separate set of all available extensions (2,790 in total) for the Opera browser at the time of analysis. Our analysis flagged 3,868 (2.13%) extensions as potentially leaking privacy-sensitive information. The top 10 most popular Chrome extensions that we confirmed to be leaking privacy-sensitive information have more than 60 million users combined, highlighting the privacy threat posed by third-party browser extensions. From

the analysis results, we also uncovered multiple encoding/obfuscation techniques employed by extensions. Our analysis of all 181,683 extensions was run on a local Kubernetes cluster and finished within a month, showing the feasibility of applying Mystique to large-scale analysis of browser extensions. Thus, Mystique can provide "analysis as a service", e.g., integrated as part of a triage system for online extension repositories such as the Chrome Web Store.

We provide a web interface through which users can submit extensions to Mystique and get back the analysis results. More details can be found at `https://mystique.csc.ncsu.edu/`.
Contributions: The major research contributions of this chapter are:

- We propose a novel taint analysis technique that leverages both dynamic taint tracking and static analysis. We provide the first full implementation of hybrid taint tracking for the V8 JavaScript engine, based on techniques that leverage information gathered from static data flow and control-flow dependency analysis.
- We present Mystique, an analysis framework that builds on our hybrid taint tracking implementation to analyze and detect third-party browser extensions that abuse privacy-sensitive information.
- We conducted the first large-scale study of 181,683 third-party browser extensions in regard to their privacy practices.
- We advance the state of the art by uncovering obfuscation schemes used by extensions that escaped the attention of similar previous research efforts.

3.2 Background

In this section, we first give an overview of the Chrome browser's extension framework, and the opportunities that this framework presents for extension authors to obtain and exfiltrate users' privacy-sensitive information. We also provide the relevant technical background of the V8 JavaScript engine that will be used for this work. Note that since Chromium is the open-source version of the Google Chrome browser, in the rest of this chapter the names Chrome and Chromium will be used interchangeably.

3.2.1 Chrome Extension Framework

Chrome supports extensions that modify or enhance the functionality of the browser [Chri]. Extensions are written using JavaScript, HTML and/or CSS, and packaged together with a mandatory manifest file and distributed as a single zip archive. The manifest file describes the extension, and contains, among other parameters, the declared permissions that determine which Chrome extension API calls it can access, as well as what web pages (i.e., URLs) the extension can operate

on. API permissions are typically requested by indicating the API names, while the allowed web pages is known as *host permissions* and are specified using match patterns. Match patterns allow wild-carding, and the special token `<all_urls>` matches any URL. We discuss the relevant aspects of the extensions architecture in the rest of this section. Please refer to Barth et.al [Bar10] for a detailed treatment of the Chrome extension framework.

3.2.1.1 Content Scripts

Extensions can inject and run JavaScript code inside web pages. This injected code is known as content script [Con]. There are two ways to inject content scripts: 1) statically by declaring in the manifest the JavaScript files and their corresponding match patterns, so that Chrome automatically injects the scripts contained in the specified files into every web page whose URL satisfies one of the match patterns, or 2) dynamically by using the so-called programmatic injection method. Either way, the intent to use content scripts need to be declared (explicitly or implicitly) in the manifest file. For the former method, it is obvious that the JavaScript files containing the content scripts must be listed in the manifest file; for the latter method, the extension must: a) declare the "`tabs`" permission (in order to use the `chrome.tabs.executeScript` API), and b) also declare host permissions for the URLs where this injection should be allowed.

Since content scripts are injected into web pages and executed there, they run in the same environment as the normal JavaScript code that is downloaded as part of the web pages. For example, they have the same access to the Document Object Model (DOM) interface, and are therefore able to query it for page details or to make changes. Although content scripts are sandboxed from the normal JavaScript code of the web pages [Con], they nevertheless provide a powerful feature and allow extensions the opportunity to access information that would otherwise not be available to them.

3.2.1.2 Background Pages

In addition to content scripts, extensions can also run JavaScript code in the background page [Bac], which is a special HTML page that is not visible to the user. Unlike content scripts, there can only be one background page per extension, and its purpose is to allow a long-running script to manage states for the lifetime of the extension. Background pages have full access to the Chrome extension API, as long as the appropriate permissions have been declared. For example, they can register callback functions to the `chrome.tabs.onUpdated` event, so as to be notified about the details of any tab update event (e.g., URL of a newly loaded page), provided they have declared the "`tabs`" permission and that the loaded page URL matches one of the host permissions.

3.2.1.3 Message Passing API

Given that content scripts execute inside web pages, there needs to be a way for them to communicate with the rest of the extension. This is achieved by using the message passing API [Chrh], which allows callback functions to be registered that listen for messages being sent on the same channel. A message may be delivered to multiple registered callbacks; inside each of the callback functions further logic can be implemented to decide if actions should be performed for the received message. Both content scripts and background pages can register callbacks, as well as to send messages. The message passing API can also be used to communicate across extensions.

3.2.2 V8 JavaScript Engine

The Chromium browser uses V8 [Goo18] to JIT-compile and execute JavaScript code. One characteristic of the V8 architecture is its use of two separate compilers, i.e., the full compiler (Full-codegen) and the optimizing compiler (Crankshaft). Both of them compile JavaScript to native code. Recently, V8 has moved away from this architecture in favor of an interpreter-compiler pair (i.e., Ignition and TurboFan) [V8 18]. Nevertheless, the basic idea is still the same: begin executing JavaScript code with as little delay as possible, and during execution collect runtime information which will aid the optimizing compiler (Crankshaft or TurboFan) in generating efficient code for portions of the JavaScript source that are frequently executed. Our prototype implementation of Mystique (described in Section 3.4) is based on an earlier version of V8 that uses the Full-codegen/Crankshaft architecture, and we turned off the optimizing compiler (i.e., Crankshaft) so that JavaScript compilation is handled exclusively by Full-codegen. This implementation choice is primarily motivated by simplicity in a proof-of-concept prototype, as well as the intended "analysis as a service" usage scenario of Mystique. Nevertheless, we note that the JavaScript parser (and therefore V8's internal AST representation of JavaScript source code) is shared between Full-codegen and Crankshaft, so the methodology we present in this chapter can also be adapted to Crankshaft. In addition, the code base for the JavaScript parser remains largely stable across V8 versions, and as a result our methodology can also be ported to the latest Ignition/TurboFan architecture.

3.3 Technical Approach

Mystique utilizes dynamic taint analysis to track the runtime data flows inside third-party extensions and identify the ones that leak privacy-sensitive information. Specifically, it extends the Chromium browser and its JavaScript engine (V8) with taint tracking capabilities so that any values that can potentially contain privacy-sensitive information are marked (as tainted). To analyze an extension, Mystique launches an instance of our taint-enhanced Chromium browser with the extension preloaded, inside a monitored environment. The browser is then automatically driven to browse

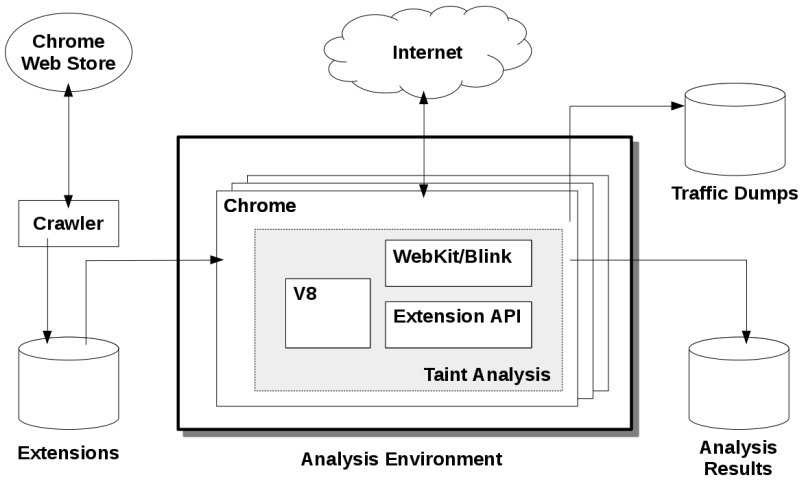

Figure 3.1 Architectural overview of Mystique, showing major components of Chrome that Mystique augments with taint tracking capabilities.

the web. Mystique logs any extension that triggers a taint sink with tainted values. Figure 3.1 shows the architectural overview of Mystique.

Previous research [Lek13; Mel18] that implemented taint analysis for Chromium handled only string-to-string propagation. To detect privacy-leaking extensions, Mystique's taint analysis needs to be generic and therefore should consider all object types and data flow paths available to extension JavaScript code. To achieve this, Mystique needs to overcome the challenges that 1) JavaScript is a complex dynamically-typed language and as a result operations have different meanings that depend on object types; 2) the V8 JavaScript engine is highly optimized so it requires significant engineering efforts to patch all possible data flow paths such that they are taint-aware: for example, V8 can emit native code differently for arithmetic operations (e.g., "+") as either integer or floating-point operations depending on the operand values, despite JavaScript having only one (floating) number type; and furthermore 3) additions to the JavaScript semantics made by both the Chrome extension API [Chrg] and the DOM interface create data flow paths that are not reflected at the JavaScript source level.

To address them, Mystique does not attempt to instrument each individual data-flow operation in order to propagate taint, as was done by Lekies et al. [Lek13] and Melicher et al. [Mel18] for Chromium (and also [Vog07; Dha09; Dje10], which were for Firefox). This design choice primarily follows from the first two challenges, which together imply that it is not feasible to manually identify and patch all the possible data flow operations that can be emitted by the V8 JavaScript engine. On the other hand, since JavaScript is an interpreted language, all the JavaScript source code that is to be executed will become available to the runtime interpreter or JIT compiler at some point. This provides Mystique the opportunity to combine dynamic taint tracking with static source

Table 3.1 Taint sources considered by Mystique.

Category	Taint source	Type	Requires permission?
DOM	`document.URL`	Property evaluation	Content scripts
DOM	`window.location`, `document.location`	Property evaluation	Content scripts
DOM	`document.cookie`	Property evaluation	Content scripts
DOM	`<input type="password">`	DOM query	Content scripts
Extension API	`chrome.tabs`	Event callbacks	"`tabs`"
Extension API	`chrome.webRequest`	Event callbacks	"`webRequest`"
Extension API	`chrome.webNavigation`	Event callbacks	"`webNavigation`"
Extension API	`chrome.history`	Direct query	"`history`"

code analysis. Specifically, Mystique leverages information from static data flow and control-flow dependency analysis at the JavaScript source level to determine which additional objects should also be tainted, given a set of already-tainted objects. For simplicity, Mystique currently employs a flow-insensitive, intra-procedural analysis for this purpose. In the rest of this section, we present the details of Mystique's taint analysis framework.

3.3.1 Sensitive Data Sources

There are two broad categories of sources from which an extension can obtain users' privacy-sensitive information: 1) the DOM interface, and 2) the Chrome extension API. The DOM interface of a web page is accessible to all JavaScript code executing in it, including the content scripts injected by extensions. For example, the DOM property `document.location.href` gives the URL of the page in which it is evaluated. In order for an extension to interact with web pages through the DOM interface, it needs to have the permission to inject content scripts (Section 3.2). On the other hand, the Chrome extension API allows extensions to register callbacks for page loading events (e.g., `chrome.webRequest`), as well as to directly query for privacy-sensitive information (e.g., the `chrome.history` API gives access to the user's browsing history information). These APIs typically require permissions to be declared in the manifest files before they can be used.

Mystique's goal is to track the flow of data containing privacy-sensitive information inside third-party extensions. Therefore, it needs to mark the values obtained from these sources as tainted. Table 3.1 summarizes the taint sources considered by Mystique. We note that this is not intended to be an exhaustive list of all sources from which privacy-sensitive information can be obtained by extensions. Additionally, extensions can also leak privacy-sensitive information without having to access it *themselves*. For example, they can inject into the DOM of the current page an `img` element whose `src` attribute points to a third-party host and does not encode any tainted values, but the `Referer` field in the HTTP header of resulting request would be set by the browser and sent to the third-party host. This way, by reading the `Referer` filed of the incoming request, the third party

```
1  function encode_page_url() {
2    var loc = location.href;
3    var obj = { url: loc, length: loc.length };
4    var length = obj.length;
5
6    var output = "";
7    for (var i = 0; i < length; i++) {
8      var c = obj.url[i];
9      if (c == "a")
10       output += "a";
11     else if (c == "b")
12       output += "b";
13     /* repeated for all valid URL characters */
14   }
15
16   var result = window.btoa(output);
17   return result;
18 }
```

Listing 3.1 Sample JavaScript code, showing control dependency.

can learn of the URL that the user is browsing while the extension that injected the element never accessed such information directly. However, this behavior is easy to detect at the network level since the `Referer` field is sent by the browser in plaintext, and although Mystique's taint tracking currently does not handle this scenario, it is nevertheless easy to modify Chromium so that any DOM element injected by extensions can be differentiated (whether it contains tainted values or not). For example, one way to achieve this is to hook the relevant DOM APIs to mark the elements when they are injected by extension JavaScript, which can itself be differentiated because the associated `Context` objects are different from those of normal website JavaScript (see Section 3.4.2 for explanation regarding the `Context` objects). We consider leakage via the `Referer` field out-of-scope for our current work since Mystique's focus is on tracking the usage of privacy-sensitive information inside extensions.

3.3.2 Taint Propagation with Static Analysis

To monitor the complete data flow of extensions, we not only need to consider all object types available in JavaScript, but also the conversion and interaction among different object types. For example, a `base64` encoding routine might first convert the input to integers, which are then used to index into a table to produce the output string. In this case, if the intermediate integers are not tainted, then we might fail to taint the output string. As mentioned, to be generic and avoid the complexities of JavaScript and the V8 engine, Mystique does not patch individual data flow operations to be taint aware, but rather leverages the fact that V8 has access to all the JavaScript source code that is to be executed, and propagates taint according to information obtained from a flow-insensitive, intra-procedural static analysis of the JavaScript source code.

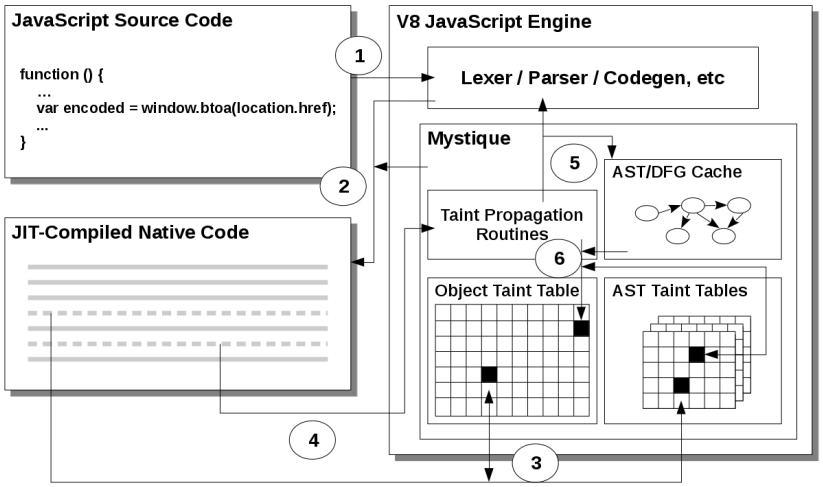

Figure 3.2 Overview of Mystique's approach to taint propagation. (1) and (2): JavaScript source code is compiled by V8 to native code and instrumented by Mystique; (3) During runtime, as expressions get evaluated, the instrumented native code looks up the evaluated values in the object taint table and updates the AST taint table; (4) At taint propagation points, the instrumented native code invokes Mystique to propagate taint; (5) Mystique reuses V8's existing infrastructure to parse for the function's AST and also constructs DFG, caching both; (6) Mystique propagates taint according to the DFG and AST taint table.

Figure 3.2 illustrates an overview of this approach. The basic idea is to use the taint status of concrete runtime objects that the JavaScript code operates on (e.g., strings and numbers) to taint nodes (e.g., variables) in the abstract syntax tree (AST) parsed from the JavaScript source code. Taint propagation then starts from these tainted AST nodes by using a data flow graph (DFG) constructed from the AST. For each AST node that taint propagates to, their corresponding runtime objects are then also tainted. Mystique utilizes V8's existing infrastructure to parse the JavaScript source code for its AST. Mystique also collects control dependency information from the AST, and augments the DFG with this information (Figure 3.2). Listing 3.1 contains an example showing why control dependencies need to be taken into account: assuming `location.href` is tainted (and therefore also `obj.url` and `c`), if control dependencies are not considered, then the `if` branches starting from line 9 would allow an attacker to evade taint analysis (notice that string literals such as `"a"` are not tainted).

Mystique generates DFG in the unit of individual JavaScript functions, since V8 also parses for AST on a function-by-function basis. Note that the JavaScript top-level, or global, scope is treated as an unnamed function by V8. The V8 JavaScript engine does not currently cache but throws away the AST after the JavaScript source has been compiled. However, the JavaScript source code is always kept available, in order to support re-compilation by the optimizing compiler (as well as possible future de-optimization) [Con18]. To minimize performance overhead, Mystique caches the parsed

AST and DFG (Figure 3.2).

To store taint data, two sets of tables are used by Mystique: one for storing taint status of concrete runtime JavaScript objects, and the other for tainted AST nodes (Figure 3.2). For convenience, we refer to the former as *object taint table*, and the latter as *AST taint table*. We need the latter since Mystique propagates taint based on the DFG, which is constructed from the AST. Note that there should be one AST taint table per function *invocation*, since each invocation (even of the same function) can operate on tainted values differently. This allows Mystique to handle taint propagation for recursive calls. During execution of the JavaScript code, the taint status of concrete runtime objects are looked up in the object taint table and used to update the AST taint tables.

Mystique triggers taint propagation for each individual JavaScript function at critical points during its execution. For example, since Mystique's static analysis is intra-procedural, it cannot readily track data flow across function boundaries. Therefore, taint propagation needs to be triggered on function calls so that if taint data should propagate to any of the callee's parameters (or any objects that the callee can access), it will be correctly reflected in the callee. This mechanism shown as the fourth step in Figure 3.2.

In the following, we present the details of Mystique's taint propagation approach.

3.3.2.1 Taint Representation for Runtime Objects

As mentioned, Mystique uses two sets of tables to store taint data. For the runtime JavaScript objects, their taint status is recorded in the object taint table, and we simply implement it as a global hash table that is keyed on the addresses of tainted runtime JavaScript objects. However, this simplistic scheme is complicated by the fact that V8 uses a garbage collector to recycle memory occupied by "dead" objects that are no longer accessible from the JavaScript code. Internally, the heap-allocated runtime JavaScript objects are always given word-aligned addresses, and V8 tags the pointers to heap-allocated objects by setting their lowest bit. This is done so that during garbage collection cycles these pointers can be distinguished from non-tagged (i.e., non-heap-allocated) values. There is only one type of non-tagged values used by V8, i.e., `Smi` for storing small integers (e.g., in the range $[-2^{30}, 2^{30}-1]$) whose values fit in a word (minus the tag bit, since it is always cleared for untagged values). If the integer value is not in the given range, then it is stored in a normal heap-allocated object (i.e., `HeapNumber`, which is also used to store floating-point numbers). Note that no heap memory is allocated to `Smis` - their values are stored directly in the non-tagged word. This arrangement by V8 leads to two problems: *a*) V8's garbage collector might move objects around in memory, so that the addresses stored in the object taint table become outdated, and *b*) if we taint `Smis` by storing their values inside the object taint table, then subsequent accesses to those same integer values will all be considered as tainted, leading to false positives.

Mystique addresses the first problem by making V8's garbage collector aware of the object taint

table, i.e., if tainted objects are either moved in memory or freed, the object taint table is updated to reflect the change. Note that if tainted objects are freed, they can be safely deleted from the object taint table since they are no longer reachable from the JavaScript code. To solve the second problem, Mystique requires that the types of AST nodes that can be included in the DFG (e.g., variable nodes) can never refer to `Smi` values. Specifically, we modify V8's code generation phase so that assignments to such nodes are instrumented with checks to test if the values being assigned are `Smi`s, and if so, replace them with equivalent (i.e., same numerical value) `HeapNumber`s. See Section 3.3.2.3 for AST node types that are included in the DFG. We remark that for function call nodes, since they are "assigned" when the callee returns, we also instrument return statements similarly so that `Smi` values are never returned.

3.3.2.2 Taint Representation for AST Nodes

Mystique stores taint status of AST nodes in the AST taint table, which is implemented as a hash table keyed on the memory addresses of tainted AST nodes (e.g., of a variable node). Notice that since V8 constructs AST at runtime, memory for AST nodes are always dynamically allocated. As mentioned, to account for different invocations of the same function, each invocation should be given its own table. Thus, to check the taint status of an AST node (of a function invocation), two levels of lookups are needed: starting from a global table, using in sequence *a*) the invocation's frame pointer, and *b*) the AST node's memory address as keys. It should be noted that in practice, most JavaScript functions never operate on tainted values, so the majority of lookups will not go beyond the first level. Also, for most of its operations involving the AST taint table, Mystique does not need to traverse the two levels of table lookups for *every* AST node. For example, when propagating taint for a function invocation, it would first get a reference to the second-level table, and all lookups then query this table directly.

Unlike runtime objects, memory allocated to the stack frames and the AST nodes is not managed by V8's garbage collector, so it does not need to be made aware of either of the two levels of tables.

3.3.2.3 DFG Generation

Mystique considers the JavaScript assignment operation as the primary means in which taint propagates, i.e., if the right-hand side expression of an assignment operates on tainted values, then the target on the left-hand side should also be tainted. To account for control dependencies (e.g., introduced by control structures such as `while` statements), Mystique treats their conditional expressions in the same way as the right-hand side of assignments, from which taint should propagate to the left-hand side of every assignment operation contained in that control structure (including nested statements). For control structures having more than one branch (e.g., `if-else` statements), taint propagates from the conditional expression to *all* branches. For convenience we use the term

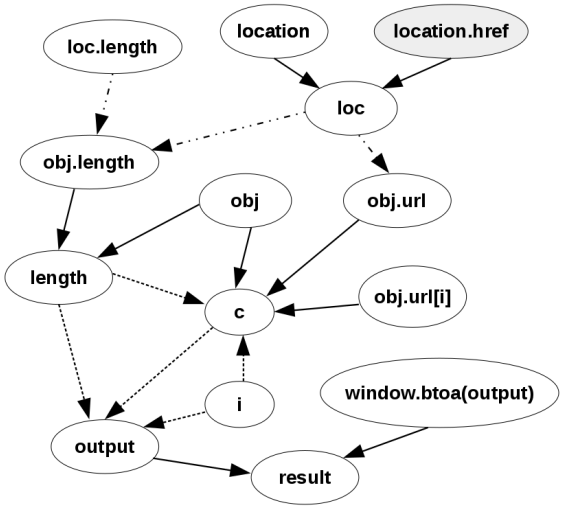

Figure 3.3 Data-flow graph (DFG) generated by Mystique for the code sample in Listing 3.1, augmented with control flow dependencies (dashed lines). Taint source (`location.href`) is highlighted. Also shown are implicit data flows (dashed-and-dotted lines), which are *not* encoded into the DFG.

RHS expressions to refer to both the right-hand side of assignments and conditional expressions of control structures.

RHS expressions are processed recursively, and when an AST node representing 1) variable, 2) object property, or 3) function call is encountered, an edge going from it to the left-hand side target (corresponding to the containing RHS expression) is added to the DFG. Note that array indexing (e.g., `str[i]`) is treated in JavaScript the same as object property access. We remark that if a function call is processed as part of an RHS expression, then Mystique not only generates a DFG edge from the call expression itself (representing the return value), but also recursively processes each argument expression as an RHS expression. Similarly, for object property accesses (e.g., `obj.x`), an edge is generated from the property itself, then the home object expression is also processed recursively. However, if the current RHS expression is an object or array literal, then Mystique does not process the expression. This is due to how Mystique handles tainting for objects and arrays (see Section 3.3.2.5).

Figure 3.3 shows the DFG, augmented with control dependency information, that Mystique generates for the example in Listing 3.1 according to the rules given above. We represent direct data flows as solid lines (e.g., from `location.href` to `loc`, corresponding to the assignment on line 2 in Listing 3.1), and control dependencies as dashed lines (e.g., from `i` to `output`). Note that the initialization of a variable in the `var` statement is treated in V8 as two separate operations: one as variable declaration and the other as an assignment with the initializer expression being the right-hand side. Figure 3.3 also shows implicit data flows (dashed-and-dotted lines), which we

describe next, that are *not* generated as part of the DFG.

3.3.2.4 Implicit Data Flows

Besides explicit assignment operations and control dependencies, data flow paths can also be introduced implicitly in JavaScript. For example, at function calls, the actual parameters are themselves expressions which are evaluated and the resultant values "assigned" to the callee's formal parameters. In this case, the formal parameter should be treated in the same way as the left-hand side of an assignment, and the actual parameter expression the right-hand side. Generally, implicit data flows are introduced whenever the evaluated value of an expression can be "caught" in some manner other than explicit assignment at the source level. For convenience we refer to this type of expressions as *implicit-flow expressions*.

Further examples of this include function return values and literals of compound types (e.g., object/array literals). For the former, if the return value expression is tainted, then the return value needs to be tainted; for the latter, if the expression that specifies the value of a property in the object literal is tainted, its corresponding property value should be tainted as well (Mystique does not taint the object literal itself, see Section 3.3.2.5). Figure 3.3 illustrates an example of implicit flow in the case of object literals (corresponding to line 3 in Listing 3.1), shown as dashed-and-dotted lines pointing from `loc` to `obj.url` and `obj.length`, and from `loc.length` to `obj.length`. Note that in this example, implicit flows are crucial in linking `location.href`, one of the taint sources considered by Mystique, to the final `result` variable (i.e., the return value of the function in Listing 3.1).

Mystique does not generate DFG edges for implicit data flows and their taint propagation is handled separately (discussed in Section 3.3.2.7).

3.3.2.5 Tainting of Object Properties

In contrast to previous approach (e.g., [Vog07]), when tainting a specific property value, Mystique does not propagate taint to its containing object. For instance, if the property `obj.x` should be tainted, we only taint the property's AST node (and its corresponding runtime object), but we do not taint `obj`. This rule also applies to arrays, since array indexing is treated semantically in the same way as property access. However, note that due to how the DFG is constructed (Section 3.3.2.3), if an object is tainted, then all property accesses from that object will propagate taint from it (e.g., if `str` is tainted and `str.length` is assigned to a variable, then that variable will be tainted as well). Similarly, property accesses of a tainted object, if they constitute parts of an implicit-flow expression (Section 3.3.2.4), would also propagate taint to the evaluated value of that expression (Section 3.3.2.7 details how Mystique propagates taint for implicit-flow expressions).

3.3.2.6 Updating the AST Taint Tables

To accurately update the AST taint tables during runtime, we instrument the code emitted by V8 for the AST node types that can be included in the DFG. This instrumentation makes sure that as the expression represented by an AST node gets evaluated at runtime, the taint status of the resultant value is looked up in the object taint table and, if tainted, the AST node must also be tainted (i.e., it is recorded in the AST taint table). On the other hand, if the evaluated value is not tainted but the corresponding AST node is tainted, then the AST node should be untainted (this can potentially be the case for AST nodes belonging to loop statements). Doing so prevents overtainting and ultimately false positives. Note that if the value that an AST node refers to is replaced by a `HeapNumber` (as described in Section 3.3.2.1), this instrumentation will update the AST taint table using the replaced (i.e., `HeapNumber`) value.

3.3.2.7 Taint Propagation Points

Since Mystique adopts an intra-procedural static analysis, data flows across function boundaries are not reflected in the DFG. To solve this problem, Mystique requires that taint propagation be triggered on function calls and returns in order to update taint data for the callee and caller, respectively. Additionally, to more accurately capture the taint data flows in JavaScript, two more propagation points need be included in Mystique's analysis.

First, as mentioned in Section 3.3.2.6, to precisely model taint data flows in loop statements, Mystique allows untainting AST nodes when they no longer refer to tainted runtime objects. However, this can introduce inaccuracies into the analysis. For example, if a loop statement operated on tainted values during its execution, but on its last iteration it did not, then on exit from the loop statement none of the AST nodes belonging to it will be tainted. To address this scenario, Mystique requires that taint propagation be triggered at the end of a basic block.

Second, taint propagation should also be triggered when encountering implicit data flows (Section 3.3.2.4), so that if taint should propagate *to* the implicit-flow expressions, it is correctly reflected. Then, to handle taint propagation *from* implicit-flow expressions, Mystique treats them as RHS expressions: for all of the containing AST nodes from which a DFG edge should otherwise be generated (as described in Section 3.3.2.3), if any of them are tainted, then the evaluated values of the implicit-flow expressions need to be tainted as well (i.e., recorded in the object taint table).

3.3.2.8 Handling `eval` of Tainted Strings

Mystique handles `eval` of tainted strings in a manner similar to [Vog07]: if a JavaScript function is compiled from a tainted string, Mystique then taints all left-hand side targets of assignment expressions in that function; for implicit-flow expressions, Mystique always taints their evaluated values.

3.3.3 Additional Data Flow Paths

Besides the data flow paths mentioned in Section 3.3.2, the Chromium browser also augments the semantics of JavaScript to implement both the DOM interface and its extension API, which creates additional data flow paths that are accessible to extensions. In the following we detail how Mystique propagates taint across these paths. Note that the data flow paths mentioned in this section are those that are currently considered by Mystique's analysis, and should not be considered as an exhaustive list of data flow paths accessible to extensions through the DOM or the extension API.

3.3.3.1 The DOM Interface

The DOM interface is not implemented by V8. Instead, it is implemented as an add-on to JavaScript in WebKit/Blink [Chra]. Since the DOM implementation is external to V8, JavaScript values written to DOM need to be converted to their corresponding representations in WebKit/Blink (and vice versa when values are read from the DOM). Extensions that have declared the permission to inject content scripts can use them to interact with the DOM interface of web pages. For example, they can invoke the `setAttribute` method on an element in the DOM, and later read back that attribute's value. Given the value conversion between V8 and WebKit/Blink, the read-back value may be a separate copy of the original and therefore not tainted. To solve this problem, Mystique also taints in WebKit/Blink any values that are tainted in V8. When such tainted values are read by JavaScript, Mystique ensures that they are tainted in V8.

We remark that as an implementation choice, our current Mystique prototype (see Section 3.4) does not cover string manipulations (e.g., concatenation) that are *internal* to WebKit/Blink, nor do we handle the HTML tokenizer that WebKit/Blink invokes to parse HTML content written to the DOM via JavaScript (e.g., by writing to `innerHTML`). We stress that these are particular implementation choices and not a fundamental limitation of Mystique's methodology. Previous work [Lek13] has already made the internal operations of WebKit/Blink taint-aware.

3.3.3.2 Chrome Extension API

As mentioned in Section 3.2, the Chromium browser provides an API for message passing between content scripts, which execute inside of web pages, and the rest of the extension (e.g., the background page) [Chrh]. This API allows JavaScript objects to be passed through it. Internally, the objects being sent are first serialized (using `JSON.stringify`) and then given to the (C++) IPC implementation to be delivered to the receiving end of the message pipe, where the original objects are then parsed back. However, if the original objects contain tainted values, then the objects parsed back need to reflect the same taint status. To minimize changes to the code base, we chose not to make the IPC implementation taint-aware. Instead, for each JavaScript object sent through the message passing API, Mystique visits each of its properties recursively and constructs a "meta-object" that describes

its taint status. This meta-object is then also stringified, and the result is then prepended to the stringified output of the original JavaScript object. On the receiving end, the meta-object is used to reconstruct the taint status of the parsed back object.

Another data flow path introduced by the extension API is the `executeScript` method in `chrome.tabs` [Chre], which as mentioned allows programmatic injection of strings that execute as content scripts in web pages. If the injected scripts are tainted, we treat them in the same way as `eval` of tainted strings (see Section 3.3.2.8).

Finally, Chromium provides an API (`chrome.storage`) that allows extensions to serialize JavaScript objects to storage [Chrd]. Similar to the messaging API, serialization is done using `JSON.stringify`. Therefore, Mystique also handles taint propagation for it by use of meta-objects that are serialized together with the original objects.

3.3.4 Taint Sinks

To detect extensions that abuse their privileges to gather and exfiltrate users' privacy-sensitive information, Mystique currently considers the following as taint sinks in its analysis:

- `XMLHttpRequest`: An alert is raised if tainted values form any part of the request URL parameter or the request body.
- `WebSocket`: Similar to `XMLHttpRequest`, an alert is raised if any tainted values are sent via this interface.
- `chrome.storage`: Raise an alert if tainted values are given to the `chrome.storage` API for persistence.
- For the DOM elements injected by extensions, raise an alert if their `src` attributes contain tainted values.

We consider the `chrome.storage` API to be a taint sink, for the case where an extension stores privacy-sensitive information first and only exfiltrates them later in bulk (e.g., only after collecting a threshold number of URL visits). Note that as with all dynamic analysis systems, Mystique might not be able to generate the threshold number of events at runtime in order to trigger the leaking behaviors for an extension. In such cases, Mystique will still be able to flag the extension if it uses the `chrome.storage` API to persist data across runs. Previous approaches (e.g., [Sta17; Wei17]) that rely solely on observing the network traffic generated by an extension would fail to detect such cases.

DOM elements injected by extensions are also considered as taint sinks, since for such elements the browser will try to fetch their content from the URL specified in the `src` attribute. Thus, ex-

tensions can leak sensitive information by, for example, encoding it as part of the `src` attribute URL.

3.3.5 Taint Propagation Logs and Sink Report

To keep track of tainted data flows inside extensions, each step of taint propagation needs to be logged. In its basic form, for each tainted JavaScript object, Mystique logs the JavaScript object from which taint propagated to it, along with the JavaScript function and source code position in that function where propagation occurred. For JavaScript objects that are taint sources, whose taint did not propagate from another object (e.g., `document.location.href`), Mystique logs them in a separate table along with the JavaScript function and position inside that function where they are accessed.

In cases where taint propagation is due an meta-object (see Section 3.3.3.2), Mystique first dumps on the filesystem the propagation logs of all the tainted JavaScript values that the meta-object describes. Then, the filesystem paths of the propagation logs are encoded inside the meta-object. These filesystem paths would then be used to indicate the previous step in taint propagation once the original object is parsed back.

For JavaScript functions that are compiled from tainted strings (due to either `eval` or the extension API's `executeScript` method), as mentioned Mystique taints all left-hand side targets of assignments as well as the evaluated values of implicit-flow expressions. For these values, the previous step in taint propagation would be the tainted string from which the function was compiled.

When tainted values reach taint sinks, Mystique logs the event by recording 1) the tainted values that triggered the taint sink and their propagation logs, 2) the current JavaScript stack trace, and 3) the source code of JavaScript functions along the propagation paths (including the functions that accessed taint sources).

3.4 Implementation

We implemented a prototype of Mystique for the Chromium browser. Apart from our description in Section 3.3, we detail in the rest of this section the additional changes that our prototype implementation added to V8 and WebKit/Blink.

3.4.1 Mapping AST Nodes to JavaScript Objects

Given Mystique's approach to taint propagation detailed in Section 3.3, it is necessary to know, for each AST node that taint propagates to, what JavaScript object they referred to *at the instant* when their values were evaluated during runtime. This is needed so that the object taint table can be updated correctly. To obtain this mapping information (from AST nodes to JavaScript objects), we

used a method similar to that described in Section 3.3.2.6, namely, for each AST node that can be included in the DFG, we instrument the native code emitted for it. This instrumentation makes sure that the evaluated value of the expression corresponding to the AST node is recorded in a table. We refer to this table as the *AST-to-object mappings table*. Note that as with the AST taint table, there needs to be one AST-to-object mappings table per function invocation.

3.4.2 Optimizing Taint Propagation

V8 uses a `Context` object [V8e] to model an execution environment that corresponds to a global variable scope in JavaScript. To minimize runtime overhead incurred by taint propagation, in our prototype implementation of Mystique we chose to only propagate taint for JavaScript code with a `Context` (i.e., global variable scope) belonging to extensions. This optimization is sound since JavaScript code with a given global variable scope cannot access objects defined in another. Furthermore, even though content scripts are injected and run in the "context" web pages, they cannot use variables or functions defined by 1) the web pages, 2) other content scripts, or even 3) their own extension's pages [Con]; internally, this rule is enforced by defining separate `Context` objects for content scripts. Lastly, JavaScript code from different extensions is given separate `Contexts`.

Although the `Context` class is defined and implemented in V8, their runtime instantiation is initiated by WebKit/Blink, which differentiates the `Contexts` of web pages and extension content scripts by the terms *main* worlds and *isolated* worlds, respectively [V8b]. That is, the content scripts of an extension are run inside isolated worlds. To implement our optimization of propagating taint only for extension `Contexts`, we modified WebKit/Blink so that it notifies Mystique whenever a `Context` is instantiated for isolated worlds. On the other hand, the background page of an extension is treated by WebKit/Blink as a normal web page, in the way that the JavaScript code of the background page is run inside the main world. Thus, we also need to have WebKit/Blink notify Mystique when a `Context` is instantiated for a background page in the main world. In our prototype implementation, we currently identify background pages by checking if the page URL begins with "`chrome-extension://`".

We note that for JavaScript code with a non-extension `Context`, it is also not necessary to: 1) taint any values obtained from the taint sources described in Section 3.3.1, and 2) maintain the corresponding AST taint table or the AST-to-object mappings table.

3.4.3 `JSON.stringify` and `JSON.parse`

Evidently, if the JavaScript object passed to `JSON.stringify` contains tainted properties, then the output string needs to be tainted as well. However, given how Mystique handles tainting of object properties (Section 3.3.2.5), if the input object *itself* is not tainted but nevertheless contains property values that are tainted, then taint would not propagate to the output string. This is due to the fact

that V8 implements `JSON.stringify` as a built-in function directly in C++. To propagate taint, we added a JavaScript trampoline function that calls the underlying C++ implementation. This trampoline is also responsible for constructing a meta-object (see Section 3.3.3.2) that describes the taint status of the object being serialized. If the input to `JSON.stringify` contains tainted values, the trampoline function will taint the output, as well as update the propagation logs using the meta-object as the previous step in taint propagation.

Given our changes to `JSON.stringify`, if the input string to `JSON.parse` is tainted, we have two possibilities: 1) the string's previous step in taint propagation is a meta-object, in which case Mystique reconstructs taint according to the meta-object, otherwise 2) Mystique sets taint for the output object and recursively for all of its properties. As with `JSON.stringify`, we also inserted a JavaScript trampoline function for `JSON.parse` to achieve these.

Our treatment of `JSON.stringify` and `JSON.parse` is similar to how we handle the Chrome message passing and storage APIs (Section 3.3.3.2). However, note the difference here is that the output of `JSON.stringify` and `JSON.parse` stays within the V8 heap memory, instead of being given to the C++ IPC implementation or to external storage.

3.4.4 jQuery Request Protocol

The jQuery library is frequently used by third-party Chrome extensions. To support protocol-less URLs (which start with "`//`"), the jQuery library first retrieves the current page's URL by reading the DOM property `location.href` and parses for its protocol (e.g. `https`), which is then prepended to the request URL if its protocol is not specified [Cra]. The request URL is later passed to the `open` method of `XMLHttpRequest`. Since `location.href` is treated by Mystique as a taint source and `XMLHttpRequest`'s request URL as taint sink, this will cause an alarm to be triggered regardless of whether the original request URL is tainted or not. To fix this false positive case, Mystique creates a signature based on the AST structure of the particular assignment expression that is responsible for falsely propagating taint (i.e., [Cra]), and does not propagate taint for the expression when it is encountered.

3.5 Evaluation

In this section, we describe our experimental setup and present the results of applying Mystique to large-scale analysis of browser extensions. The dataset used in the analysis include 178,893 extensions that were crawled from the Chrome Web Store between September 2016 and March 2018, as well as a separate set of all available extensions (2,790 in total) for the Opera browser at the time of our analysis. The Chrome extensions in our dataset typically contain multiple versions of the same extension. Discounting version differences, our dataset contains 118,083 unique Chrome extensions.

Table 3.2 Summary of dataset and analysis results.

	# Extensions	# Flagged	Percentage
Chrome	178,893	3,809	2.13%
Opera	2,790	59	2.11%
Total	181,683	3,868	2.13%

Our analysis loads the Opera extensions in our modified version of Chromium - this works in most cases since the Opera browser is based on Chromium [Web18]. We were able to analyze all of the 2,790 Opera extensions using this method. As shown in Table 3.2, Mystique flagged 3,868 Chrome extensions and 59 Opera extensions as potentially leaking privacy-sensitive information. The total number of flagged extensions is 3,868 (2.13%). We finished analyzing all of the 181,683 extensions in less than a month with our experimental setup, as described next.

3.5.1 Experimental Setup

Our analysis is automated using Selenium's ChromeDriver [Chrc]. For each extension, we launch a fresh instance of our modified Chromium with the extension pre-installed using Chromium's `-load-extension` command line argument. We then simulate web browsing by using Selenium to drive the Chromium browser to visit a fixed set of URLs.

We divide this fixed set of URLs that we use to simulate web browsing into two categories: *real URLs* and *mock URLs*. For the real URLs, we serve real website responses; for the mock URLs, we serve only a static mock page. The motivation for using mock pages is to shorten analysis time by avoiding spending the time needed to load a real page inside the browser, while still be able to generate many URL load events for the extension being analyzed. In total we have 10 real URLs (of popular websites such as wikipedia.org). Mock URLs are derived from the real URLs: for each real URL, we visit it in the browser and randomly select 10 URLs that it links to (the selection was done programmatically by using Python's `random` module). Thus, in total, we drive the Chromium browser to visit 110 URLs for each extension analyzed.

Using mock pages is a common methodology that was adopted in similar works that dynamically analyze browser extensions (e.g. [Sta17]). However, one potential shortcoming of this is that an extension might expect the web page to have specific structural layouts (e.g., certain DOM elements need to be present) before it manifests malicious behaviors. To elicit malicious behaviors from such extensions, we incorporated HoneyPage [Kap14] into our analysis whenever mock URLs are visited. Note that for real URLs, it is not necessary to use HoneyPage since the real website responses are served in this case.

To minimize network traffic as well as to make the analysis more reproducible, we used a

Table 3.3 Quantifying true positive rates (TP = True Positive, FP = False Positive). Numbers in the "Precision" column are calculated as TP/(TP + FP).

	Sample Size	# TP (%)	# FP (%)	Precision
Chrome	349	272 (77.94%)	30 (8.60%)	90.07%
Opera	59	45 (76.27%)	6 (10.17%)	88.24%
Total	408	317 (77.70%)	36 (8.82%)	89.80%

tool [Cat18] to implement a replay cache that serves pre-recorded responses for real URLs. We pre-record the responses for a real URL by visiting that URL and using the same tool to save the website responses into the replay cache. We also modified the tool so that when a response is not found in the replay cache, it fetches the latest content from the Internet instead of the default behavior of returning an error status. This modification is needed since we don't want to restrict an extension's network access (e.g., extensions might query a remote server for configuration files, which is not pre-recorded in the replay cache). We remark that for dynamic websites (e.g., amazon.com), their responses are non-deterministic so that we cannot pre-record all of the possible responses (i.e., there will likely be cache misses for such websites).

We use `mitmproxy` [Mit] to serve mock page for the mock URLs, as well as to log network traffic generated during the analysis. We arranged the replay cache and `mitmproxy` in a way such that contents already pre-recorded in the replay cache will not be logged again by `mitmproxy`.

The analysis infrastructure used in our experimental evaluation consists of a local Kubernetes [Kub] cluster that can run 120 threads simultaneously. We used this cluster for parallel processing of the extensions. Each instance of our modified Chromium browser is run in a separate Docker [Doc] container.

3.5.2 Quantifying True Positive Rates

We attempt to quantify the true positive rate of Mystique's analysis by manual verification of the flagged extensions. Specifically, for each flagged extension, we examine their taint sink objects to see if they contain privacy-sensitive information and/or any data derived from such information (e.g., encrypted/hashed). We manually verified all of the 59 flagged extensions for the Opera browser. To estimate the true positive rate for the Chrome extensions, we manually verified a randomly selected sample of 349 extensions (out of a total of 3,809 flagged Chrome extensions). This sample size was chosen to target a confidence interval of 5% at a 95% confidence level, according to the standard theory on confidence intervals for proportions (e.g., [Buk92], Chapter 13).

Most of the taint sink objects encountered during this process are in plaintext. For the taint sink objects that are apparently obfuscated, encrypted and/or cryptographically hashed, we confirm

by examining the relevant portions of the extension's source code (given by the propagation logs as described in (Section 3.3.5)). Table 3.3 summarizes the results. We confirmed 272 and 45 true positives (TP) for the Chrome and Opera extensions, respectively. We were also able to confirm 30 and 6 false positives (FP) for the Chrome and Opera extensions, respectively. Note that we were not able to, from examining the taint sink objects, definitively confirm the true/false positive status for 47 of the Chrome extensions in our sample and 8 of the Opera extensions. Their taint sink objects do not seem to indicate the use of obfuscation, encryption and/or hashing. To ascertain their true/false positive status would require us to manually go through their *entire* propagation logs one by one, a resource expenditure that we currently do not have. We give our best effort estimate of the actual true positive rate in the "Precision" column in Table 3.3, calculated as TP/(TP + FP).

Since Mystique propagates taint across control-flow dependencies in addition to tracking direct data flows (Section 3.3.2.3), which as we mentioned is necessary for detecting privacy leakage from extensions, some false positives in the analysis result are inevitable. During the development of Mystique we did investigate some false positive extensions and found overtainting due to control-flow dependencies. However, verifying that this is indeed the cause of all the false positives would again require us to manually examine the taint propagation logs one by one, which as we mentioned is beyond our current resource expenditure. To ascertain the reason for the false positives, further improvements to Mystique's taint propagation logs should 1) label the taint data propagated from control-flow dependencies differently than normal data flows, and 2) include more detailed information to facilitate easy pin-pointing of where a control-flow dependency originates. These improvements would also help a human analyst to quickly discern false positive results, mitigating the inaccuracy introduced by control-flow dependencies. Nevertheless, our prototype implementation of Mystique already provides reasonable accuracy for detection of privacy-leaking extensions, and we argue that in its current form, Mystique can be incorporated as part of a triage system for online extension repositories such as the Chrome Web Store.

3.5.3 Case Studies

In this section we present the details of Mystique's analysis results, highlighting the privacy threats posed by third-party extensions.

3.5.3.1 Number of Affected Users

We begin by capturing the number of users that are affected by extensions that were flagged by Mystique. These are extensions that were found and/or have the potential to leak users' privacy-sensitive information. To do this, we first sort all of the 3,809 flagged Chrome extensions according to the number of users they have. Starting from the extension with the most users, we then manually verify their taint sink objects to see if they are true positive results, as we have done in Section 3.5.2.

Table 3.4 Top 10 true positive extensions flagged by Mystique.

	# Users	Taint Sink(s) Triggered
Avast SafePrice	10,000,000+	`chrome.storage`
Avira Browser Safety	10,000,000+	`XMLHttpRequest`
Avast Online Security	10,000,000+	`chrome.storage`
Pinterest Save Button	10,000,000+	`XMLHttpRequest`
Unlimited Free VPN - Hola	8775275	`XMLHttpRequest`
AVG SafePrice	5585975	`chrome.storage`
Pop up blocker for Chrome™- Poper Blocker	2292266	`XMLHttpRequest`
Block Site - Website Blocker for Chrome™	1468846	`XMLHttpRequest`
Trustnav Safesearch	1340990	`XMLHttpRequest`
WOT Web of Trust, Website Reputation Ratings	1231219	`XMLHttpRequest`

Table 3.4 shows the number of users for the top 10 true positive Chrome extensions that were flagged by Mystique (10,000,000+ means more than 10 million users). The data on the number of users were obtained by crawling the Chrome Web Store after Mystique has finished analyzing the extension.

From the top 10 extensions in Table 3.4, 7 used `XMLHttpRequest` directly to send privacy-sensitive information to third-party, while the remaining 3 used the `chrome.storage` API to persist such information. Note that all `XMLHttpRequests` made by the extensions listed in Table 3.4 are sent to remote network hosts (instead of `localhost`, see Section 3.5.3.4). Although third-parties cannot immediately learn of privacy-sensitive information that was persisted using the `chrome.storage` API, extensions that stored tainted values with this API should still be labeled suspicious and further investigated, for the reasons that 1) the extension might be bulk-sending such information and our analysis did not trigger the conditions necessary for the actual leaking behavior, and 2) an updated version of the extension may leak the stored information, even though the current version only stores the information for local use. The latter is especially problematic when the original author decides to sell her extension to another entity, or when her account was hijacked by a malicious actor to push out bogus updates.

It should be noted that many of the Chrome extensions flagged by Mystique are no longer available on the Chrome Web Store (either taken down by Google or the extension authors), and therefore we were not able to collect data on the number of users for these extensions. There are 1,084 unique extensions (i.e., discounting version differences) in the 3,809 Chrome extensions flagged by Mystique. Out of these 1,084 unique extensions, we obtained data on the number of users for 659 of them.

In the rest of this section, we give case studies of representative extensions that were detected by Mystique. We also show the strength of Mystique by comparing it with similar previous efforts.

3.5.3.2 SimilarWeb Library and Web of Trust

Here we look at Mystique's effectiveness at flagging extensions that have been known to leak privacy-sensitive information. The SimilarWeb tracking library and the Web of Trust extension are two cases that have drawn attention from the security community in the past. In particular, The SimilarWeb library was identified in [Wei18], and it is often bundled by extension developers to serve as a revenue source. In the report [Wei18], the author was able to find 42 extensions out of the 7,000 most popular extensions on the Chrome Web Store by searching in the extension package for specific source code strings. In comparison, Mystique found a total of 382 extensions (or 99 unique extensions discounting version differences) that include the SimilarWeb library. *Note that all of the 382 extensions were detected by Mystique's automatic analysis. Our emphasis here is to highlight Mystique's ability to detect additional cases that were missed by the approach in [Wei18].*

Specifically, to find extensions that contain the SimilarWeb library from Mystique's analysis results, we filtered the extensions flagged by Mystique (3,868 in total) by similarity in the general format of the taint sink objects. We further verified that they indeed executed the SimilarWeb library by manually examining the taint propagation logs generated by Mystique (Section 3.3.5). Overall, we were able to identify 5 more domains that received the leaked data than the original report [Wei18] (these are: starwebnet.com, fvdsuggestions.com, upgit.com, analyticstats.com, and connectwebonline.net). We also found extensions that 1) does not package the SimilarWeb library code directly but nevertheless download and execute it at runtime, and/or 2) use minified versions of the library. Note that this highlights the drawbacks of the approach used in [Wei18]: it cannot catch extensions that load tracking code externally at runtime, neither can it catch minified/obfuscated cases, since it only searched in the extension package for specific source code strings.

We next turn our attention to Web of Trust (WOT) [Chr18], an extension that provides its users with reputation and safety information about the websites based on popular reviews. Note that in order to provide its advertised functionalities, WOT has to collect the current browsing activity and use it to query a remote server. However, as mentioned, WOT has reportedly been selling its users' browsing history to third parties [PC 18], which demonstrates the potential threat of privacy abuse by such extensions.

Mystique was able to flag the WOT extension as leaking privacy-sensitive information. By examining the taint propagation report, we found that the WOT extension obfuscates the leaked data by first applying RC4 encryption and then encoding the result with double-base64 (i.e., twice base64-encode). Additionally, Mystique also flagged another extension, Filter by WOT, released by the same developers as WOT, that demonstrates the same behavior. Since these two extensions use encryption, the method used by Starov et al. [Sta17] cannot use its heuristics to decrypt the data, and therefore would not be able to detect them.

3.5.3.3 Comparison with Traffic Analysis Methods

Previous research efforts at identifying privacy-leaking browser extensions using dynamic analysis focused only on the network traffic generated by extensions (e.g., [Sta17]). Since they lack the insights into the detailed data flows inside the extensions, they have to rely on heuristics to attempt de-obfuscation of any encoded parameters and recover the plaintext. As such, they cannot handle 1) encoding schemes that are not anticipated beforehand, and 2) more importantly, extensions that use one-way hashing (e.g., MD5) or encryption. Furthermore, they cannot identify cases where extensions persist privacy-sensitive information on local storage, which as we mentioned in Section 3.5.3.1 presents opportunities for abuse and should be labeled for further investigation. Finally, since Mystique logs each step of taint propagation, it is also able to give detailed information on how sensitive data are abused *within* third-party extensions, an insight that pure network-level analysis lacks.

We present in this section case studies of extensions that were detected by Mystique, but would have been missed by previous traffic analysis methods. We start by giving two example encoding schemes that we found through Mystique, but would have evaded the heuristics used by Starov et al. [Sta17]. The first one, which we term string-to-hex encoding, is a method that simply converts each character to a two-digit hexadecimal number according to its integer value in the ASCII table. For example, in this scheme, the string "abc" would be encoded as "616263" (0x61 is the ASCII value for the character 'a'). The second encoding scheme uses plain base64 encoding, but appends to the end a fixed string "`/version=2.x/*`" (`x` ranges from 0 to 2 in our detected extensions). Since the Starov et al. [Sta17] only considered URL encoding, base64, repeated base64, gzip/deflate, and JSON-packing to attempt decoding of each individual parameters, their methodology cannot detect the string-to-hex encoding scheme. For the second scheme, although plaintext can still be recovered by attempting unmodified base64 decoding, this would not be the case if the version string was *prepended* rather than appended (i.e., slight modifications to the standard encoding schemes would invalidate their heuristics). The simplicity of these two encoding schemes nevertheless points to the problems faced by approaches that rely on heuristics to decode network traffic.

On the other hand, if the extension employs one-way hashing or encryption, then it is not possible to recover the plaintext by merely applying heuristic to the captured network traffic. For one-way hashing, third-parties that are interested in learning the browsing activities of the extension users can simply build beforehand a table of hashes of all the popular website's URLs (e.g., from Alexa top sites). The most frequently encountered hash algorithm used by extensions is currently MD5. For extensions that use encryption, apart from the aforementioned WOT extension, which uses RC4, we also observed another extension that uses ROT-13 (a simple encryption that maps a letter to another that is 13 places after it in the alphabet).

We point out that although traffic analysis methods that match the traffic features to those

generated by privacy-leaking extensions (e.g. [Wei17]) can potentially catch extensions that use encoding methods not thought of beforehand, or even those that use encryption, such methods are prone to evasion whereby attackers mask their network traffic with noise. Mystique's tracking of tainted data flows is not affected by such evasion.

3.5.3.4 Extensions that leak to `localhost`

From Mystique's analysis results, we frequently find extensions whose taint sink objects are seemingly sent to `localhost` on a particular port. Upon closer examination, these extensions typically serve as a complement to native applications that are already installed on the user's computer. These native applications are the ones responsible for listening for the requests on `localhost`. It is possible that the desktop application would then be the one that actually leaks browsing activities to a remote server. To be conservative, any such extension should be labeled for further investigation (i.e., leakage to `localhost` should be treated in the same way as triggering of local storage taint sink).

3.6 Limitations and Future Work

Mystique fundamentally relies on runtime dynamic analysis of browser extensions in order to detect privacy-intrusive behaviors, and as with all dynamic analysis systems, the successful detection of malicious behaviors depends on triggering such behaviors during the analysis. And even though we incorporated HoneyPage [Kap14] into our analysis to aid in actively triggering malicious behaviors from extensions, HoneyPage is not without limitations and cannot guarantee complete coverage. While code coverage is an important metric to obtain for a dynamic analysis system such as Mystique, its measurement poses challenges in the web context. For example, a browser extension can implement some part of its functionality in JavaScript code that is fetched remotely from a server during runtime, and the fetched content might be different depending on a number of factors, such as whether the server is being queried by a security crawler (i.e., web cloaking [Inv16]). To this end, our future research efforts will look into complementing Mystique with static program analysis techniques, such as [Guh11; Sax10], in order to deduce properties of the JavaScript source code and automatically trigger privacy-intrusive behaviors. It will also be helpful to combine techniques from previous works to detect when when cloaking is employed by malicious sites.

Privacy-sensitive information can also be leaked via side-channel attacks (e.g., [Wei11]). In this chapter, along with other similar previous research efforts, we consider side-channel attacks to be out-of-scope. Orthogonal techniques are needed to mitigate the impact of side-channel attacks.

3.7 Conclusion

In this chapter, we presented the design and implementation of the first information flow tracking tool for the Chromium browser. To overcome the complexities posed by both the JavaScript language as well as the V8 engine, we adopted a novel hybrid approach to runtime taint propagation. Based on this tool, we proposed Mystique, a framework for analyzing Chrome extensions. We applied Mystique to conduct a large-scale study of extensions from the Chrome Web Store, with respect to their privacy practices. To this end, we analyzed 178,893 Chrome extensions and 2,790 Opera extensions, and flagged 3,868 (2.13%) of them as potentially leaking privacy-sensitive information. We found that the top 10 of the Chrome extensions that we confirmed to be leaking privacy-sensitive information have more than 60 million users combined. We also uncovered a number of obfuscation methods that were missed by previous work. Our results demonstrate the feasibility and effectiveness of Mystique, and shed light on the privacy practices of browser extensions, highlighting the threat posed to user privacy.

CHAPTER

4

DETECTING FILTER LIST EVASION WITH EVENT-LOOP-TURN GRANULARITY JAVASCRIPT SIGNATURES

Content blocking is an important part of a performant, user-serving, privacy respecting web. Current content blockers work by building trust labels over URLs. While useful, this approach has many well understood shortcomings. Attackers may avoid detection by changing URLs or domains, bundling unwanted code with benign code, or inlining code in pages.

The common flaw in existing approaches is that they evaluate code based on its delivery mechanism, not its behavior. In this chapter we address this problem by building a system for generating signatures of the privacy-and-security relevant behavior of executed JavaScript. Our system uses as the unit of analysis each script's behavior during each turn on the JavaScript event loop. Focusing on event loop turns allows us to build highly identifying signatures for JavaScript code that are robust against code obfuscation, code bundling, URL modification, and other common evasions, as well as handle unique aspects of web applications.

This chapter makes the following research contributions to the problem of measuring and improving content blocking on the web: First, we design and implement a novel system to build per-event-loop-turn signatures of JavaScript behavior through deep instrumentation of the Blink and V8 runtimes. Second, we apply these signatures to measure how much privacy-and-security harming

code is missed by current content blockers, by using EasyList and EasyPrivacy as ground truth and finding scripts that have the same privacy and security harming patterns. We build 1,995,444 signatures of privacy-and-security relevant behaviors from 11,212 unique scripts blocked by filter lists, and find 3,589 unique scripts hosting known harmful code, but missed by filter lists, affecting 12.48% of websites measured. Third, we provide a taxonomy of ways scripts avoid detection and quantify the occurrence of each. Finally, we present defenses against these evasions, in the form of filter list additions where possible, and through a proposed, signature based system in other cases.

As part of this chapter, we share the implementation of our signature-generation system, the data gathered by applying that system to the Alexa 100K, and 586 AdBlock Plus compatible filter list rules to block instances of currently blocked code being moved to new URLs.

4.1 Introduction

Previous research has documented the many ways content blocking tools improve privacy, security, performance, and user experience online (e.g., [Mir18; Gar17; Li12; Puj15]). These tools are the current stage in a long arms race between communities that maintain privacy tools, and online trackers who wish to evade them.

Initially, communities identified domains associated with tracking, and generated hosts files that would block communication with these undesirable domains. Trackers, advertisers, and attackers reacted by moving tracking resources to domains that served both malicious and user-serving code, circumventing host based blocking. In response, content blocking communities started identifying URLs associated with undesirable code, to distinguish security-and-privacy harming resources from user-desirable ones, when both were *served from the same domain*, such as with content delivery networks (CDNs).

URL-based blocking is primarily achieved through the crowd-sourced generation of filter lists containing regular-expression style patterns that determine which URLs are desirable and which should be blocked (or otherwise granted less functionality). Popular filter lists include EasyList (EL) and EasyPrivacy (EP). The non-filter-list based web privacy and security tools (e.g., Privacy Badger, NoScript, etc.) also use URL or domain-level determinations when making access control decisions.

However, just as with hosts-based blocking, URL-based blocking has several well known weaknesses and can be easily circumvented. Undesirable code can be moved to one-off, rare URLs, making crowdsourced identification difficult. Furthermore, such code can be mixed with benign code in a single file, presenting content blockers with a lose-lose choice between allowing privacy or security harm, or a broken site. Finally, unwanted code can also be "inlined" in the site (i.e., injected as text into a `<script>` element), making URL level determinations impossible.

Despite these well known and simple circumventions, the privacy and research community lacks even an understanding of the scale of the problem, let alone useful, practical defenses. Put differently,

researchers and activists know they *might* be losing the battle against trackers and online attackers, but lack measurements to determine if this is true, and if so, by how much. Furthermore, the privacy community lacks a way of providing practical (i.e., web-compatible) privacy improvements that are robust regardless of how the attackers choose to deliver their code.

Fundamentally, the common weakness in URL-based blocking tools is, at its root, a mismatch between the targeted behavior (i.e., the privacy-and-security harming behavior of scripts), and the criteria by which the blocking decisions are made (i.e., the delivery mechanism). This mismatch allows for straightforward evasions that are easy for trackers to implement, but difficult to measure and defend against.

Addressing this mismatch requires a solution that is able to identify the behaviors already found to be harmful, and base the measurement tool and/or the blocking decisions on those behaviors. A robust solution must target a granularity *above* individual feature accesses (since decisions made at this level lack the context to distinguish between benign and malicious feature use) but *below* the URL level (since decisions at this level lack the granularity to distinguish between malicious and benign code delivered from the same source). An effective strategy must target harmful behavior independent of how it was delivered to the page, regardless of what other behavior was bundled in the same code unit.

In this chapter, we address the above challenges through the design and implementation of a system for building signatures of privacy-and-security harming functionality implemented in JavaScript. Our system extracts script behaviors that occur in one JavaScript event loop turn [Moz20], and builds signatures of these behaviors from scripts known to be abusing user privacy. We base our ground truth of known-bad behaviors on scripts blocked by the popular crowdsourced filter lists (i.e., EL and EP), and generate signatures to identify patterns in how currently-blocked scripts interact with the DOM, JavaScript APIs (e.g., the Date API, cookies, storage APIs, etc), and initiate network requests. We then use these signatures of known-bad behaviors to identify the same code being delivered from other URLs, bundled with other code, or inlined in a site.

We generate per-event-loop-turn signatures of known-bad scripts by crawling the Alexa 100K with a novel instrumented version of the Chromium browser. The instrumentation covers Chromium's Blink layout engine and its V8 JavaScript engine, and records script interactions with the web pages into a graph representation, from which our signatures are then generated. We use these signatures to both measure how often attackers evade filter lists, and as the basis for future defenses.

In total we build 1,995,444 high-confidence signatures of privacy-and-security harming behaviors (defined in Section 4.3) from 11,212 scripts blocked by EasyList and EasyPrivacy. We then use our browser instrumentation and collected signatures to identify 3,589 new scripts containing identically-performing privacy-and-security harming behavior, served from 1,965 domains and affecting 12.48% of websites. Further, we use these signatures, along with code analysis techniques from existing research, to categorize the *method* trackers use to evade filter lists. Finally, we use

our instrumentation and signatures to generate new filter list rules for 720 URLs that are moved instances of known tracking code, which contribute to 65.79% of all instances of filter list evasion identified by our approach, and describe how our tooling and findings could be used to build defenses against the rest of the 34.21% instances of filter list evasions.

4.1.1 Contributions

This chapter makes the following research contributions to improving the state of web content blocking:

1. The **design and implementation** of a system for generating signatures of JavaScript behavior. These signatures are robust to popular obfuscation and JavaScript bundling tools and rely on extensive instrumentation of the Blink and V8 systems.

2. A **web-scale measurement of filter list evasion**, generated by measuring how often privacy-sensitive behaviors of scripts labeled by EasyList and EasyPrivacy are repeated by other scripts in the Alexa 100K.

3. A **quantified taxonomy of filter list evasion techniques** generated by how often scripts evade filter lists by changing URLs, inlining, or script bundling.

4. 586 new **filter list rules** for identifying scripts that are known to be privacy-or-security related, but evade existing filter lists by changing URLs.

4.1.2 Research Artifacts and Data

As part of this chapter we also share as much of our research outcomes and implementation as possible. We share the source of our Blink and V8 instrumentation, along with build instructions. Further, we share our complete dataset of applying our JavaScript behavior signature generation pipeline to the Alexa 100K, including which scripts are encountered, the execution graphs extracted from each measured page, and our measurements of which scripts are (or include) evasions of other scripts.

Finally, we share a set of AdBlock Plus compatible filter list additions to block cases of websites moving existing scripts to new URLs (i.e., the subset of the larger problem that *can* be defended against by existing tools) [Sem]. We note many of these filter list additions have already been accepted by existing filter list maintainers, and note those cases.

4.2 Problem Area

This section describes the evasion techniques that existing content blocking tools are unable to defend against, and which the rest of this chapter aims to measure and address.

4.2.1 Current Content Blocking Focuses on URLs

Current content-blocking tools, both in research and popularly deployed, make access decisions based on URLs. AdBlock Plus and uBlock Origin, for example, use crowd-sourced filter lists (i.e. lists of regex-like patterns) to distinguish trusted from untrusted URLs.

Other content blockers make decisions based on the domain of a resource, which can be generalized as broad rules over URLs. Examples of such tools include Ghostery, Firefox's "Tracking Protection", Safari's "Intelligent Tracking Protection" system, and Privacy Badger. Each of these tools build trust labels over domains, though they differ in both how they determine those labels (expert-curated lists in the first two cases, machine-learning-like heuristics in the latter two cases), and the policies they enforce using those domain labels.

Finally, tools like NoScript block all script by default, which conceptually is just an extremely general, global trust label over all scripts. NoScript too allows users to create per-URL exception rules.

4.2.2 URL-Targeting Systems Are Circumventable

Relying on URL-level trust determinations leaves users vulnerable to practical, trivial circumventions. These circumventions are common and well understood by the web privacy and security communities. However, these communities lack both a way to measure the scale of the problem and deploy practical counter measures. The rest of this subsection describes the techniques used to evade current content-blocking tools:

4.2.2.1 Changing the URL of Unwanted Code

The simplest evasion technique is to change the URL of the unwanted code, from one identified by URL-based blocking tools to one not identified by blocking tools. For example, a site wishing to deploy a popular tracking script (e.g., *https://tracker.com/analytics.js*), but which is blocked by filter lists, can copy the code to a new URL, and reference the code there (e.g., *https://example.com/abc.js*). This will be successful until the new URL is detected, after which the site can move the code again at little to no cost. Tools that generate constantly-changing URLs, or which move tracking scripts from a third-party to the site's domain (first party) are a variation of this evasion technique.

4.2.2.2 Inlining Tracking Code

A second evasion technique is to inline the blocked code, by inserting the code into the text of a `<script>` tag (as opposed to having the tag's `src` attribute point to a URL, i.e., an external script). This process can be manual or automated on the server-side, to keep inlined code up to date. This technique is especially difficult for current tools to defend against, since they lack a URL to key off.[1]

4.2.2.3 Bundling Tracking Code with Benign Code

Trackers also evade detection by bundling tracking-related code with benign code into a single file (i.e., URL), and forcing the privacy tool to make a single decision over both sets of functionality. For example, a site which includes tracking code in their page could combine it with other, user-desirable code units on their page (e.g., scripts for performing form validation, creating animations, etc.) and bundle it all together into a single JavaScript unit (e.g., *combined.min.js*). URL-focused tools face the lose-lose decision of restricting the resource (and breaking the website, from the point of view of the user) or allowing the resource (and allowing the harm).

Site authors may even evade filter lists unintentionally. Modern websites use build tools like WebPack[2], Browserify[3], or Parcel[4] that combine many JavaScript units into a single, optimized script. (Possibly) without meaning to, these tools bypass URL-based blocking tools by merging many scripts, of possibly varying desirability, into a single file. Further, these build tools generally "minify" JavaScript code, or minimize the size and number of identifiers in the code, which can further confuse naive code identification techniques.

4.2.3 Problem - Detection Mismatch

The root cause for why URL-based tools are trivial to evade is the mismatch between what content blockers want to block (i.e., the undesirable script behaviours) and how content blockers make access decisions (i.e., how the code was delivered to the page). Attackers take advantage of this mismatch to evade detection; URLs are cheap to change, script behavior is more difficult to change, and could require changes to business logic. Put differently, an effective privacy-preserving tool should yield the same state in the browser after executing the same code, independent of how the code was delivered, packaged, or otherwise inserted into the document.

We propose an approach that aligns the content blocking decisions with the behaviors which are to be blocked. The rest of this chapter presents such a system, one that makes blocking decisions

[1] One exception is uBlock Origin, which, when installed in Firefox, uses non-standard API's[Moz] to allow some filtering of inline script contents. However, because this technique is rare, and also trivially circumvented, we do not consider it further in this chapter.

[2] `https://webpack.js.org/`

[3] `http://browserify.org/`

[4] `https://parceljs.org/`

based on patterns of JavaScript behavior, and not delivery URLs. Doing so provides both a way to measuring how often evasions currently occur, and the basis of a system for providing better, more robust privacy protections.

4.3 Methodology

This section presents the design of a system for building signatures of the privacy-and-security relevant behavior of JavaScript code, per event loop turn [Moz20], when executed in a web page. The web has a single-threaded execution model, and our system considers the sum of behaviors each script engages in during each event loop turn, from the time the script begins executing, until the time the script yields control.

In the rest of this section, we start by describing why building these JavaScript signatures is difficult, and then show how our system overcomes these difficulties to build high-fidelity, per event-loop-turn signatures of JavaScript code. Next, we discuss how we determined the ground truth of privacy-and-security harming behaviors. Finally, we demonstrate how we build our collection of signatures of known-harmful JavaScript behaviors (as determined by our ground truth), and discuss how we ensured these signatures have high precision (i.e., they can accurately detect the same privacy-and-security harming behaviors occurring in different code units).

4.3.1 Difficulties in Building JavaScript Signatures

Building accurate signatures of JavaScript behavior is difficult for many reasons, many unique to the browser environment. First, fingerprinting JavaScript code on the web requires instrumenting both the JavaScript runtime *and* the browser runtime, to capture the downstream effects of JavaScript DOM and Web API operations. For example, JavaScript code can indirectly trigger a network request by setting the `src` attribute on an `<img>` element.[5] Properly fingerprinting such behavior requires capturing both the attribute modification and the resulting network request, even though the network request is not *directly* caused by the script. Other complex patterns that require instrumenting the relationship between the JavaScript engine and the rendering layer include the unpredictable effects of writing to `innerHTML`, or writing text inside a `<script>` element, among many others.

Second, the web programming model, and the extensive optimizations applied by JavaScript engines, make attributing script behaviors to code units difficult. Callback functions, `eval`, scripts inlined in HTML attributes and JavaScript URLs, JavaScript microtasks,[6] and in general the async nature of most Web APIs make attributing JavaScript execution to its originating code unit extremely difficult, as described by previous work.[7] Correctly associating JavaScript behaviors to the

[5] Google Analytics, for example, uses this pattern.

[6] `https://javascript.info/microtask-queue`

[7] Section 2.C. of [Iqb20] includes more discussion of the difficulties of JavaScript attribution

responsible code unit requires careful and extensive instrumentation across the web platform.

Third, building signatures of JavaScript code on the web is difficult because of the high amount of indeterminism on the platform. While in general JavaScript code runs single threaded, with only one code unit executing at a time, there is indeterminism in the ordering of events, like network requests starting and completing, behaviors in other frames on the page, and the interactions between CSS and the DOM that can happen in the middle of a script executing. Building accurate signatures for JavaScript behavior on the web requires carefully dealing with such cases, so that generated signatures include only behaviors and modifications deterministically caused by the JavaScript code unit.

4.3.2 Signature Generation

Our solution for building per-event-loop signatures of JavaScript behavior on the web consists of four parts: (i) accurately attributing DOM modifications and Web API accesses to the responsible JavaScript unit (ii) enumerating which events occur in a deterministic order (and excluding those which vary between page executions) (iii) extracting both immediate and downstream per-event-loop-turn activities (iv) post-processing the extracted signatures to address possible ambiguities.

This subsection proceeds by giving a high-level overview of each step, enough to evaluate its correctness and boundaries, but excluding some low-level details we expect not to be useful for the reader. However, we are releasing all of the code of this project to allow for reproducibility of our results and further research [Sem].

4.3.2.1 JavaScript Behavior Attribution

The first step in our signature-generation pipeline is to attribute all DOM modifications, network requests, and Web API accesses to the responsible actor on the page, usually either the parser or a JavaScript unit. This task is deceptively difficult, for the reasons discussed in Section 4.3.1, among others.

To solve this problem, we used and extended PageGraph,[8] a system for representing the execution of a page as a directed graph. PageGraph uses nodes to represent elements in a website's environment (e.g., DOM nodes, JavaScript units, fetched resources, etc.) and edges to describe the interaction between page elements. For example, an edge from a script element to a DOM node might encode the script setting an attribute on that DOM node, while an edge from a DOM node to a network resource might encode an image being fetched because of the `src` attribute on an `<img>` node. Figure 4.1 provides a simplified example of a graph generated by PageGraph.

All edges and nodes in the generated graphs are fully ordered, so that the order that events occurred in can be replayed after the fact. Edges and nodes are richly annotated and describe, for

[8] `https://github.com/brave/brave-browser/wiki/PageGraph`

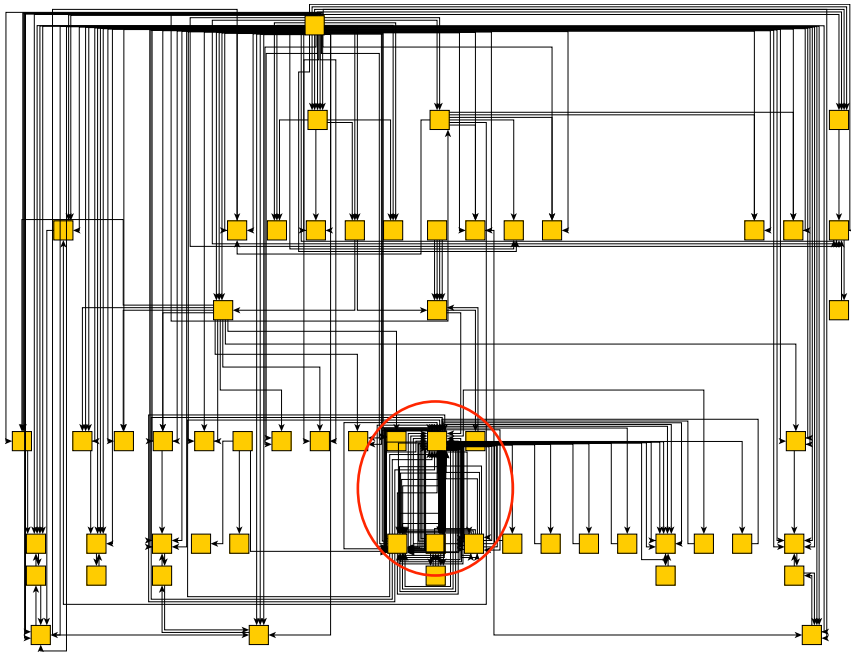

Figure 4.1 Simplified rendering of execution graph for `https://theoatmeal.com`. The highlighted section notes the subgraph attributed to Google Analytics tracking code.

example, the type of DOM node being created (along with parents and siblings it inserted alongside), the URL being fetched by a network request, or which internal V8 script id[9] a code unit in the graph represents.

We use PageGraph to attribute all page activities to their responsible party. In the following steps we use this information to determine what each script did during each turn of the event loop.

4.3.2.2 Enumerating Deterministic Script Behaviors

Next, we selected page events that will happen in a deterministic order, given a fixed piece of JavaScript code. While events like DOM modifications and calls to (most) JavaScript APIs will happen in the same order each time the same script is executed, other relevant activities (e.g., the initiation of most network requests and responses, timer events, activities across frames) can happen in a different order each time the same JavaScript code is executed. For our signatures to match the same JavaScript code across executions, we need to exclude these non-deterministic behaviors from the signatures that we generate.

Table 4.1 presents a partial listing of which browser events occur in a deterministic order (and so are useful inputs to code signatures) and which occur in a non-deterministic ordering (and so should not be included in signatures).

[9] `https://v8docs.nodesource.com/node-0.8/d0/d35/classv8_1_1_script.html`

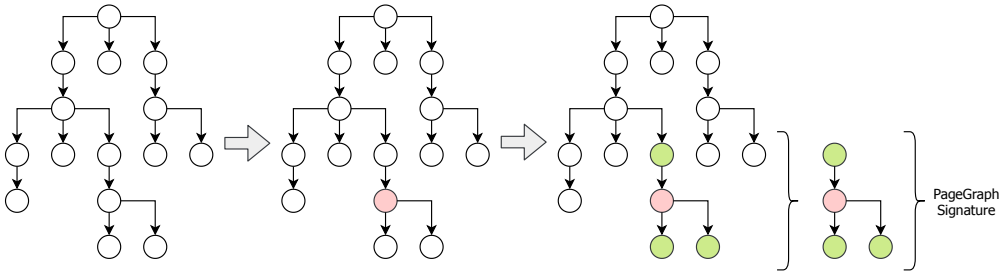

Figure 4.2 PageGraph signature generation. The red node represents a script unit that executed privacy-related activity and the green nodes are the ones affected by the script unit during one event loop turn. The extracted signature is a subgraph of the overall PageGraph.

Table 4.1 Partial listing of events included in our signatures, along with whether we treat those events as privacy relevant, and whether they occur in a deterministic order, given the same JavaScript code.

Instrumented Event	Privacy?	Deterministic?
HTML Nodes		
node creation	no	yes
node insertion	no	yes
node modification	no	yes
node deletion	no	yes
remote frame activities	no	no
Network Activity		
request start	yes	some[10]
request complete	no	some[10]
request error	no	some[10]
API Calls		
timer registrations	no	yes
timer callbacks	no	no
JavaScript builtins	no	some[11]
storage access	yes	yes[12]
other Web APIs	no[13]	some

4.3.2.3 Extracting Event-Loop Signatures

Next, we use the PageGraph generated graph representation of page execution, along with the enumeration of deterministic behaviors, to determine the behaviors of each JavaScript unit during

[10] Non-async scripts, and sync AJAX, occur in a deterministic order.

[11] Most builtins occur in deterministic order (e.g. Date API), though there are exceptions (e.g. setTimeout callbacks).

[12] `document.cookie`, `localStorage`, `sessionStorage`, and `IndexedDB`

[13] While many Web API can have privacy effects (e.g. WebRTC, browser fingerprinting, etc.) we do not consider such cases in this chapter, and focus only on the subset of privacy-sensitive behaviors relating to storage and network events.

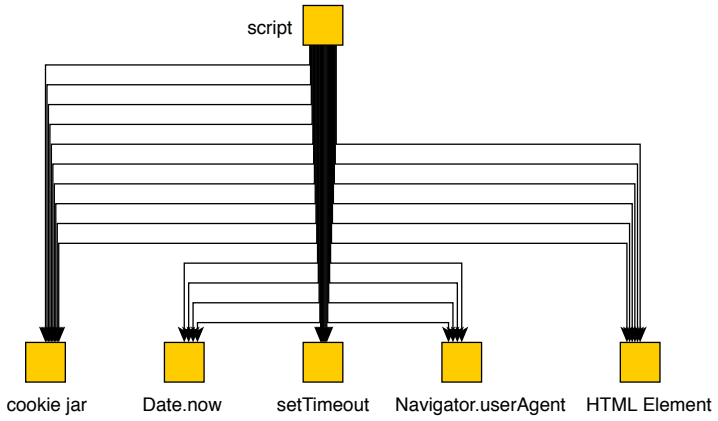

Figure 4.3 Simplified signature of Google Analytics tracking code. Edges are ordered by occurrence during execution, and nodes depict Web API and DOM elements interacted with by Google Analytics.

each event loop turn (along with deterministically occurring "downstream" effects in the page). Specifically, to obtain the signature of JavaScript behaviors that happened during one event-loop turn, our approach extracts the subgraph depicting the activities of each JavaScript unit, for each event loop turn, that occurred during page execution (see Figure 4.2). However, for the sake of efficiency, we do not generate signatures of script activity that do not affect privacy. Put differently, each signature is a subgraph of the entire PageGraph generated graph, and *encodes at least one privacy-relevant event*. Hereafter we refer to the extracted signatures, which depict the per-event-loop behaviors of JavaScript, as *event-loop signatures*.

For the purposes of this chapter, we consider privacy-and-security related events to consist solely of (i) storage events,[14] because of their common use in tracking, and (ii) network events,[15] since identifiers need to be exfiltrated at some point for tracking to occur. We note that there are other types of events that could be considered here, such as browser fingerprinting-related APIs, but reserve those for future work.

As an example, Figure 4.3 shows a (simplified) subgraph of the larger graph from Figure 4.1, depicting what the Google Analytics script did during a single event loop turn: accessing cookies several times (storage events), reading the browser user-agent string, creating and modifying an `<img>` element (and thus sending out network requests), etc.

At a high level, to extract event-loop signatures from a PageGraph generated graph, we determined which JavaScript operations occurred during the same event-loop turn by looking for edges with sequential ids in the graph, all attached to or descending from a single script unit. As a page executes, control switches between different script units (or other actions on the page); when one

[14] i.e. cookies, localStorage, sessionStorage, IndexedDB

[15] both direct from script (e.g., AJAX, fetch) and indirect (e.g., `<img>`)

script yields its turn on the event loop, and another script begins executing, the new edges in a graph will no longer be attached to the first script, but to the newly executing one. Event-loop turns are therefore encoded in the graph as subgraphs with sequential edges, all related to the same script node. We discuss some limitations of this approach, and why we nevertheless preferred it to possible alternatives in Section4.6.5.

More formally, we build signatures of privacy-and-security affecting JavaScript behavior using the following algorithm:[16].

(i) Extract all edges in the graph representing a privacy-effecting JavaScript operation (as noted in Table 4.1).

(ii) Attribute each of these edges to the JavaScript unit responsible. If no script is responsible (e.g., a network request was induced by the parser), abort.

(iii) Extract the maximum subgraph containing the relevant edge and responsible JavaScript code unit comprising all sequentially occurring nodes and edges. This is achieved by looking for edges that neighbor the subgraph, and which occurred immediately before (or after) the earliest (or latest) occurring event in the subgraph. If an edge is found, add it and the attached nodes to the subgraph.

(iv) Repeat step 3 until no more edges can be added to the subgraph.

Once each subgraph is extracted, a hash representation is generated by removing any edges that represent non-deterministically ordered events (again, see Table 4.1), chronologically ordering the remaining edges and nodes, concatenating each elements' type (but omitting other attributes), and hashing the resulting value. This process yields a SHA-256 signature for the deterministic behavior of every event-loop turn during which a JavaScript unit carried out at least one privacy-relevant operation.

4.3.3 Privacy Behavior Ground Truth

Next, we need a ground truth set of privacy harming signatures, to build a set of known privacy-harming JavaScript behaviors. We then use this ground truth set of signatures to look for instances where the same privacy-harming code reoccurred in JavaScript code not blocked by current content blockers, and thus evaded detection.

We used EasyList and EasyPrivacy to build a ground truth determination of privacy-harming JavaScript behaviors. If a script was identified by an EasyList or EasyPrivacy filter rule for blocking, and was not excepted by another rule, then we considered all the signatures generated from that

[16]This description omits some implementation specific details and post-processing techniques that are not fundamental to the approach. They are fully documented and described in our shared source code [Sem]

code as privacy-harming, and thus should be blocked. This measure builds on the intuition that filter rules block known bad behavior, but miss a great deal of additional unwanted behavior (for the reasons described in Section 4.2). Put differently, this approach models filter lists as targeting behaviors in code units (and that they target URLs as an implementation restriction), and implicitly assumes that filter lists have high precision but low (or, possibly just lower) recall in identifying privacy-and-security harming behaviors.

To reduce the number of JavaScript behaviors falsely labeled as privacy harming, we removed a small number of filter list network rules that blocked all script on a known-malware domain. This type of rule does not target malicious or unwanted resources, but *all* the resources (advertising and tracking related, or otherwise) fetched by the domain. As these rules end up blocking malicious and benign resources alike, we excluded them from this chapter. An example of such a rule is `$script,domain=imx.to`, taken from EasyList.

4.3.4 Determining Privacy-Harming Signatures

To generate a collection of signatures of privacy-harming JavaScript behaviors on the web, we combine our algorithm that extracts event-loop signatures (Section 4.3.2), with the ground truth of privacy-harming behaviors given by EL/EP (Section 4.3.3). Specifically, we produce this collection of signatures by visiting the Alexa top 100K websites and recording their graph representations (one graph per visited website), using our PageGraph-enhanced browser. For each visited website, we gather from its graph representation every script unit executing on the page, including remote scripts, inline scripts, script executing as JavaScript URLs, and scripts defined in HTML attributes. We then extracted signatures of JavaScript behavior during each event loop turn, and recorded any scripts that engaged in privacy-relevant behaviors.

Next, we omitted signatures that were too small to be highly identifying from further consideration. After an iterative process of sampling and manually evaluating code bodies with signature matches, we decided to only consider signatures that consisted of at least 13 JavaScript actions (encoded as 13 edges), and interacting with at least 4 page elements (encoded as nodes, each representing a DOM element, JavaScript builtin or privacy-relevant Web API endpoint).

This minimal signature size was determined by starting with an initial signature size of 5 edges and 4 nodes, and then doing a manual evaluation of 25 randomly sampled matches between signatures (i.e., cases where the same signature was generated by a blocked and not-blocked script). We had our domain-expert then examine each of the 25 randomly sampled domains to determine whether the code units actually included the same code and functionality. If the expert encountered a false positive (i.e., the same signature was generated by code that was by best judgement unrelated) the minimum graph size was increased, and 25 new matches were sampled. This process of manual evaluation was repeated until the expert did not find any false positives in the sampled matches,

Table 4.2 Statistics regarding our crawl of the Alexa 100k, to both build signatures of known tracking code, and to use those signatures to identify new tracking code.

Measurement	Value
Crawl starting date	Oct 23, 2019
Crawl ending date	Oct 24, 2019
Date of filter lists	Nov 2, 2019
Num domains crawled	100,000
Num domains responded	88,035
Num domains recorded	87,941

resulting in a minimum graph size of 13 edges and 4 nodes.

Finally, for scripts that came from a URL, we noted whether the script was associated with advertising and/or tracking, as determined by EasyList and EasyPrivacy. We labeled all signatures generated by known tracking or advertising scripts as privacy-harming (and so should be blocked by a robust content blocking tool). We treated signatures from scripts not identified by EasyList or EasyPrivacy, but which matched a signature from a script identified EasyList or EasyPrivacy, as *also* being privacy-harming, and so evidence of filter list evasion. The remaining signatures (those from non-blocked scripts, that did not match a signature from a blocked script) were treated as benign. The results of this measurement are described in more detail in Section 4.4.

4.4 Results

In this section we report the details of our web-scale measurement of filter list evasion, generated by applying the techniques described in Section 4.3 to the Alexa 100K. The section proceeds by first describing the raw website data gathered during our crawl, then discusses the number and size of signatures extracted from the crawl. The section follows with measurements of how this evasion impacts browsing (i.e., how often users encounter privacy-and-security harming behaviors that are evading filter lists) and concludes with measurements of what web parties engage in filter list evasion.

4.4.1 Initial Web Crawl Data

We began by using our PageGraph-enhanced browser to crawl the Alexa 100K, which we treated as representative of the web as a whole. We automated our crawl using a puppeteer-based tool, along with extensions to PageGraph to support the DevTools interface [17].

For each website in the Alexa 100K, our automated crawler visited the domain's landing page

[17] `https://chromedevtools.github.io/devtools-protocol/`

Table 4.3 The number of scripts whose behaviors match signatures from our ground truth set, both in total and broken down by whether they are blocked by EL/EP. The last row shows the total unique scripts (external plus inline) that evaded blocking by EL/EP. For comparison we also show the same statistics for the discarded small signatures.

	# Matched by Ground Truth Signatures	# Matched by Small Signatures
Scripts generating relevant signatures (unique)	14,801	195,727
Scripts blocked by EL/EP (total)	68,278	145,500
Scripts blocked by EL/EP (unique)	11,212	45,327
External scripts not blocked (total)	11,546	133,153
External scripts not blocked (unique)	3,091	82,483
Inline scripts not blocked	498	67,917
Total unique scripts not blocked	3,589	150,400

and rested for 60 seconds to allow for sufficient time for scripts on the page to execute. We then retrieved the PageGraph generated graph-representation of each page's execution, encoded as a GraphML-format XML file.

Table 4.2 presents the results of this crawl. From the Alexa 100K, we got a successful response from the server from 88,035 domains, and were able to generate the graph representation for 87,941. We attribute not being able successfully crawl 11,965 domains to a variety of factors, including bot detection scripts [Inv16], certain sites being only accessible from some IPs [Tsc18; Afr18], and regular changes in website availability among relatively unpopular domains. This number of unreachable domains is similar to those found by other automated crawl studies [Sny16; Jue19b]. A further 4,286 domains could not be measured because they used browser features that PageGraph currently does not correctly attribute (most significantly, module scripts).

4.4.2 Signature Extraction Results

Next, we run our signature generation algorithm (Section 4.3.2) on the graph representation of the 87,941 websites that we crawled successfully from the Alexa top 100K. In total this yielded 1,995,444 "raw" event-loop signatures from all the encountered scripts (of these 1,995,444 generated signatures, 400,166 are unique; the same script can be included in multiple websites and thus generate the same signatures for those websites). Overall, the average number of signatures generated for a website is 22.70, with a standard deviation of 22.92. The maximum number of signatures generated for a website is 368, while 6,281 out of the 87,941 crawled websites did *not* generate signatures. On the other hand, the average number of signatures generated from a single script unit is 2.54 (that is, the average from scripts that *did* generate signatures), with a standard deviation of 2.59. In our dataset, the maximum number of signatures generated from a script is 302.

We then filtered the above set of generated raw event-loop signatures to those matching the following criteria:

1. Contained at least one privacy-or-security relevant event (defined in Section 4.3.2.3)
2. Occurred at least once in a script blocked by EasyList or EasyPrivacy (i.e., a *blocked script*)
3. Occurred at least once in a script *not* blocked by EasyList and EasyPrivacy (i.e., an *evaded script*).
4. Have a minimum size of 13 edges and 4 nodes (see Section 4.3.4)

This filtering resulted in 2,001 *unique* signatures. We refer to this set of signatures as *ground truth signatures*. Our goal here is to focus only on the signatures of behaviors that are identified by EasyList and EasyPrivacy as privacy-harming, but also occur in other scripts not blocked by these filter lists. Note that this filtering implies that an evaded script is identified as long as at least one of its event-loop signatures matches one from the scripts blocked by EasyList and EasyPrivacy (i.e., we do not need multiple signature matches to confirm an evaded script). Also, recall from Section 4.3.4 that we impose a lower bound (13 edges and 4 nodes) on the signature size determined manually by our domain expert in order to reduce false positives (and hence the fourth requirement in our filtering criteria above). If we remove the restriction on the minimum signature size, then the above filtering would give us a total of 5,473 unique signatures (i.e., 3,472 were discarded as too small).

Table 4.3 summarizes the scripts from which our signature generation algorithm (Section 4.3.2) produced at least one signature in our ground truth set, both in total and broken down according to whether they are blocked by EasyList and EasyPrivacy. For comparison, we also show the corresponding statistics for the 3,472 signatures that we discarded as too small. Not surprisingly, the discarded small signatures were found in more scripts than our ground truth set. This is because the specificity of a signature is proportional to the number of script actions that it registers (e.g., a signature consisting of only one storage write operation would be found in many scripts that use the local storage API).

For our purposes we prefer precision over recall, by utilizing expert domain knowledge to set a reasonable cut-off signature size. Notice that our approach is optimized towards minimizing false positives, which means that the behavior of the script needs to be expressive enough (have enough edges/nodes) to indicate privacy-harming behavior (see §4.3.4). Small signatures are less expressive, so they resulted in our experiments in matching more scripts, which include both true/false positives.

Figure 4.4 shows the distribution of the number of unique signatures in our ground truth set that were found in scripts on each visited domain in our crawl of the Alexa top 100K (56,390 domains have at least one script where signatures from our ground truth set were found), as well as the distribution

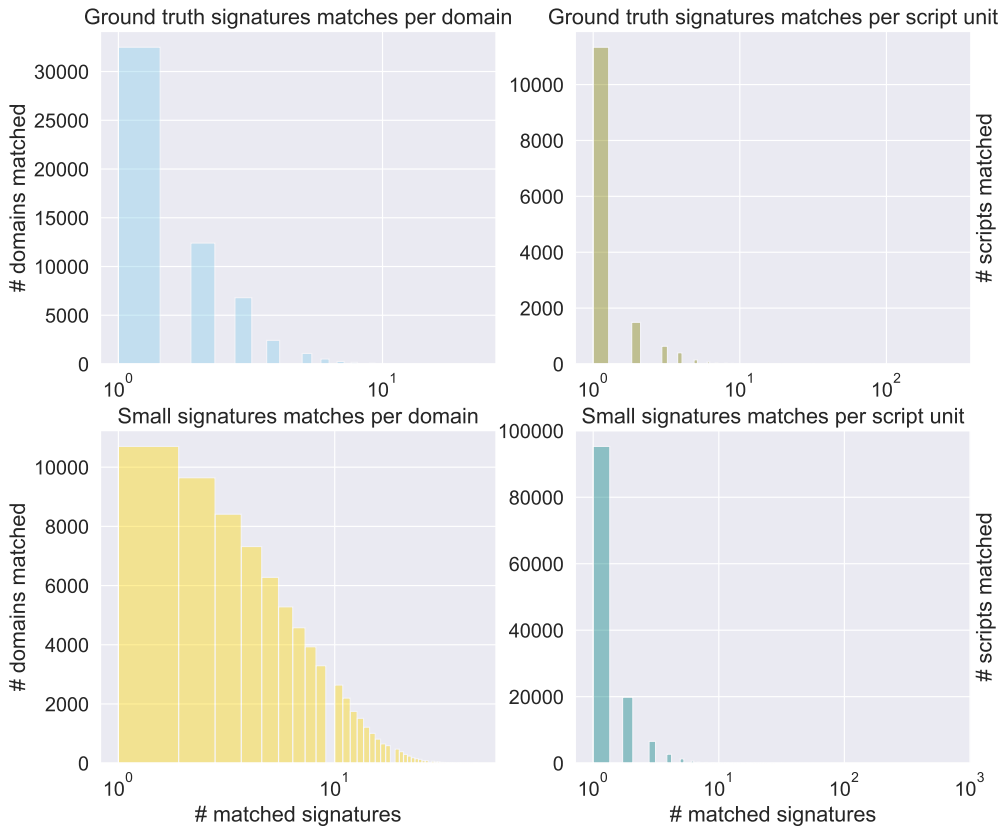

Figure 4.4 Distribution of the number of signatures per domain and the number of such signatures in each matched script unit for our ground truth dataset and for the small signatures dataset.

of the number of unique ground truth signatures in each script unit where such signatures were found. As we did in Table 4.3, for comparison here we also plot the same statistics for the small signatures.

In total, our ground truth signatures identified 3,091 new unique external script URLs (11,546 instances) hosting known-harmful behavior, but missed by filter lists, an increase in 27.57% identified harmful URLs (when measured against the number of scripts only identified by filter lists and which contain the ground truth signatures). These evading scripts were hosted on 2,873 unique domains. In addition to these evaded external scripts, our signatures also matched *inline* scripts. Inline scripts are those whose JavaScript source is contained entirely within the text content of a `script` tag, as opposed to external scripts whose URL is encoded in the `src` attribute of a `script` tag, and thus cannot be blocked by existing tools. We identified 498 instances of privacy-relevant behavior from EL/EP blocked scripts moved inline, carried out on 231 domains.

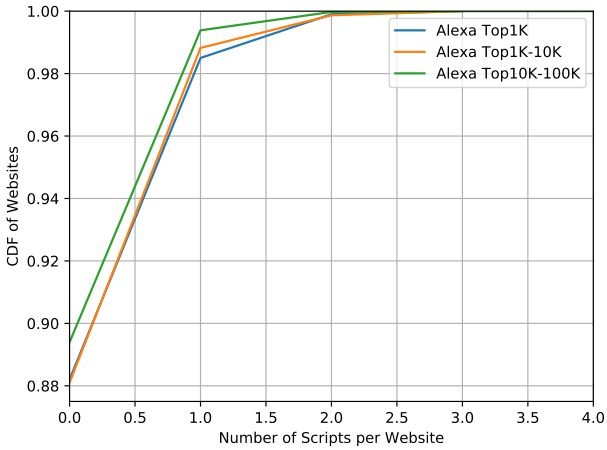

Figure 4.5 Total number of evaded scripts per website, for "popular" (Alexa top 1K), "medium" (Alexa top 1K - 10K), and "unpopular" (Alexa top 10K - 100K) websites.

4.4.3 Impact on Browsing

Next, we attempt to quantify the practical impact on privacy and security from filter list evasion. Here the focus is not on the number of parties or scripts engaging in filter list evasion, but on the number of websites users encounter on which filter list evasion occurs. We determined this by looking for the number of sites in the Alexa 100K (in the subset we were able to record correctly) that included at least one script matching a signature from a blocked script, but which was not blocked.

We find that 10,973 of the 87,941 domains measured included at least one known privacy-or-security harming behavior that was not blocked because of filter list evasion. Put differently, 12.48% of websites include at least one instance of known-harmful functionality evading filter lists.

We further measured whether these evasions occurred more frequently on popular or unpopular domains. We did so by breaking up our data set into three groups, and comparing how often filter list evasion occurred in each set. We divided our dataset as follows:

1. **Popular sites**: Alexa rank 1–1k
2. **Medium sites**: Alexa rank 1,001–10k
3. **Unpopular sites**: Alexa rank 10,001–100k

Figure 4.5 summarizes this measurement as a CDF of how many instances of filter list evasion occur on sites in each group. As the figure shows, filter list evasion occurred roughly evenly on sites in each group; we did not observe any strong relationship between site popularity and how frequently filter lists were evaded.

4.4.4 Popularity of Evading Parties

Finally, we measure the relationship, in regards to domain popularity (i.e., the delta in their ranking), between the sites hosting the scripts blocked by EasyList and EasyPrivacy, and the sites that host scripts with the same privacy-harming behaviors but evade filter list blocking. Our goal in this measurement is to understand if harmful scripts are being moved from popular domains (where they are more likely to be encountered and identified by contributors to crowdsourced filter lists) to less popular domains (where the domain can be rotated frequently). We point out that this measurement does not have a temporal component, i.e., we make no distinction with regard to whether the blocked scripts appeared earlier than the evaded ones, but merely the fact that both matched they same signature(s) from our ground truth set (see Section 4.4.2).

Specifically, we determine these ranking deltas by extracting from our results all the *unique* pairs of domains that host scripts matching the same signature of privacy-affecting behavior. That is, for a given signature in our ground truth signature set, if there are `n` unique domains hosting blocked scripts matching that signature, and `m` unique domains hosting evading scripts matching the same signature, then we would extract $n \times m$ domain pairs for that signature. Note that the final set of domain pairs that we extract across all ground truth signatures contain only unique pairs (e.g., if the domain pair `(s, t)` is extracted for both signature `sig1` and `sig2`, then it appears only once in the final set of domain pairs).

We arrange the domains in each pair as a tuple `(blocked_domain, evaded_domain)` to signify the fact that the scripts hosted on the `evaded_domain` contain the same privacy-harming semantics as those on the `blocked_domain`, and that the scripts hosted on the `evaded_domain` are not blocked by filter lists. In total we collected 9,957 such domain pairs. For the domains in each pair, we then look up their Alexa rankings and calculate their delta as the ranking of `blocked_domain` subtracted by `evaded_domain` (i.e., a negative delta means `evaded_domain` is less popular than `blocked_domain`). Since we only have the rankings for Alexa top one million domains, there are 2,898 domain pairs which we do not have the ranking information for either the `blocked_domain` or the `evaded_domain` (i.e., their popularity ranking is outside of the top 1M), including 538 pairs where *both* domains are outside of the top 1M. We use a ranking of one million whenever we cannot determine the ranking of a domain.

Figure 4.6 shows the distribution of all the ranking deltas, calculated as described above (since we cannot approximate the relative popularity for the 538 pairs where both domains are outside of Alexa top 1M, we excluded them from Figure 4.6). Note that this distribution is very symmetric: about as many of the domain pairs have negative delta as those that have positive delta, and the distribution on both sides of the x-axis closely mirrors each other. We believe this is mostly due Alexa favoring first-party domains when calculating popularity-metrics; according to Alexa's documentation, multiple requests to the same URL by the same user counts as only one page view for that URL on that

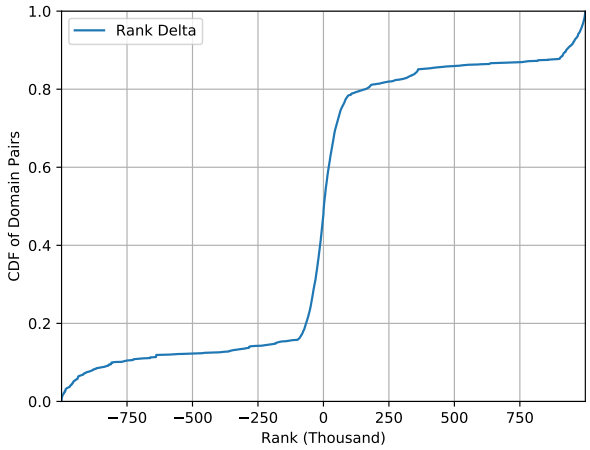

Figure 4.6 Distribution of the delta in Alexa ranking of domains hosting EL/EP blocked scripts vs. evaded scripts that matched the same signature. A negative delta means the script is moved from a popular domain to a less popular domain. The x-axis of domain rank delta is in thousands.

day [Ale]. Thus, if on a single day the user visits multiple sites that contain tracking scripts loaded from the same tracker domain, then Alexa counts those as only one visit to that tracker domain. As a result, domains that host tracking scripts tend to occupy the middle range of the Alexa ranking, and their tracking scripts are equally likely to be hosted on websites both before and after them in the Alexa rankings (web developers often choose host third-party tracking scripts on domains that they control, while at the same time minifying/obfuscating, or otherwise bundling the scripts, in order to evade filter list blocking, and we provide a taxonomy of such evasion attempts in Section 4.5).

In addition, we note that there are 140 domain pairs (out of the 9,957 extracted pairs above) where if a pair `(s, t)` is extracted, `(t, s)` is also extracted. In 72 of these we have `s == t`, while the other 68 have `s != t`. If EL/EP blocks some, but not all script URLs from a domain, it could contribute to an extracted domain pair where `s == t`; likewise, if EL/EP misses blocking some scripts on two distinct domains, then it would lead to both `(s, t)` and `(t, s)` being extracted.

Incidentally, we also measured what domains hosted scripts most often blocked by filter lists, and which domains hosted scripts that contained known-harmful behavior, but evaded detection. Tables 4.4 and 4.5 record the results of these measurements. We find that Google properties are the most frequently blocked resources on the web (Table 4.4), both for tracking and advertising resources, followed by the `addthis.com` widget for social sharing (that also conducts tracking operations).

Unsurprisingly then, we also find that these scripts are also the most common culprits in filter list evasion. Code originally hosted by Google and AddThis are the most frequently modified, inlined, moved or bundled to evade filter list detection.

Table 4.4 Most popular domains hosting resources that were blocked by filter lists. The first column records the hosting domain, the next column the number of domains loading resources from the hosting domain, the third column the number of unique URLs requested from the domain, and the final column the count of (non-unique) blocked, harmful scripts loaded from the domain.

Hosting Domain	Requesting Domains	Script URLs	Matches
google-analytics.com	47,366	44	55,980
googletagmanager.com	6,963	6,158	6,967
googlesyndication.com	5,711	38	5,711
addthis.com	1,600	51	2,464
facebook.net	1,479	1,313	1,479
adobedtm.com	1,076	1,133	2,973
amazon-adsystem.com	915	1	943
adroll.com	814	5	1,931
doubleclick.net	774	5	985
yandex.ru	610	3	684

4.5 Evasion Taxonomy

This section presents a taxonomy of techniques site authors use to evade filter lists. Each involves attackers leveraging the common weakness of current web content blocking tools (i.e., targeting well known URLs) to evade defenses and deliver known privacy-or-security harming behavior to websites.

We observed four ways privacy-and-security harming JavaScript behaviors evade filter lists: (i) moving code from a URL associated with tracking, to a new URL, (ii) inlining code on the page, (iii) combining malicious and benign code in the same file (iv) the same privacy-affecting library, or code subset, being used in two different programs.

Each of the following four subsections defines an item in our taxonomy, gives a representative observed case study demonstrating the evasion technique, and finally describes the methodology for programmatically identifying instances of the evasion technique. Table 4.6 presents the results of applying our taxonomy to the 3,589 unique scripts (12,044 instances) that we identified in Section 4.4.2 as evading filter lists.

For each taxonomy label, we perform code analysis and comparison techniques using Esprima,[18] a popular and open-source JavaScript parsing tool. We use Esprima to generate ASTs for each JavaScript file to look for structural similarities between code units. By comparing the AST node types between scripts we are resilient to code modifications that do not affect the structure of the program, like renaming variables or adding/changing comments. We consider signatures from scripts *not blocked* by EasyList and EasyPrivacy, but matching a signature generated by a script blocked by EasyList and EasyPrivacy to determine the relationship of the non-blocked script to the

[18] `https://esprima.org/`

Table 4.5 Most popular domains hosting scripts that evaded filter lists, but matched known harmful scripts. The first column records the hosting domain, the second column the number of domains that referenced the hosting domain, the third column the number of unique, evading urls on the hosting domain, and the final column the count of (non-unique) non-blocked, harmful scripts loaded from the domain.

Hosting Domain	Requesting Domains	Script URLs	Matches
google-analytics.com	5,157	4	6,412
addthis.com	1,596	50	2,455
shopify.com	543	4	545
adobedtm.com	398	331	756
tiqcdn.com	311	248	709
googletagservices.com	136	1	143
segment.com	114	107	122
tawk.to	85	85	90
outbrain.com	73	4	78
wistia.com	71	5	85

Table 4.6 Taxonomy and quantification of observed filter list evasion techniques in the Alexa 100k.

Technique	# Instances (% Total)	Unique Scripts (% Total)
Moving	7,924 (65.79%)	720 (20.06%)
Inlining	498 (4.13%)	498 (2.37%)
Bundling	117 (0.97%)	85 (13.88%)
Common Code	3,505 (29.10%)	2,286 (63.69%)

blocked scripts.

Finally, this taxonomy is not meant to categorize or imply the *goals* of the code or site authors, only the mechanisms that causes the bypassing of URL-based privacy tools. Additionally, each of the case studies are current as of this writing. However, we have submitted fixes and new filter lists rules to the maintainers of EasyList and EasyPrivacy to address these cases. As a result, sites may have changed their behavior since this was written.

4.5.1 Moving Code

The simplest filter list evasion strategy we observed is moving tracking code from a URL identified by filter lists to a URL unknown to filter lists. This may take the form of just copying the code to a new domain but leaving the path fixed,[19] leaving the script contents constant but changing the path,[20] or some combination of both. We also include in this category cases where code was moved to a new URL and minified or otherwise transformed without modifying the code's AST.

Site authors may move code from well-known URLs to unique ones for a variety of reasons. In

[19] `https://tracker.com/track.js` → `https://example.org/track.js`
[20] `https://tracker.com/track.js` → `https://tracker.com/abdcjd.js`

some cases this may be unrelated to evading filter lists. Changes to company policies might require all code to be hosted on the first party, for security or integrity purposes. Similarly, site authors might move tracking code from the "common" URL to a new URL out of some performance benefit (e.g., the new host being one that might reach a targeted user base more quickly).

Nevertheless, site authors also move code to new URLs to avoid filter list rules. It is relatively easy for filter list maintainers to identify tracking code served from a single, well known URL, and fetched from popular sites. It is much more difficult for filter list maintainers to block the same tracking code served from multitudes of different URLs.

4.5.1.1 Classification Methodology

We detect cases of "moving code" evasion by looking for cases where code with the identical AST appears at both blocked and not-blocked URLs. For each script that generated a signature that was *not* blocked by EasyList or EasyPrivacy (i.e, an evading script), we first generated the AST of the script, and then generated a hash from the ordered node types in the AST. We then compared this AST hash with the AST hash of each blocked script that also produced the same signature. For any not-blocked script whose AST hash matched one of the blocked scripts, we labeled that as a case of evasion by "moving code". We observed 720 unique script units (7,924 instances) that evade filter lists using this technique in in the Alexa 100K.

4.5.1.2 Case Study: Google Analytics

Google Analytics (GA) is a popular tracking (or analytics) script, maintained by Google and referenced by an enormous number of websites. Generally websites get the GA code by fetching one of a small number of well known URLs (e.g., `https://www.google-analytics.com/analytics.js`). As this code has clear implications for user privacy, the EasyPrivacy filter list blocks this resource, with the rule `||google-analytics.com/analytics.js`.

However, many sites would like to take advantage of GA's tracking capabilities, despite users' efforts to protect themselves. From our results, we see 125 unique cases (i.e., unique URLs serving the evaded GA code) where site authors copy the GA code from the Google hosted location and move it to a new, unique URL. We encountered these 125 new, unique Google-Analytics-serving URLs on 5,283 sites in the Alexa 100k. Google Analytics alone comprised 17.36% and 66.67% of the unique scripts and instances, respectively, of all cases in our "moving code" category. Most memorably, we found the GA library, slightly out of date, being served from `https://messari.io/js/wutangtrack.js`, and referenced from `messari.io`.

4.5.2 Inlining Code

Trackers and site authors also bypass filter lists by "inlining" their code. While usually sites reference JavaScript at a URL (e.g., `<script src=...>`), HTML also allows sites to include JavaScript as text in the page (e.g. `<script>(code)</script>`, which causes to the browser to execute script without a network request.

A side effect of this "inlining" is that URL-based privacy tools lack an opportunity to prevent the script's execution. We note that there are also additional harms from this practice, most notably performance (e.g., code delivered inline is generally not cached, even if its reused on subsequent page views). Depending on implementation, inlining scripts can also delay rendering the page, in a way remote async scripts do not.

4.5.2.1 Classification Methodology

Inlining is the most straightforward evasion type in our taxonomy scheme. Since PageGraph records the source location of each JavaScript unit that executes during the loading of a page, we can easily determine which scripts were delivered as inline code. We then compare the AST hashes (whose calculation method we described in Section 4.5.1.1) of all inline scripts to all blocked scripts that generated identical signatures. We labeled all cases where the hashed-AST of an inline script matched the hashed-AST of a script blocked by EasyList or EasyPrivacy, and both scripts generated identical signatures, as cases of evasion by "inlining". We observed 498 cases of sites moving blocked, privacy harming behaviors inline.

4.5.2.2 Case Study: Dynatrace

Dynatrace is a popular JavaScript analytics library that allows site owners to monitor how their web application performs, and to learn about their users behavior and browsing history. It is typically loaded as a third-party library by websites, and is blocked in EasyPrivacy by the filter rule `||dynatrace.com^$third-party`. Similar, client-specific versions of the library are also made available for customers to deploy on their domains, which EasyPrivacy blocks with the rule `/ruxitagentjs_`.

However, when Dynatrace wants to deploy its monitoring code on its own site `www.dynatrace.com` (and presumably make sure that it is not blocked by filter lists) it inlines the entire 203k lines JavaScript library into the header of the page, preventing existing filter lists from blocking its loading.

4.5.3 Combining Code

Site authors also evade filter lists by combining benign and malicious code into a single code unit. This can be done by trivially concatenating JavaScript files together, or by using popular build tools

that combine, optimize and/or obfuscate many JavaScript files into a single file.

Combining tracking and user-serving code into a single JavaScript unit is difficult for existing tools to defend against. Unlike the previous two cases, these scripts may be easy for filter list maintainers to discover. However, they present existing privacy tools with a no-win decision: blocking the script may prevent the privacy-or-security harm, but break the page for the user; not blocking the script allows the user to achieve their goals on the site, though at possible harm to the Web user.

4.5.3.1 Classification Methodology

We identified cases of evasion by "combing code" by looking for cases where the AST of a blocked script is a subgraph of an evaded script, where both scripts generated the same signature. To detect such cases, we again use Esprima to generate AST for all scripts that match the same signatures. We then look for cases where the AST of a blocked script is fully contained in the AST of an evaded script. More specifically, if an evaded script's AST has a subtree that is both (i) structurally identical to the AST of a blocked script (i.e., subtree isomorphism) (ii) the corresponding AST nodes in both trees have the same node type, and (iii) both scripts generated the same signature, we then labeled it as a case of evasion by "code combining". In total we observed 85 unique scripts (117 instances) that were privacy-and-security harming scripts combined with other scripts.

4.5.3.2 Case Study: Insights JavaScript SDK

Insights is a JavaScript tracking, analytics and telemetry package from Microsoft,[21] that allows application developers to track and record visitors. It includes functionality identified by EasyPrivacy as privacy-harming, and is blocked by the filter rule `||msecnd.net/scripts/a/ai.0.js`.

In order to evade EasyPrivacy, some sites copy the Microsoft Insights code from the Microsoft provided URL, and included it, among many other libraries, in a single JavaScript file. This process is sometimes called "bundling" or "packing". As one example, the website `https://lindex.com` includes the Insights library, along with the popular Vue.js and Mustache libraries, in a single URL,[22] packaged together using the popular WebPack[23] library.

4.5.4 Included Library

Finally, filter lists are unable to protect against JavaScript code including common privacy-harming libraries. Such libraries are rarely, if ever, included by the site directly, but are instead downstream dependencies by the libraries directly included on the website. These cases are common, as JavaScript build systems emphasize small, reusable libraries. Downstream libraries are difficult for filter lists

[21] `https://docs.microsoft.com/en-us/azure/azure-monitor/overview`
[22] `https://lindex.com/web-assets/js/vendors.8035c13832ab6bb90a46.js`
[23] `https://webpack.js.org/`

to target because there is no URL filter list maintainers can block; instead, filter list maintainers can only target the diverse and many bespoke JavaScript applications that include the libraries.

4.5.4.1 Classification Methodology

We identified 2,286 unique scripts (3,505 instances) in the Alexa 100K that include such privacy-and-security threatening code as a dependency. These were found by looking for common significant subtrees between ASTs. More specifically, when two scripts generated the same signature, and the AST of the blocked script and the AST of a not-blocked script, contained significant identical subtrees. We point out the possibility for false-positive here, since two scripts generating the signature might have common AST subtrees that are unrelated to the privacy-or-security-affecting behavior being signatured. (e.g., both scripts could include the jQuery library, but not have that library be the part of either code unit involved in the signature).

It is difficult to programmatically quantify the frequency of such false positives due to the complexity of the JavaScript code involved, which is often obfuscated to deter manual analysis. Nevertheless, we point out that for scripts in this category, (i) our signatures offer considerable improvements over the current state-of-the-art, by allowing automatic flagging of scripts that exhibit the same privacy-harming semantics as existing blocked scripts, and (ii) we believe these false positives to be rare, based on a best-effort, expert human evaluation (we encountered only one such case in a human evaluation of over 100 randomly sampled examples, performed during the signature size sampling described in Section 4.3).

4.5.4.2 Case Study: Adobe Visitor API

The "Visitor API" is a library built by Adobe, that enables the fingerprinting and re-identification of site visitors. It is never included directly by sites, but is instead included by many other tools, many of which also generated and sold by Adobe (e.g. Adobe Target). Some of these Adobe-generated, Visitor API-including libraries, are blocked by the EasyPrivacy rule `||adobetm.com^$third-party`.

Other libraries that include the Adobe Visitor API code though are missed by filter lists, and thus are undefended against. For example, the site `ferguson.com` indirectly loads the Visitor API code on its site, through the site's "setup" code.[24] There many other similar examples of downstream, privacy-and-security harming libraries included by diverse JavaScript applications, following this same pattern.

[24] `https://nexus.ensighten.com/ferguson/fergprod/Bootstrap.js`

4.6 Discussion

4.6.1 Comparison to Hash-Based Detection

Given the complexity of the signature-based approach presented in this chapter, we compared the usefulness of signature-based matching with a much simpler approach of detecting evasion by comparing code text. More specifically, we measured how many cases of evasion that we detected by using signatures *would have been missed* by only comparing the text (here, hash) of code units. We find that the majority of the evasion cases we identify using per-event-loop signatures would be missed by simple text-comparison approaches.

First, we note the majority of evasions discussed in Section 4.5 cannot be detected by trivial text-comparison approaches. For example, a simple technique based on comparing hashes of the script text against known-bad scripts can only find cases where the exact same script has been moved *verbatim* from one URL to another, or copied *verbatim* into a larger code unit; it would fail to find evasion resulting from even trivial modifications, minification, or processing by bundling (e.g., WebPack-like) tools.

Second, we find that our signature-based approach is able to identify a significant number of cases that text-only approaches would miss. Only 411 of the 720 unique scripts we observed in the "moving code" category of our taxonomy (Section 4.5.1) had identical script text (i.e., SHA-256 hash); in the remaining 309 cases the scripts behavior was identical but the script text was modified (a false negative rate of 42.8% *in the "moving code" category alone*). However, the simpler, hash-based approach identified 7,515 of the 7,924 incidents (i.e., not unique) of moved scripts. Put differently, a text-comparison approach would correctly handle most *cases* of scripts being moved, but would miss 42.8% unique moved scripts (note that by its nature, a script that has been moved *verbatim* to another URL is a special case of "moving code" in our taxonomy).

Furthermore, evaded scripts in the bundling and common code categories *cannot* be straightforwardly detected by comparing hashes of the script text, since by definition these scripts contain new code and thus the hash of the script will be different. Indeed, it is challenging, if not impossible, to use text-based detection methods against these evasion techniques. By comparison, since our approach targets the behavior rather than the delivery mechanism of the harmful scripts (and regardless of whether they are obfuscated and/or bundled with other scripts), it can detect evaded scripts whenever their privacy-harming functionalities are executed.

4.6.2 Countermeasures

This chapter primarily focuses on the problem of measuring how often privacy-and-security affecting code evades filter lists, by building behavioral signatures of known-undesirable code, and looking for instances where unblocked code performs the same privacy-and-security harming behaviors. In

this section we discuss how this same methodology can be used to protect web users from these privacy-and-security threatening code units.

We consider three exhaustive cases, and possible defenses against each: blocked code being moved to a new URL, privacy-and-security affecting event-loop turns that affect storage but not network, and privacy-and-security affecting event-loop turns that touch network.

4.6.2.1 Moved Code

In cases where attackers evade filter lists by moving code to a new URL, our approach can be used to programmatically re-identify those moved code units, and generate new filter lists rules for the new URLs. Using this approach, we have generated 586 new filter list URLs, compatible with existing popular content blocked tools like AdBlock Plus and uBlock Origin. Further, we have submitted many of these new filter list rules to the maintainers EasyList and EasyPrivacy; many have been upstreamed, and many more are being reviewed by the maintainers.

4.6.2.2 Event-Loop Turns Without Network

Instances of code being inlined, or privacy-or-security affecting code being combined with other code, are more difficult to defend against, and require runtime modifications. These have *not* been implemented as part of this chapter, but we discuss possible approaches for doing so here.[25] We note that the single-threaded model of the browser means that signature-matching state only needs to be maintained per JavaScript content, to track the behavior of the currently executing script; state does not need to be maintained per code unit.

In cases where the privacy-harming, event-loop signature only consists of storage events (i.e. no network behavior), we propose staging storage for the length of each event-loop turn, discarding the storage modifications if the event-loop turn matches a signature, and otherwise committing it. This would in practice be similar to how Safari's "intelligent tracking protection" system stages storage until the browser can determine if the user is being bounce-tracked.

4.6.2.3 Event-Loop Turns With Network

The most difficult situation for our signature-based system to defend against is when the event-loop turn being "signatured" involves network activity, as this may force a decision before the entire signature can be matched (i.e. if the network event occurs in the middle of signature). In such cases, runtime matching would need to operate, on average, with half as much information, and thus would not provide protection in 50% of cases. While this is not ideal, we note that this is a large improvement over current privacy tools, which provide no protections in these scenarios.

[25] These are not abstract suggestions either; we are working with a privacy-focused browser vendor to implement these proposals.

We leave ways of more comprehensively providing runtime protects against network-involving, security-and-privacy harming event-loop turns as future work.

4.6.3 Accuracy Improvements

This work generates signatures from known privacy-and-security harming scripts, using as input the behaviors discussed in Table 4.1. While we have found these signatures to be highly accurate (based on the methodology discussed in Section 4.4 and the AST-based matching discussed in Section 4.5), there are ways the signatures could be further improved. First, the signatures could be augmented with further instrumentation points, to further reduce any false positives, and build even more unique signatures per event-loop turn. Second, we expect that for many scripts, calling a given function will result in neither purely deterministic behavior, nor completely unpredictable behavior; that some subsections of code can result in more than one, but less than infinite, distinct signatures. Further crawling, therefore, could increase recall by more comprehensively generating all possible signatures for known privacy-and-security affecting code.

4.6.4 Web Compatibility

Current web-blocking tools suffer significant trade-offs between coverage and usability. Making decisions at the URL level, for example, will result in cases of over blocking (and breaking the benign parts of a website) or under blocking (and allowing the privacy harming behavior). By moving the unit of analysis to the event-loop turn, privacy tools could make finer grained decisions, and do a better job distinguishing between unwanted and benign code. While we leave an evaluation of the web-compatibility improvements of our proposed per-event-loop-turn system to further work, we note it here as a promising direction for researchers and activists looking to make practical, usable web privacy tools.

4.6.5 Limitations

Finally, we note here limitations of this work, and suggestions for how they could be addressed by future work.

4.6.5.1 Automated Crawling

The most significant limitation is our reliance on automated-crawls to build signatures. While such automated crawls are useful for covering a large portion of the web, they have significant blind spots, including missing scripts only accessible after authentication on a site, or only after performing complex interactions on a page. Prior work has attempted to deal with this though paid research-subject volunteers [Sny17], or other ways of approximating real world usage. Such efforts are beyond

the scope of this project, but we note them here for completeness.

4.6.5.2 Evasion

Second, while our behavioral-based signature approach is far more robust to evasion than existing URL focused web privacy-and-security tools, there are still cases where the current approach could be fooled. For example, if an attacker took a privacy-harming behavior currently carried out by a single script, and spread the functionality across multiple colluding code units, our system would not detect it (though it could with some post-processing of the graph to merge the behavior of colluding scripts). Similarly, attackers might introduce intentional non-determinism in their code, by, for example, shuffling the order of some operations in a way that does not affect the code's outcome.

While our system could account for some of these cases through further crawling (to account for more possible code paths) or generalizations in signature generation, we note this attack as a current limitation and area for future work.

We note, however, that our signature-based approach would be robust to many forms of obfuscation that would confuse other signature-based approaches. Because our approach relies on code's behavior, and not text representation, our approach is resilient against common obfuscation techniques like code rewriting, modifying the text's encoding, and encrypting the code. We also note that our approach would not be fooled by obfuscation techniques that only changed control flow *without also changing script behavior*; our technique would be robust against obfuscation techniques that only modify JavaScript structure.

4.6.5.3 False Positives

Our approach, like all signature-based approaches, makes trade offs between false-positive and false-negative rates. Encoding more information in a signature increases the confidence in cases where the signature matches observed behavior, but at the risk of missing more similar-but-not-identical cases. As described in Section 4.3.4, our system only builds signatures for graphs including at least 13 edges and at least 4 nodes. This minimum graph size was selected by iteratively increasing the minimum graph size until we no longer observed any false positives through manual examination.

However, it is possible that despite the above described process, our minimum signature size is not sufficient to prevent some false positives; given the number and diversity of scripts on the web, it is nearly a certainty that there are instances of both benign and undesirable code that perform the same 13 behaviors, interacting with 4 similar page structures, even if we observed no such instances in our manual evaluation. Deployments of our work that prefer accuracy over recall could achieve such by increasing the minimum graph size used in signature generation.

4.7 Conclusion

The usefulness of content blocking tools to protect Web security and privacy is well understood and popularly enjoyed. However, the URL-focus of these tools means that the most popular and common tools have trivial circumventions, which are also commonly understood, though frequently ignored for lack of alternatives.

In this chapter we make several contributions to begin solving this problem, by identifying malicious code using highly granular, event-loop turn level signatures of runtime JavaScript behavior, using a novel system of browser instrumentation and graph-based signature generation. We contribute not only the first Web-scale measurement of how much evasion is occurring on the Web, but also the ground work for practical defenses. To further contribute to the goal a privacy-and-security respecting Web, we also contribute the source code for our instrumentation and signature generation systems, the raw data gathered during this work, and filter list rules that can help users of existing tools defend against a subset of the problem [Sem].

CHAPTER

5

COOKIE SWAP PARTY: ABUSING FIRST-PARTY COOKIES FOR WEB TRACKING

As a step towards protecting user privacy, most web browsers perform some form of third-party HTTP cookie blocking or periodic deletion by default, while users typically have the option to select even stricter blocking policies. As a result, web trackers have shifted their efforts to work around these restrictions and retain or even improve the extent of their tracking capability.

In this chapter, we shed light into the increasingly used practice of relying on *first-party* cookies that are set by *third-party* JavaScript code to implement user tracking and other potentially unwanted capabilities. Although unlike third-party cookies, first-party cookies are not sent automatically by the browser to third-parties on HTTP requests, this tracking is possible because any included third-party code runs in the context of the parent page, and thus can fully set or read existing first-party cookies—which it can then leak to the same or other third parties. Previous works that survey user privacy on the web in relation to cookies, third-party or otherwise, have not fully explored this mechanism. To address this gap, we propose a dynamic data flow tracking system based on Chromium to track the leakage of first-party cookies to third parties, and used it to conduct a large-scale study of the Alexa top 10K websites. In total, we found that 97.72% of the websites have first-party cookies that are set by third-party JavaScript, and that on 57.66% of these websites there is at least one such cookie

that contains a unique user identifier that is diffused to multiple third parties. Our results highlight the privacy-intrusive capabilities of first-party cookies, even when a privacy-savvy user has taken mitigative measures such as blocking third-party cookies, or employing popular crowd-sourced filter lists such as EasyList/EasyPrivacy and the Disconnect list.

5.1 Introduction

Most of the JavaScript (JS) [Jav] code on modern websites is provided by external, third-party sources [Agt12; Nik12; Yue09; Lau17]. Third-party JS libraries execute in the context of the page that includes them and have access to the DOM interface of that page. In many scenarios it is preferable to allow third-party JS code to run in the context of the parent page. For example, in the case of analytics libraries, certain user interaction metrics (e.g., mouse movements and clicks) cannot be obtained if JS code executes in a separate `iframe`.

This cross-domain inclusion of third-party JS code poses security and privacy risks. In the web context, third-party tracking is arguably the most persistent intrusion to user privacy. Multiple previous efforts [Kri09; May12; Eng16; Pan15; Roe12] attempt to understand such tracking. Browser vendors are also beginning to restrict third-party tracking: Firefox by default blocks HTTP cookies set for known trackers [Fira], and Safari introduced Intelligent Tracking Prevention (ITP) that uses machine learning techniques to identify trackers [Web20]. However, these efforts largely focused on web tracking that utilizes third-party cookies, and not much attention has been paid on exploring how *first-party* cookies are being used in regard to web tracking. Since third-party scripts that are included in the page do not have to adhere to the restrictions of the *Same Origin Policy* (SOP) [Lek15], they can set first-party cookies on behalf of the third-party tracker. On subsequent visits, the same JS code (or code by other third parties that runs in the page) can read the cookies and transmit them back to the trackers.

One of the first works on web tracking, by Roesner et al. [Roe12], explored the case of *within-site* tracking where a third-party script, such as Google Analytics, sets a first-party cookie and then sends its value back to the third-party's server. However, that work only explored the case where the first-party cookie is sent back to the domain from which the script originated, but not cases where other different third parties are involved in receiving the cookie's value. A recent work by Fouad et al. [Fou18] explores the case where a first-party cookie's value is included in a request to a third party domain, and characterize this behavior as *"first to third party cookie syncing,"* but they do not distinguish the origin of the diffused cookies, i.e., whether they are set by third-party code. Since previous works have not fully explored first-party cookies with respect to their use in web tracking, in this work we aim at bridging this gap by shedding light into how third parties are currently leveraging first-party cookies to coordinate their efforts and share identifiers to track users.

Specifically, we conducted a large-scale analysis of the Alexa top 10K websites aimed at measuring

the prevalence of web tracking that is based on first-party cookies set by third-party scripts, which we hereafter refer to as *external cookies*. To do so, we utilize an analysis system that observes the data flows of JS code at runtime using dynamic taint analysis. This system automatically marks as tainted the cookies that are written to `document.cookie` by code that has a different origin than the current first-party context. When such cookies are accessed by JS code, dynamic taint tracking makes sure to propagate the taint to any values that are derived from the tainted cookies, and as the tainted values reach network sinks, such as `XMLHttpRequest`, an alert is raised and the event is logged.

In total, our analysis shows that 9,772 (97.72%) of the Alexa top 10K websites have first-party cookies that are set by third-party JavaScript code, and that alarmingly, 57.66% of these websites have at least one such cookie that contains a unique user identifier (i.e., tracking ID) that is diffused, whether by the original script that set the cookie or by scripts from a different third-party, to third-party destinations that are *different* than the party from which the cookie originated. This analysis is enabled by two additional core contributions of this paper: (i) we leverage and improve the methods proposed by previous works [Eng16; Eng15] to automatically detect first-party cookies, set by third-party JS, whose values contain information that uniquely identifies a user (which we refer to as UID-containing cookies), and (ii) we develop heuristics that match the values of UID-containing cookies with parts of the outgoing requests that contain tainted values, in order to identify information flows between the third party that initially set the cookies, and the third parties that receive information derived from those cookies. Our results demonstrate the privacy risk posed by external cookies, even when popular mitigation techniques, such as completely blocking third-party cookies, or crowd-sourced filter lists (e.g., EasyList, EasyPrivacy) are employed, and motivate the need for more effective privacy protection countermeasures.

We summarize our major contributions as follows:

- We bridge an important gap that has been largely neglected in previous research by shedding light into how *first-party* cookies that are set by third-party JavaScript code, which we refer to as *external cookies*, are currently being used in web tracking scenarios.
- We leverage data flow analysis techniques that give us detailed insights into the runtime behaviors of JavaScript code with respect to whether, and if so how, external cookies are being utilized by third-party JavaScript code.
- We conduct a large-scale study of the Alexa top 10K websites to assess the prevalence of external cookies and explore whether they are being used for tracking purposes.
- We develop techniques to automatically detect tracking ID cookies, and identify information flows between third parties that exchange tracking IDs. We find that external cookies are

already being actively used on a large majority of the Alexa top 10K websites to facilitate web tracking.

5.2 Background

5.2.1 HTTP Cookies

HTTP cookies can be categorized as first-party or third-party, depending on their domain of origin. The cookies set when visiting a website are considered as first-party, while those set by other domains as a result of loading external resources are considered as third-party. Consequently, if the same third-party resource (e.g., a popular JavaScript library) is present on multiple websites, it enables cross-site tracking: any third-party domain that host resources referenced by multiple websites can track users across these sites.

Recognizing the potential for this privacy abuse, all major browsers provide a way to block third-party cookies, with some browsers even blocking them by default [Firb]. Furthermore, an experimental policy that blocks cookies from known tracker domains has recently been introduced in Firefox [Fira]. Safari also implements Intelligent Tracking Prevention (ITP) that uses machine learning to classify whether a domain is likely to be a tracker, and block its cookies [Web20].

5.2.2 Accessing Cookies from JavaScript

Besides including cookies in the HTTP `Set-Cookie` header field, another way to set cookies in the user's browser is to do so programmatically via JavaScript, through the `document.cookie` property provided by the Document Object Model (DOM). We note that `document.cookie` is the only interface that JavaScript has to the cookies of a website, except those offered by the browser's extension APIs (e.g., Chrome's `chrome.cookies` [Chrb]) that are *only* accessible to browser extensions, and which we do not consider in this paper. Our focus here is how third parties that already have their JS code embedded in first-party websites can abuse this access to track users via cookies set by the embedded code.

5.2.3 Evading Third-Party Cookie Policies

Instead of using traditional third-party cookies, online trackers can simply have their JavaScript code execute in the first-party context and use `document.cookie` to set first-party *external* cookies. In contrast to third-party cookies that can be easily blocked or deleted without affecting the functionality and usability of the website, this is not so straightforward in the case of first-party cookies.

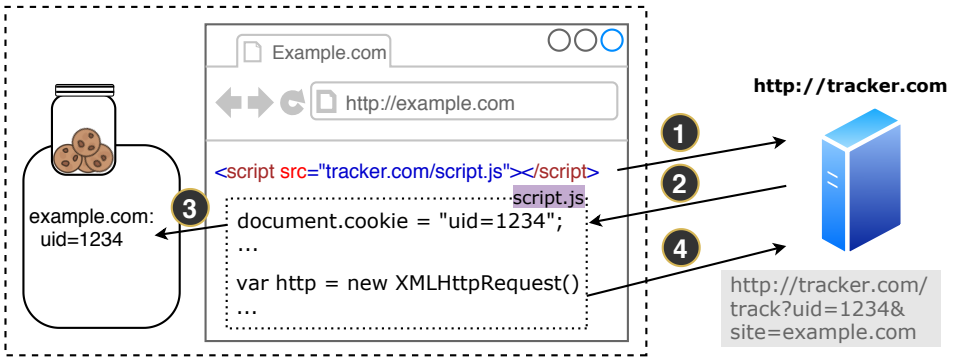

Figure 5.1 Example of *within-site* tracking [Roe12], as used by web analytics services. Third-party code stores UID by setting an *external* cookie (3), which is later sent back to the same third-party (4).

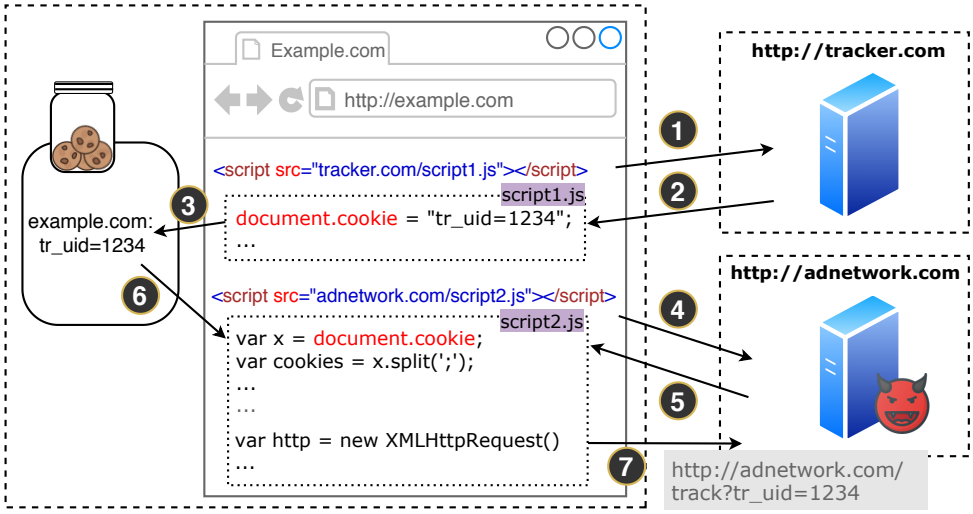

Figure 5.2 Example of UID sharing between two third parties. One third party reads the external cookie (6) that was set by another third party (3), and "steal" it (7).

In Figure 5.1 we present the common case of third-party services that have code running in the first-party website and utilizing external cookies. In this case, the third-party code sets an external cookie ❸ (note that the cookie is set for the first-party domain) and sends its value back to the third party from which the code originated ❹. This tracking method, which only allows tracking a user returning to a previously visited website, is typically employed by web analytics services, such as Google Analytics.

Although this approach may affect user privacy, as it can track the visits of a user to the first-party website, and possibly artifacts of the user's behavior while navigating the page, it can be considered as benign, or at least acceptable. However, this is not the case when the value of an external cookie is derived based on fingerprinting, when it encodes information that can be used to link the user to

a specific user profile (e.g., the user's IP address), and when the same values are being set as both a first-party and a third-party cookie.

The most important and interesting case is when the external cookie is leaked to a third party that is *different* from the one that previously set that cookie. This could be either the result of a cooperation between the two third-party services, or the result of a shady third party that uses its own code to read the cookies that were set by other third parties. As depicted in Figure 5.2, the code from `http://tracker.com` sets an external cookie (`tr_uid`) ❸, and then different code that is fetched from another third-party domain (`script2.js` from `http://adnetwork.com`) ❺ reads all the first-party cookies of the website ❻, extracts the value of `tr_uid`, and sends it to `http://adnetwork.com` ❼, essentially "stealing" the external cookie. To make matters worse, it is quite possible to have a large number of third parties being involved in reading those external cookies and exchanging identifiers between them.

5.3 Methodology

5.3.1 Challenges in Measuring External Cookies

There are many technical challenges involved in accurately measuring and understanding the usage of external cookies. First, since external cookies are *not* set upon receiving an HTTP response with a `Set-Cookie` header, but rather by third-party JS code running in the first-party context, it is not possible to distinguish external cookies from other first-party cookies simply by observing the network traffic. Thus, the measurement system needs to keep track of the origins of each JS unit executing on the page, and be able to attribute the setting of first-party cookies to the initiating script. That is, it needs to hook the JS interface for updating the cookie storage of a website, identify the initiator script domain responsible for setting cookies through this interface and, if the initiator's domain is different from the first-party domain, record the cookies being set.

Second, unlike traditional third-party cookies, external cookies are also *not* sent back to third parties encoded verbatim in the HTTP request header, but are instead read back and sent by JS code, which can potentially apply arbitrary transformations (e.g., encryption or hashing) before sending them. Therefore, the JS cookie storage interface also needs to be hooked for read accesses. Considering that the read-back values from this interface might contain both external cookies and non-external ones (i.e., first-party cookies *not* set by third-party JS code), the measurement system needs a way to attach this information to the read-back values and track any values derived from external cookies. Furthermore, the transformations applied by JS code to external cookies are not necessarily limited to using the string type: for instance, an implementation of the base64 encoding would first convert the characters of the input string to integers, which are then used as indexes in a table that produces the characters of the output string. For these reasons, a system that aims to

understand how external cookies are used needs to be able to gain detailed insights into the data flows of the JS code running on a page, *across all available data types* (and not limited to, e.g., just the string type).

Finally, given that we are interested in a large-scale measurement of external cookie usage across the web, and that a significant portion of JS code is likely to be minified/obfuscated to deter analysis [Sko19], we need an automated way to analyze the results without resorting to manual analysis of all the encountered JS code. Ideally, such a technique should allow us to (i) distinguish cookies that are used for tracking purposes from those that are not related to tracking (e.g., timestamps), (ii) find the relationship between the domains setting the cookies and those that receive them, and (iii) categorize the common usage patterns of external cookies and filter out representative cases where manual analysis yields the most benefit.

5.3.2 Dynamic Taint Analysis

Our solution to the above challenges relies on information flow analysis techniques. Specifically, we leverage and extend Mystique [Che18], which implemented dynamic taint tracking for JS by modifying the Chromium browser's V8 JS engine and its Blink layout engine. Mystique's runtime taint propagation covers *all* JS data types, and it was originally built to identify browser extensions that leak the user's browsing history. We tailored Mystique to our usage scenarios. In the following, we document all the changes we made and show how the challenges mentioned in Section 5.3.1 are addressed.

5.3.2.1 Defining `document.cookie` as Taint Source

As discussed, external cookies are set by third-party JS code, that runs in the page, using the `document.cookie` property. For our analysis such cookies need to be distinguished from other first-party cookies (e.g., traditional first-party cookies, or cookies set by JS code that originates from the first party), so that we can properly taint and track them when they are read back (together with other first-party cookies). Thus, defining `document.cookie` as taint source needs to account for both read and write accesses. To that end, we made the following changes to Chromium: (i) whenever third-party JS code *writes* to `document.cookie` to either set a new cookie or update an existing one, we mark that cookie as tainted by recording it in a global table (keyed on *both* the cookie name and the domain on which it is set), and (ii) when `document.cookie` is *read* we taint the return value (i.e., a string that concatenates all the cookies of the website) if it contains cookies that we have previously marked as tainted. However, the value of `document.cookie`, which concatenates all the cookies of the website, can potentially contain non-external first-party cookies. To avoid over-tainting those cookies, we extend Mystique's dynamic taint tracking implementation with precise byte-level taint tracking for the string type, which we discuss in the following section.

Incidentally, we consider scripts to be third-party when they originate from an eTLD+1 domain that is different from the first-party origin. We taint a cookie that is set via `document.cookie` when the *initiator* script is third-party. The eTLD+1 domain from which the initiator script is loaded is considered as the source domain of the cookie (we will use this definition in Section 5.4). The initiator script can be found from Chromium's call stack at the point when `document.cookie` is written to. We attribute the initiator script to be the bottom script on the call stack. Note that this attribution works even in the presence of asynchronously invoked JS code, since Chromium's call stack is able to trace across asynchronous functions [Chrf]. For instance, if a script registered a callback which is later invoked asynchronously, the call stack will contain information about the original script responsible for registering the callback.

5.3.2.2 Byte-Level Taint Tracking for Strings

We implement byte-level taint tracking for strings by modifying V8 to associate a Boolean array with each string that is *partially* tainted: if byte `i` in the string is tainted, then offset `i` of the Boolean array would be set to `true`, otherwise `false`. Note that if a string is *fully* tainted, then it is not necessary to allocate a Boolean array for it - we treat tainted strings that do not have an associated Boolean array as fully tainted.

To keep this byte-level taint information up-to-date, we propagate the taint across string manipulation functions such as `concat` and `split`, as well as regular expression operations. This taint propagation is similar to Lekies et al. [Lek13]. Our work, however, differs from theirs in the following ways: (i) Lekies et al. [Lek13] only handled tainting for the string type, which is not adequate for measuring external cookies. As we mentioned, our approach requires tracking the complete data flow, *across all JS data types*, a capability we leverage from Mystique, and (ii) unlike Lekies et al. [Lek13], precise byte-level taint propagation is not always possible with the approach taken by Mystique, on which our work is based (e.g., when taint is propagated across control-flow dependencies; see [Che18] for details).

5.3.2.3 Taint Sinks

To detect tainted values (i.e., information derived from external cookies) that are transmitted to third parties, we define the (i) `XMLHttpRequest`, (ii) `WebSocket`, and (iii) `src` attributes of HTML elements as taint sinks. Our system raises an alert whenever a tainted value reaches one of the sinks, for being sent to the network. Note that our approach considers all HTML elements that have a `src` attribute as taint sinks, as the browser would make a network request to fetch the resource pointed to by the `src` attribute. Since this results in a network request, affecting the `src` attribute of elements is a potential way for passing information to the third parties that provide the referenced resources.

5.3.2.4 Discussion

The modifications we made to Chromium address the first two challenges mentioned in Section 5.3.1. Specifically, as stated in Section 5.2, to the best of our knowledge, the DOM property `document.cookie` is the only interface that JavaScript has to the cookies of a website. Therefore, any external cookies that are set for a website will be detected by our modified Chromium browser; when these cookies are read back by JS code, they will be tainted by our instrumentation. Dynamic taint tracking then ensures that taint data is propagated as the JS code executes: any data that is derived from tainted values (e.g., base64-encoded) will also be tainted. Given this, we are able to track how external cookies are used by JS, and whether they are leaked over the network (i.e., whether the taint sinks are triggered by tainted values) back to third parties. Note that HTTP requests made to third-party domains do *not* have the external cookies embedded in the request header, as external cookies are set for the first-party domain. Thus, the only way for third parties to read back the external cookies is through JS that runs in the page, which ultimately needs to read them from `document.cookie`.

5.3.3 Detecting Tracking Cookies

Not all external cookies are used for web tracking. For example, some cookies may store a `true/false` flag to indicate whether a user has opted in to certain features of the website. Other examples include the client's language, geolocation, timezone, and timestamps. These values contain only coarse-grained information and as such they cannot uniquely identify a particular user. Note that in the case of timestamps, they are often paired with an additional random number to avoid collision and form a unique tracking ID (e.g., Google Analytics' `_ga` cookie [Gacc]), but by themselves they are not uniquely identifying. This classification is in line with previous works [Eng15].

To explore how external cookies are used for web tracking, we need to focus only on cookies that contain tracking IDs. To that end, we follow the methodology used in previous works [Eng15; Eng16]. The key insight here is that tracking cookies have two important properties: (i) persistent value over time, and (ii) uniqueness across different browser instances. However, as the authors themselves noted in those works, their methods are intentionally conservative since they were interested in establishing lower bounds. Also, their methods are not specific to external cookies. Thus, we adapted their methods to effectively detect tracking IDs in external cookies. Specifically, we filter the external cookies recorded by our instrumented Chromium (see Section 5.3.2) according to the following criteria:

(i) The cookie should not be a session cookie, (session cookies are deleted by the browser when the session ends [Mdnb]).

(ii) The length of the cookie's unquoted value is at least 8 characters (unquote converts URL-

encoded %xx sequences into their single character equivalents).

(iii) The cookie is always set when visiting the website using different browser instances.

(iv) The values of a cookie differ significantly across different visits to the same website using different browser instances.

Similarly to previous works, we define a "significant difference" in cookie values using the Ratcliff-Obershelp algorithm to compute similarity scores. Note that the above criteria are essentially the same ones used in previous works [Eng15; Eng16]. Our methodology differs primarily in how similarity scores are computed. In [Eng15; Eng16] the similarity scores are computed over the *entire* cookie values. However, we have frequently observed external cookie values such as "`{visitor_id: abcd, timestamp: 1234}`," where the common parts skew the similarity scores. In such cases, only the similarity between the actual visitor ID value is of interest. For this reason, we compute the similarity scores only after we have removed all the common parts between the cookie values. Specifically, we do so by first removing all timestamps from the cookie values, and then all common subsequences having a length of more than 2. The latter is done by applying the longest common subsequence (LCS) algorithm and removing the LCS from both values, and repeating until the LCS length is less than or equal to 2. The similarity score of the residual values are then computed using Ratcliff-Obershelp, and we set a cutoff value of 66% (i.e., the original cookie values are significantly different when this score is below the cutoff). The same cutoff value of 66% was also chosen in Englehardt et al. [Eng16].

5.3.4 Heuristic-Based Identifier Matching

Having identified external cookies that contain tracking IDs, we are then interested in measuring (i) which of these tracking cookies are later read back by third-party JS and leaked over the network, and (ii) the relationship among the entity (i.e., TLD+1 domain) that originally set the cookie, the entity that triggered the reading back and leaking of this cookie, and the entity that receives the leaked cookie. However, one technical challenge here is that since Mystique [Che18], on which we base our dynamic taint tracking implementation, only tracks the binary status (i.e., tainted or not) of JS values, it does not provide information at taint sinks about the provenance of the tainted values (i.e., the tracking cookies from which the tainted values triggering the sinks are derived). Thus, we need an orthogonal method to automatically extract this information. In order to address this challenge, we design and apply a set of heuristics that attempt to match the tainted values that triggered a taint sink against the identified tracking cookies. For convenience, we hereafter refer to these values (i.e., values that triggered a taint sink) as *sink objects*.

It should be noted that these heuristics are complementary to our taint tracking mechanism: they allow us to (i) verify the results of taint tracking by confirming that indeed the cases of triggered

sink objects correspond to true positives, and (ii) filter out cases of information flows that do not include unique identifiers and thus do not affect the user's privacy (as we mentioned in Section 5.3.3, some tracking cookies contain both unique tracking ID and non-unique parts). If we did *not* have the taint tracking mechanism, and we relied solely on heuristics for matching the cookies against the network traffic, then (i) we would not have any information about which third-party scripts set each external cookie (nor whether the external cookies contain tracking IDs), and which read and diffuse these cookies over the network; indeed, it is not even possible to know which of the first party cookies are external cookies, and (ii) we might miss cases where the leaked information is transformed (i.e, encrypted or obfuscated) before being sent.

The first and simplest heuristic tries to match the name or value of a tainted cookie with a substring in the sink object. In the case where the sink object is a URL, we extract the URL parameters (e.g., the argument passed to `XMLHttpRequest`'s `open` method), and check whether the name or value of a tainted cookie matches the extracted URL parameters. In the great majority of cases, this heuristic was able to match the *entire* name or value of the tainted cookie against the sink object. However, there were also some cases where the name or value of the cookie did not match a substring of the tainted sink object, but it was evident that either the name or the value of the cookie was generated by the concatenation of two or more string literals (e.g., `value1.value2`, `value1_value2`). To also account for such cases, we apply a slightly more complex heuristic that splits the cookie value to substrings, based on a set of special characters, and attempts to match these substrings with parts of the sink object.

Finally, we apply a heuristic that tries to detect cases that involve transformed values. This heuristic, similarly to the previous one, splits the tainted cookie values into substrings and then applies the base64 encoding, and the MD5, SHA1, and SHA256 hashing algorithms both to the entire cookie value and the substrings, and attempts to match them with parts of the triggered tainted sink object. While this heuristic is more complex than the previous ones, it allows us to verify cases that we cannot verify manually, where the tainted information is transformed before reaching the sink object. We present in Section 5.4 the results of applying our heuristics to the data gathered from crawling the Alexa top 10K websites.

5.3.5 Experimental Setup

We measure the prevalence of external cookie usage by crawling the Alexa top 10K websites with our instrumented Chromium browser. Each instance of Chromium runs in a separate Docker [Doc] container. After each website has finished loading, we wait for an additional two minutes, to allow sufficient time for the JavaScript code on the page to execute. This process is then repeated two more times inside the same Chromium instance, since external cookies are typically set on the first visit and are read back and sent only on subsequent visits. Note that depending on the purpose of

the analysis, we either launch a fresh instance of Chromium (i.e., no prior state) for *each* one of the websites we analyze, or a fresh instance *per cluster* of websites (e.g., websites that have external cookies with the same name) and visit each website of a cluster in the same Chromium instance *one by one*. We will provide more details about this type of analysis in Section 5.4.

Finally, we parallelize the analysis by running the Dockerized instances of the Chromium browser on a local Kubernetes cluster [Kub]. The analysis tasks (e.g., the URLs to visit) are distributed to workers via Redis [Red]. We note that since the tracking cookie detection algorithm described in Section 5.3.3 requires two separate browser instances visiting the same website at about the same time, we enqueue each website twice consecutively to Redis, so that they are picked up, one immediately after the other, by the workers in the Kubernetes cluster. Since each website is visited by two separate browser instances, essentially this means that there are two separate crawls of the Alexa top 10K. Hereafter we refer to these two separate crawls as the *main crawl* and the *control crawl*, and we treat the website visit that results from the first time it is enqueued as belonging to the main crawl (and thus the visit from the second enqueue as the control crawl).

5.4 Taint Analysis Results

In this section, we report the results of our analysis on the Alexa top 10K websites. We begin by describing the raw data from our crawl and giving an overview on the number of external cookies that we encountered. This is followed by a taxonomy breakdown of the recorded external cookies, based on the output of our algorithm for the detection of tracking cookies that is described in Section 5.3.3. We then present the results of our heuristics (Section 5.3.4) and explore in detail the relationship between the *source domains* (i.e., entities that originally set external cookies) and the *sink domains* (i.e., entities that receive information derived from confirmed tracking cookies).

5.4.1 Overview of the Initial Crawl

In total, we recorded external cookies being set in 9,772 (97.72%) out of the Alexa top 10K websites that we crawled. Table 5.1 gives an overview on the number of external cookies that were recorded in our crawl. Recall from Section 5.3.5 that in our experimental setup we essentially have two simultaneous crawls, and we refer to them as the *main* and the *control* crawl, respectively. We show in Table 5.1 the statistics for both crawls. We note that in both crawls each website is visited by a fresh instance of Chromium.

To give a more comprehensive picture regarding the external cookies on the Alexa top 10K websites, we count the number of external cookies using a variety of different criteria. In Table 5.1 we present both the number of *unique* cookies and the number of *instances* of such cookies (i.e., the total number of times those unique cookies were set/updated via `document.cookie`), accord-

Table 5.1 Statistics on the number of recorded external cookies, both in the main crawl as well as the control crawl of the Alexa top 10K. The numbers in parentheses indicate the subset in the same category that are non-session cookies.

Uniqueness Criterion (Tuple)	Main Crawl		Control Crawl	
	# Instance	# Unique	# Instance	# Unique
<cookie_domain, js_domain, cookie_name>	185,910 (112,185)	100,146 (65,871)	184,619 (111,108)	99,725 (65,273)
<cookie_domain, cookie_name>	184,714 (112,024)	98,746 (65,250)	183,454 (110,944)	98,335 (64,648)
<js_domain, cookie_name>	138,565 (93,300)	26,632 (13,323)	138,026 (92,380)	26,560 (12,906)
<cookie_name>	136,585 (92,660)	23,909 (11,876)	136,093 (91,776)	23,869 (11,543)

ing to different criteria. For example, if we define unique cookies based on the tuple <cookie_domain, js_domain, cookie_name>, where js_domain denotes the third-party domain serving the JavaScript code that sets the cookie, then in our main crawl we observed 100,146 unique external cookies, and our Chromium instrumentation recorded that these cookies were updated (via document.cookie) for a total of 185,910 times. From the numbers in Table 5.1 one can make a few observations: since the number of unique cookies according to <js_domain, cookie_name> is much lower than those based on <cookie_domain, cookie_name>, it follows that some third-party domains are responsible for serving scripts that set external cookies in a large number of websites. Another similar example is the criterion <js_domain, cookie_name> vs. <cookie_name>, which indicates either cookie name conflicts, or that the same cookie-setting script is hosted on multiple domains (e.g., to avoid filter list blocking [Che21]).

In Table 5.1 we also show the subset of cookies in each category that correspond to *non*-session cookies (i.e., numbers in parentheses). These numbers are provided to give a sense of the input to our tracking ID detection algorithm described in Section 5.3.3, the results of which we discuss next. Unless otherwise specified, hereafter we count unique external cookies based on the tuple <js_domain, cookie_name>, since it is the most appropriate given the scope of this paper, and allows us to easily cluster different first-party websites based on the external cookies that are set on them.

5.4.2 Taxonomy of Non-Session External Cookies

Next, we present the results of our tracking ID detection algorithm, that we described in Section 5.3.3. Recall that the algorithm looks for tracking IDs in *non*-session cookies, and as shown in Table 5.1, in the main crawl there are 13,323 non-session cookies (by <js_domain, cookie_name>). Note that since session cookies are deleted when the current session ends [Mdnb], by themselves they are not suitable for persistent tracking purposes. Therefore, in the rest of this paper we primarily focus on non-session cookies.

Table 5.2 Taxonomy of non-session external cookies collected from the main crawl.

Category	#Instance (% Total)	# Unique (% Total)
Tracking ID	76,617 (82.12%)	4,212 (31.61%)
Similar Value	6,500 (6.97%)	1,451 (10.89%)
Constant Value	2,245 (2.41%)	1,078 (8.09%)
Short Value	6,838 (7.33%)	5,690 (42.71%)
No Control Value	1,100 (1.18%)	892 (6.70%)
Total	93,300 (100.00%)	13,323 (100.00%)

Table 5.2 presents a taxonomy of the non-session cookies collected from our main crawl. The taxonomy categories are based on the various conditions that need to be satisfied in order for a cookie to be considered by our algorithm as a tracking cookie. We give the details of each category below.

5.4.2.1 Tracking ID

This category includes the non-session external cookies that our algorithm identifies as tracking cookies. Out of the 13,323 unique non-session external cookies, we found 4,212 (31.61%) that contain tracking IDs. We extracted the URLs serving the scripts that set these cookies and cross-referenced them against the crowd-sourced filter list EasyList/EasyPrivacy (on which the popular Adblock Plus [Adb] extension is based). Alarmingly, we found that 1,673 (nearly 40% of UID-containing cookies) would have evaded filter list blocking. We will present more details about these tracking ID cookies in Section 5.4.3, where we explore (i) the relationship between domains serving the scripts that initially set those cookies (i.e., *source domains*), and domains serving scripts which later read back these cookies and send them over the network (i.e., *retriever domains*), and (ii) the top 10 source domains, which set the most unique tracking ID cookies, and the top 10 sink domains that received the most unique tracking ID cookies. Note that for a cookie that is read back and sent over the network, the retriever domain does not necessarily need to be the same as the sink domain. For example, a third-party script might collect all external cookies set by other well-known trackers (e.g., Google Analytics) and send this information back to their servers, a commonly encountered scenario which we discuss in Section 5.4.3. In that section we will also give sample values of the top 10 tracking ID cookies of this category that are set on the most websites in our crawl of the Alexa top 10K, shown in Table 5.4.

The rest of the categories in this taxonomy of the non-session cookies are regarded by our algorithm as not containing tracking IDs, so they are not likely to be used for persistent tracking of users. For completeness, we describe each category, with the purpose of giving a sense of what the values of these cookies contain, as well as showing which of the filtering conditions in our tracking

ID detection algorithm they failed to satisfy. We provide sample values of cookies in each category as appropriate.

5.4.2.2 Similar Value

One of the more interesting categories in our taxonomy is the category of cookies that otherwise passed all the conditions of the tracking ID detection algorithm (i.e., they are non-session cookies, having minimum unquoted value length, and non-constant values between visits by different browser instances), except their values do not differ significantly in the two crawls. That is, the Ratcliff-Obershelp similarity score of their values between the two separate crawls are above our cutoff of 66%, after all the timestamps and common parts are removed. As mentioned in Section 5.3.3, timestamps *by themselves* are not uniquely identifying. We implemented our tracking ID detection algorithm so that for a given cookie under consideration, it keeps track of whether timestamps are removed from its value as part of the detection process. In total, *all but one* cookie in this category have their values differ between the two crawls due to having different timestamps. Examples of this category of cookies include `_ym_d` set by scripts from yandex.ru (e.g., on academic.ru) whose value is composed entirely of a timestamp; another example is `smct_last_ov` by smct.io (set on e.g., theclutcher.com), with a value that is a serialized JSON object, e.g., `[{"id":35060, "loaded":1594819221556, "open":null, "eng":null, "closed": null}]`. The latter example highlights the importance of removing *both* the timestamp and common parts from the cookie value, in the way that our algorithm does, prior to computing the similarity score, since the properties in a serialized JSON object are not guaranteed to have the same order [Mdna].

Incidentally, the single cookie in this category whose value difference between the crawls is not due to timestamps is `ckBAHAADS`, that is set by bahamut.com.tw on the website gamer.com.tw, and its values are `{"FA": {"a3": 9, "a1": 1}}` and `{"FA": {"a3": 11, "a1": 0}}` in the main and control crawl, respectively.

5.4.2.3 Constant Value

This category includes the non-session external cookies that are set both in the main and the control crawl, but their values are always the same, despite the fact that the websites were visited by different browser instances, and thus are not unique per user. Many of these cookies would have fallen under the "Short Value" category if we used a larger cutoff for the minimum value length (e.g., the JavaScript literal `undefined`). We also observe many cookies that contain information specific to the visited website, e.g., the website's domain name.

In addition, we also see a few cases where the cookie values resemble a user ID, for instance the `ucfunnel_uid`[1] cookie, set by aralego.net, when visiting pcstore.com.tw. Since all of our browser

instances in the crawl run from the same configuration, this might indicate the use of fingerprinting. Indeed, one limitation of our tracking ID detection algorithm is that it is unable to find cases where the cookie values are always the same due to them being generated from fingerprinting the browser. In Section 5.4.4, we explore more systematically how many such cases there are in the external cookies collected from crawling the Alexa top 10K.

Nevertheless, here we attempt a first approximation to addressing the above limitation, by cross-referencing with the popular Disconnect list [Disb], which is used by Firefox's Enhanced Tracking Protection to block known fingerprinters. Overall, we found 8 unique cookies from this taxonomy category that are set by known fingerprinters, according to Disconnect. However, we manually verified that almost none of these 8 cookies contain unique user IDs, except one that appears to be a long UID-containing string.[2]Note that the `ucfunnel_uid` cookie, mentioned above, was not among the 8 identified by Disconnect. On the other hand, as a reference we remark that Disconnect identified 36 from the "Tracking ID" category as being set by known fingerprinters (more precisely, by scripts served from known fingerprinter domains).

5.4.2.4 Short Value and No Control Value

Cookies in the "Short Value" category are only set with values that fail to meet the minimum value length requirement of our tracking ID detection algorithm (i.e., their unquoted value length is less than 8). Example values for cookies frequently found in this category include boolean flags (e.g., `0`, `1`, `true` and `false`), JavaScript literals such as `null`, as well as short strings such as "en-US", "enabled", etc. In this paper, as in previous works [Eng15; Eng16], we do not consider these to be UID-containing.

Finally, the "No Control Value" category contains cookies that were observed in the main crawl, but were not set in the control crawl. These cookies comprise 6.70% (totaling 892) of all the non-session external cookies in the main crawl. We do not consider such cookies to be stable enough to reliably track visitors to a website.

5.4.3 Cross-Domain Cookie Sharing

Having established which of the external cookies recorded in the main crawl can be used to persistently track users, we now focus on the relationships among: (i) the *source domains* that serve scripts responsible for initially setting the tracking ID cookies, which we identified in Section 5.4.2.1, (ii) the *retriever domains* which serve scripts that read back the tracking ID cookies and send them over the network, and (iii) the *sink domains* which receive information derived from tracking ID cookies,

[1]Having values of the form `88d501e0-40f3-3fcf-bf48-c8fa59bc7efd`.

[2]Cookie: `segmento`. It is set on oi.com.br by a script that originates from maxmind.com. Its value in our crawl is (truncated): `56c4339f58ee7410VgnVCM10000031d02 00a____-e99eecf7c7cc5410VgnVCM10000031d02...`

Table 5.3 Top 10 source domains, whose cookies are shared to other third parties. Note that we do not include cases where the destination is the same as the source. Also, the third-party destinations column represents the total number of third parties that receive the corresponding external cookie, across all websites.

Source Domain	Websites	Top External Cookie Shared to Third Parties		
		Cookie Name	Websites	Third-Party Destinations
google-analytics.com	3,620	_ga	3,456	329
facebook.net	2,377	_fbp	2,377	76
googletagmanager.com	1,124	_gcl_au	882	124
doubleclick.net	792	__gads	742	114
googlesyndication.com	551	__gads	551	33
cloudfront.net	469	__asc	310	4
go-mpulse.net	280	RT	279	424
chartbeat.com	243	_cb	243	6
cookielaw.org	242	OptanonConsent	242	381
adobedtm.com	239	mbox	153	15

sent to them by scripts that are served from the retriever domains. In particular, we are interested in cases where the sink domain is different from *both* the source domain, and the cookie domain (i.e., domain on which the cookie was set), since this indicates cross-origin sharing of identifiers, which further undermines user privacy.

As mentioned our taint tracking implementation only tracks the binary status (i.e., tainted or not) during taint propagation. We overcome this limitation by proposing an orthogonal method (see Section 5.3.4), that uses heuristics to match sink objects against the identified tracking ID cookies. In the following we give an overview of the results from our heuristic matching, and present the domain relationships that we identified.

5.4.3.1 Overview of Heuristic Matching Results

We applied our heuristic matching algorithms described in Section 5.3.4 to the 4,212 cookies that we identified to be containing tracking IDs (see Section 5.4.2.1), and we were able to match 3,256 (77.30% of all identified UID-containing cookies) against at least one sink object that was recorded during the crawl by our taint tracking system. The main reason why our heuristics were not able to match all the UID-containing cookies to the flagged sink objects is due to the complex transformations that take place before a cookie value is eventually leaked over the network. Indeed, that is why such a measurement study cannot completely resort on matching heuristics, but also needs to use a taint tracking system for being able to track the transformations.

We manually inspected several randomly chosen cases of the matched cookies, and we were able to verify that all of these indeed correspond to true positive cases that were correctly detected by our system. Through the manual inspection process we identified which scripts set the particular

Table 5.4 Sample values for the top stolen cookies shown in Table 5.3.

Cookie Name	Example Value
_ga	GA1.2.1687927199.1594842303
_fbp	fb.1.1594781590601.135769710
_gcl_au	1.1.2086254180.1594824717
__gads	ID=18d6b32f18c77049-22c915ffccb70097: T=1594800595:S=ALNI_MZEEA_JlMOtwGQyjjw4rJp4cwDL2A
__gads	ID=7b7fb7d54109a6c6:T=1594819953: S=ALNI_MYzCXjBjUDNFPiMFgf55XXCoZAVBg
__asc	678afacf1735170ad8631aabda0
RT	"si=a1ed29ce-7795-47c5-9830-5745fa3c04bd& z=1&dm=oracle.com&ss=kcnp67pf&sl=0&tt=0..."[3]
_cb	DQVCY6B3K-HtDvrvFv
OptanonConsent	consentId=5b39d6ae-72ba-47d9-9250-4df42b23f5d6& isIABGlobal=false&interactionCount=0...[3]
mbox	session#50141a1754164b66900a3a2f030f03cd #1594825706|PC#50141a1754164b66900a3...[3]

cookies (and verified that those are the same as reported by our system), and we were able to manually track how those cookies are transformed, and how they reach the sinks. Based on our manual analysis we are confident that the cases reported by our methodology are correct in the sense that they do all correspond to true positive cases. However, unfortunately we are not able to draw any conclusions regarding any possible false negatives (i.e., cases of such cookies that are leaked but our system fails to detect) as this would require us to manually follow all the cookies of a website, which is an almost impossible task to perform manually.

5.4.3.2 Cross-Domain Cookie Sharing

Next, we focus on exploring cases where the external cookies by one third party are shared to multiple parties (i.e., third-party destinations) different from *both* the source domain (i.e., the third party that initially set the cookie), *and* the domain on which the cookie was set. We refer to this case as *cross-domain cookie sharing*. That is, in the following we focus on flows of cookie information from one third-party to another, like the example case that is presented in Figure 5.2 and thus, we exclude cases where the cookies are sent back to the same third party that have set them. Note that this also excludes the scenario where scripts loaded from a CDN controlled by the first party sets the cookies which are then only sent back to the first party.

Out of the 3,256 cookies that were matched by our heuristics, 2,354 (55.89% of all UID-containing cookies) are leaked to a third party that is different from both the source domain and the domain on which the cookie was set. For convenience, we will refer to this set of cookies as *shared cookies*.

[3]We truncated these cookies' values due to space limitations.

Table 5.5 Top 10 third-party destination domains that receive external cookies by other third parties (different source domain).

Destination Domains	Source Domains	Cookies
google-analytics.com	198 (187)	427 (405)
doubleclick.net	189 (182)	432 (380)
facebook.com	121 (118)	224 (212)
omtrdc.net	80 (45)	325 (93)
criteo.com	59 (44)	73 (56)
adnxs.com	51 (30)	61 (34)
openx.net	51 (23)	67 (26)
googleadservices.com	48 (46)	75 (73)
taboola.com	45 (43)	70 (68)
bing.com	44 (43)	68 (67)

Table 5.3 and Table 5.5 show the highlight of our findings. In Table 5.3 we focus on the *source* of shared cookies, and we list the top 10 third parties that have their external cookies shared to other parties (i.e., destinations). We rank the source domains according to the number of first-party websites in which we observe one of their cookies being shared or leaked to other third parties, and we provide information about their top cookie that is shared in most cases. For example, we find 3,620 websites that have sink objects sharing external cookies by `google-analytics.com` to other third-party domains, with the top shared cookie being `_ga`, which is shared in 3,456 of the websites. The most interesting finding, however, is that this cookie is shared to 329 different third-party domains in total, most of which clearly do not belong to Google. Unlike traditional third-party HTTP cookies, where cookie synchronization is typically performed between two third parties (with clear consent between the parties involved, i.e., through URL redirection), our findings show that external cookies are facilitating the extensive sharing (or leakage) of information to multiple third parties.

In Table 5.5 we focus on the sink domains that receive external cookies set by other parties, and we list the top 10 ranked by the number of different source domains whose external cookies they received. For each domain in Table 5.5 we also show the total number of unique cookies received. For instance, we find that and google-analytics.com and doubleclick.net receive a total of 427 and 432 unique UID-containing cookies (that were matched by the heuristics), which were set in total by 198 and 189 different source domains, respectively. These numbers demonstrate how extensive the utilization of external cookies is for information sharing currently on the web.

In addition to the above, we also identify the retriever domains (i.e., domains serving the code that is responsible for reading the cookies and diffusing them). If the retriever domain is the same as the source domain of the cookie, this indicates that the source third party cooperates with the destination third party and willingly shares the cookie. Otherwise, if the source and retriever domains

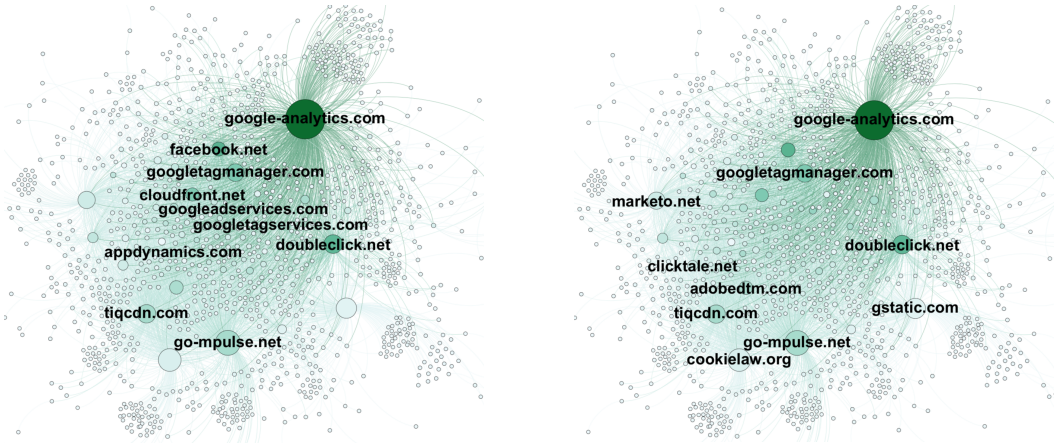

Figure 5.3 Directed graph representing the unique connections between source domains and third-party destination domains. The size of the nodes is proportional to their overall degree (i.e., number of connections). The color of nodes represents their number of unique incoming (left) and outgoing (right) connections, where the third-party domains act as a destination/source respectively.

are not the same, it indicates that the destination third party is "stealing" the cookie that was set by another party, possibly without its consent (i.e., this is a special case of cross-domain cookie sharing, in which the source domain did not initiate the sharing). We show the corresponding numbers in parentheses in Table 5.5. For example, our analysis reveals that google-analytics.com, which receives cookies set by 198 different parties, uses its own code to read and send the cookies of 187 of them. Similarly, the scripts loaded from doubleclick.net are responsible for triggering the sinks that send the cookies of 182 parties back to itself.

In total, we observe cross-domain cookie sharing in 5,635 out of the 9,772 (57.66%) websites that have external cookies. In particular, we detect 718 source third-party domains that have their external cookies shared, and 1,778 third-party destination domains that receive these shared cookies (correspondingly, for the "stolen cookies", i.e., a special case of cross-domain sharing, they are observed in 4,735 websites, or 48.45% of all websites that have external cookies; these stolen cookies are set by a total of 546 source domains, and leaked to 1,397 sink domains). Figure 5.3 shows the overall relationships among the domains engaged in cookie sharing, by representing the graphs of the incoming and outgoing connections between the various third parties. Interestingly, as can be seen in Figure 5.3, apart from a few Google-owned services that have both a high in-degree and out-degree (i.e., google-analytics.com, doubleclick.net), the other third parties have either a high in-degree (incoming connections from other parties, i.e., receiving external cookies' values) or a high out-degree (i.e., their cookies are shared to multiple destinations). Finally, in Figure 5.4 we present the distribution of the number of shared and stolen cookies per website, and in Figure 5.5 we plot the distribution of sink domains per website that correspond to the shared and stolen cookies.

5.4.3.3 Case Study: Google Analytics

We close this section by giving a deeper insight into the results presented previously in Section 5.4.3.2. Specifically, we manually examine cases regarding Google Analytics (GA), which as shown in Tables 5.3 and 5.5 (and Figure 5.3) represents the party whose external cookies are most frequently stolen, as well as being the party that receives the most external cookies set by other domains. Considering that Google owns several ad/tracking-related domains, such as `doubleclick.net`, `googletagservices.com`, etc, we further filter from our cross-domain sharing cases that involve `google-analytics.com`, but with a counterparty (i.e., receiving or stolen-from domain) that is clearly not Google property. We also focus our attention on the `_ga` cookie, which is a well-known cookie from Google Analytics that contains a per-site client ID [Gaca].

Note that while in the following we are able to confirm a third-party script *actively* reading from the `_ga` cookie and sending its value back to a third-party server, we do not know the purpose of such leaks, or whether this exchange of information is consented to by both parties, since we do not have visibility into the backend processing logic. Nevertheless, our analysis shows that external cookies do not only allow third parties to bypass the restrictions imposed on third-party HTTP cookies, but provide an easy way for third parties to exchange information *en masse* and cooperate, which endangers users' privacy beyond the extent of traditional third-party cookies: consider for example a third-party service that sets an external first-party cookie on a website that the user visits. That cookie would then be accessible to all other scripts, third-party or otherwise, executing in the first-party context. In that sense, all third parties that have their scripts in the page can access information provided by all other third parties, even without an explicit cooperation agreement between those third parties (as contrasted with traditional cookie-sharing agreements). We demonstrate this point more concretely below.

Adobe Demdex: We first focus on a case where external cookies set by Google Analytics are leaked to other third parties. We observed scripts from `adobedtm.com` sending the `_ga` cookie to `demdex.net`, which is a domain controlled by Adobe. The value of the `_ga` cookie is sent via `XMLHttpRequest` embedded in the body of a POST request (with the body being a URL-parameter-style string and the parameter name being `c_gacid` in this string, along with other parameters; the leading `GAx.x.` prefix is stripped from the value of the `_ga` cookie before it is sent, see Table 5.4 for an sample value of this cookie). An example of this leakage can be found on `http://www.uplus.co.kr` at the time of this writing. In our dataset, we found this cross-domain leakage of the `_ga` cookie to `demdex.net` in 17 of the top 10K websites. We manually analyzed the JS code responsible for sending the cookies, and found that the script, which is obfuscated, reads the `_ga` cookie at multiple places in its source code. Although to the best our knowledge we do not know whether Adobe and Google Analytics have tracking ID exchange agreements, this case illustrates the ease with which third parties can steal information from each other, without the consent of the affected parties.

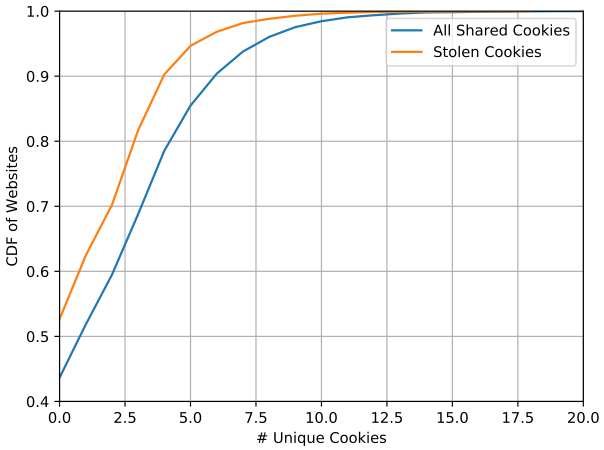

Figure 5.4 CDF of the number of both shared and stolen cookies per website of the Alexa top 10K.

GA Custom Dimensions: Here we focus on the case where external cookies set by other third parties are leaked to Google Analytics. In Table 5.5, we have shown that Google Analytics also receives external cookies that were set by 198 other third-party domains, and that in total Google Analytics uses its own code (that is, code served from `google-analytics.com`) to read cookies from 187 of them. We manually examined their corresponding sink objects, and found that the leaked external cookies containing unique tracking IDs, which were set by scripts served from domains that Google does *not* control, were being sent to `google-analytics.com` encoded in the HTTP request's URL as one of its `cdX` parameters, where `X` is a number. Upon further investigation, we found that this is due to a Google Analytics feature known as "custom dimensions and metrics" [Gacb], which can be enabled and configured by the website owners, and that the `cdX` parameters encode additional metrics (in this case external cookies set by other third parties) that Google Analytics does not automatically report by default. We point out that this usage scenario highlights the abusive nature of external cookies: website owners, instead of using their own infrastructure to record external tracking cookies set on their websites, abuse the custom dimensions feature to aggregate all external tracking cookies to another third party tracking provider, without the knowledge of the users and in doing so possibly also violating the terms of agreement with the third parties whose external cookies are reported to Google Analytics.

5.4.4 Fingerprinting

Finally, we address the limitation of our tracking ID detection algorithm at identifying tracking ID cookies whose values are always the same due to them being generated based on fingerprinting. As

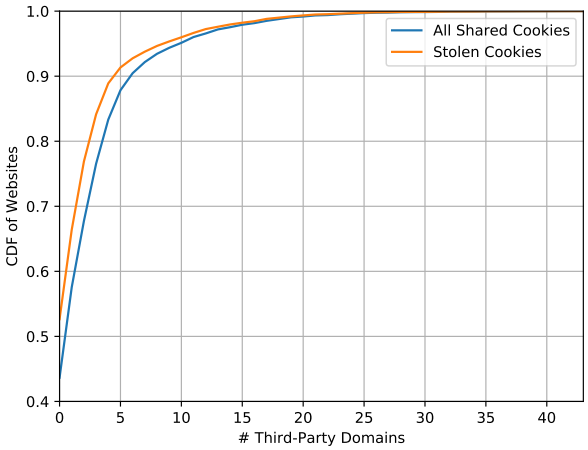

Figure 5.5 CDF of the number of sink domains per website corresponding to both the shared and stolen cookies encountered on each website.

Table 5.6 Manually identified fingerprinting-generated cookies.

Script Domain	Cookie Name	# Websites	Value
aralego.net	`ucfunnel_uid`	2	`88d501e0-40f3-3fcf-bf48-c8fa59bc7efd`
clarip.com	`c_uuid`	5	`250118664537365402840905373626008002 4`
futurecdn.net	`FTR_FingerPrint`	4	`6b2aad5d1452f2e65a0c1874aa09fee0`

mentioned in Section 5.3.3, this limitation stems from the fact that the ID detection algorithm relies on comparing the difference in values set for the same cookie on separate crawls. In Section 5.4.2.3, we manually found a cookie `ucfunnel_uid` whose value resembles a UID string (indeed the cookie's name suggests it is used as an ID cookie), and remains constant on our two separate crawls. In this section, we explore such cases in a more systematic manner, and attempt to quantify how many potentially UID-containing cookies that our ID detection algorithm missed. We remark that our purpose in this section is to establish that the results we report in this paper is not significantly impacted by the limitation of our ID detection algorithm, and we leave to future work the automatic identification of fingerprinting-generated UID cookies.

Our strategy to detecting fingerprinting-generated cookies involves: (i) we cluster the Alexa top 10K websites that we visited in the main crawl (when we visited each website using a fresh instance of Chromium) based on the names of the *non-session* cookies that are set on them, so that websites that have a cookie with the same name are placed in one cluster, and (ii) we set up our crawling infrastructure (see Section 5.3.5) to crawl the websites in the same cluster in a sequential order, one

after another, using the same browser instance (so the browser state is kept in between visits to different websites). We disable third-party cookies in this crawl to prevent trackers from using them to synchronize their UIDs across first-party boundaries (such that if any cookies are set with the same UID, then it is highly likely that the UID is generated from fingerprinting). Since the tracking ID detection algorithm does not work here, we do not need to collect control values for the cookies, and thus the websites in each cluster are visited only once and there is no distinction of main crawl and control crawl, as we had before. Note that the total number of clusters is the total number of cookie names that are common across websites (i.e., websites that have more than one cookie name in common will appear in more than one cluster). In our main crawl, there are 919 non-session unique *cookie names* that appear in more than one website, so in total we have 919 clusters for this crawl.

We find potentially fingerprinting-generated UIDs in the crawl result by looking for cookies whose value remains the same across websites in the same cluster. In total, we found 166. However, this does not indicate that all of these cookies contain tracking IDs: 119 of them do not meet the value length requirement of our tracking ID detection algorithm. We manually perused the values of the rest of the 47 cookies: the majority do not contain unique identifying information (e.g., only the geolocation, IP address, timestamps, and short strings similar to what we reported in Section 5.4.2.4). Nevertheless, we list in Table 5.6 the 3 cookies (including the previously mentioned `ucfunnel_uid`) that we found to be highly indicative of fingerprinting.

5.5 Conclusion

In this chapter, we examine a web tracking method that abuses first-party cookies that are set by third-party JavaScript code, which has been largely neglected by previous works. We refer to these third-party-originated first-party cookies as external cookies, and they are used for circumventing browser policies that seek to block traditional third-party cookies as well as facilitating mass exchange of tracking IDs. In order to measure how external cookies are being used for web tracking, we implemented dynamic taint analysis for cookies in the Chromium browser, and use it to analyze the Alexa top 10K websites. We show that external cookies are already widely used: they are encountered on 9,772 (97.72%) of the Alexa top 10K, and that 57.66% of these websites have third parties that exchange tracking IDs stored in external cookies. Our results clearly indicate that existing mitigation techniques, such as blocking third-party cookies and using filter lists such as EasyList/EasyPrivacy, are not adequate to protect users from third-party tracking, and thus the web needs additional countermeasures to protect users.

CHAPTER

6

FUTURE WORK AND CONCLUSION

6.1 Future Work

Although in this thesis we have demonstrated the feasibility of instrumenting the browser's JavaScript execution environment, and of using the proposed instrumentation to achieve important security and privacy goals such as measuring the current web ecosystem for privacy-abusive behaviors, we believe much more can be done in this area to protect the average web user against increasingly prying eyes. In the following, we summarize and propose an additional set of enhancements that can be pursued to extend the capabilities of each of our proposed systems.

6.1.1 Dynamic Data Flow Analysis for JavaScript

As we have shown in this thesis, dynamic data flow analysis is an important tool for understanding the behaviors of JavaScript code on the web, a proposition that we substantiated with two concrete use cases in terms of large-scale measurements of the current web ecosystem: one concerned with the detection of privacy-sensitive information leakage by browser extensions, and the other with the web tracking practice of using JavaScript-accessible first-party cookies. However, as noted in Chapter 3, our taint analysis engine was implemented based on an earlier compiler/interpreter model (i.e., Full-codegen and Crankshaft), which has gone out of favor in later versions of Chromium that adopts the Ignition/TurboFan duo. Nevertheless, since as we mentioned our proposed approach for implementing dynamic taint tracking is dependent more upon the parsing phase of the JavaScript

engine, which is largely stable across versions of V8 (and indeed across the two different compiler/interpreter model), and less on the actual machine/byte code generation phase, our approach can be adapted to new versions of the V8 JavaScript engine.

To this end, we propose that one direction for future work would be to adapt our approach for implementing Mystique to recent versions of V8 (and therefore the Chromium browser). Additionally, since for simplicity in the original Mystique prototype we disabled the optimizing compiler (i.e., Crankshaft), our taint propagation points are instrumented directly into the native code emitted by Full-codegen. In future adaptations of Mystique we suggest instrumenting the taint propagation points into either the Abstract Syntax Tree (AST) output of the parsing phase by extending the existing AST with new node types, or into the intermediate representation (IR) used by the target JavaScript engine, so that portability across different versions of the same JavaScript engine is improved. In the case of the V8 engine, given that the latest Ignition/TurboFan model shares the same bytecode (i.e., the IR used by V8) between the baseline compiler and the optimizing compiler, adopting our suggested approach would have the benefit of not needing to disable the optimizing compiler, as we did for the original Mystique prototype proposed in Chapter 3. Moreover, apart from better portability, this would also allow the performance overhead of dynamic taint tracking to be reduced, and contribute to making Mystique an attractive option to integrate into existing browsers to provide real-time monitoring of privacy leakage for the users.

Another worthwhile direction we would like to see in future efforts aimed at improving the capabilities of Mystique is the implementation of multi-color taint tracking, since our current prototype essentially implements only monochrome taint, i.e., a JavaScript object is either tainted or not. Multi-color taint tracking would allow for differentiating sources of information, and provide finer-grained understanding of how tainted objects are used by JavaScript. In Section 6.1.3 we give examples of how multi-color taint tracking would further improve our measurement of web tracking via external cookies, a topic we addressed in Chapter 5.

6.1.2 Real-Time Blocking of Privacy-Harming Script Behaviors

In the same vein of providing real-time privacy defense for the web users, the behavior signature scheme that we developed based on our PageGraph instrumentation, as we mentioned in Chapter 4, can also be adopted for real-time defense against known privacy-harming, web tracking scripts, when such scripts are either inlined/bundled, or when the filter lists are not update-to-date. However, as noted there, the challenges arise primarily out of the fact that by the time a complete signature of a known harmful behavior is observed, the privacy harm has already been done. The ramifications of the caused privacy harm are associated with the type of privacy-sensitive events that the signature describes (recall that we defined two privacy-sensitive events): (i) for storage API accesses, this means that the tracking script has been able to potentially persist a unique user identifier in the

storage mechanisms provided by the browser, e.g. *localStorage*, and (ii) for network requests, the privacy harm is that potentially uniquely identifying information about the user, along with their privacy-sensitive data, have been sent to web tracking providers.

As already noted in Chapter 4, we suggest future work in this area adopt different approaches that are most effective at addressing the corresponding privacy-sensitive events. For example, for signatures related to storage writes, one possible approach would be to maintain a shadow storage layer where the values persisted by JavaScript are first written to, and contents in the shadow storage are committed to the underlying storage mechanisms only when the complete event-loop signature of that write operation does not match signatures from known-bad scripts. We believe that comprehensively handling all privacy-sensitive event types captured by the behavior signatures, while maintaining good web compatibility without degrading user experience, is a worthwhile direction for future research in this domain.

6.1.3 Further Understanding of Web Tracking from Third-Party JavaScript

In Chapter 5, we focused our attention on first-party cookies as the means by which third-party JavaScript code can persist information (e.g., user identifiers) between browser sessions. This was motivated primarily by (i) previous studies focused on third-party cookies and overlooked how first-party cookies can be misused, and (ii) through some preliminary experiments we have observed a widespread use of practices that involve first party cookies, where traditionally those entities should have used third party cookies. However, for future research in this direction it would be interesting to know whether other storage mechanisms, such `localStorage`, are being used as well, how often each storage mechanism is used, and for what purpose.

In our work presented in Chapter 5, we relied on a heuristic that matches sink objects (i.e., JavaScript values that trigger a taint sink) against the recorded external cookies to identify relationships between entities that receive information derived from external cookies, and the entities that are responsible for setting them in the first place. This is primarily due to the fact that currently Mystique supports only monochrome taint. On the other hand, if a multi-color taint engine is available (a direction for future work that we mentioned in Section 6.1.1), then this would allow for more precise identification of the relationships between the setter/receiver entities, e.g., by labelling external cookies set by different origins a unique color.

Finally, cross-site tracking via link decoration is another potential area for future research that aim to understand tracking behaviors from third-party JavaScript code. Link decoration is a technique whereby tracking IDs are embedded in the URLs on a website that point to external sites. Thus, if the same tracking company has JavaScript code that runs in both the origin and destination websites (e.g., a social network widget), then uniquely identifying information about a user can be passed cross-site, even if third-party cookies are disabled. Indeed, cross-site tracking via link

decoration has seen increased usage in response to Safari's Intelligent Tracking Prevention (ITP) which forbids cookies from being set for domains classified as a web tracker [Web20]. This is another area where a multi-color taint engine would be useful: a measurement system could assign one color to JavaScript-accessible data sources that give the URL of the current page, and another color to first-party cookies that are derived from the page URL (which can be identified by their taint color). We believe that such a system would allow for better understanding of the prevalence and the major players involved in link-decoration-based tracking.

6.2 Conclusion

In this dissertation, we observed that the privacy implications of the current web ecosystem primarily stem from the fact that JavaScript code from multiple third-party entities, who often have conflicting interests, are included and executed in the same privileged context of the visited website. We then proposed that privacy-abusing JavaScript behaviors can be measured and understood by instrumenting the JavaScript execution environment of the browser, and that such instrumentation will have benefit not only for web measurement, but also for tool development that will ultimately improve user privacy on the web.

We evaluated the merit of this proposal from three different aspects of the current web ecosystem, namely (i) the privacy practice of JavaScript code injected by browser extensions, and whether they leak sensitive information such as the user's browsing history, (ii) filter list evasion that results from the ability of web trackers to arbitrarily move, inline, or otherwise bundle their privacy-harming code in order to avoid being blocked, and (iii) the ability of tracking providers to run JavaScript code in the first-party context and therefore can use first-party cookies to circumvent browser policies that block third-party cookies. We approached each of these aspects by first proposing the appropriate browser instrumentations, and then presented their evaluation in the context of the current web ecosystem. We summarize each of our works below.

In the first work, we proposed and implemented a dynamic taint analysis engine for JavaScript in the Chrome browser's context, by modifying both its V8 JavaScript engine and its WebKit/Blink layout engine to be taint-aware. Our taint engine is generic, in that it is able to propagate taint across all available JavaScript data types, as well as being able to handle control-flow dependencies. We then applied it to uncover privacy-abusing behaviors of browser extensions, shedding light on the extent to which browser extensions are impacting user privacy today. We also presented case studies of privacy-abusing extensions that were detected by our system but would have been missed by similar previous research works. Besides measurement, we also demonstrated the effectiveness of our analysis system as a triage tool for online extension repositories such as the Chrome Web Store, by providing a web interface[1] where users can upload extensions to be analyzed by our system and

[1] `https://mystique.csc.ncsu.edu`

get back the results.

In the second work, we took another approach to instrumenting the browser and its JavaScript execution environment. Instead of dynamic taint tracking, this instrumentation aims to record detailed actions taken by JavaScript code during runtime into a graph structure that extends the existing notion of the DOM maintained by the browser's rendering engine. This instrumentation was motivated by the observation of the ongoing arms race between efforts to block online advertising and tracking, and attempts by companies whose revenues depend on such tracking to evade blocking. The latest iteration of this arms race sees web trackers taking code that is blocked by crowd-source filter lists and moving them to new domains/URLs, inlining them on websites, or bundling with other code that provides benign functionalities. With our proposed instrumentation, we built robust signatures of behaviors exhibited by known-bad scripts, and used these signatures to uncover instances of filter list blocking in the Alexa top 100K websites, finding an additional 3,589 unique scripts that the current filter lists failed to block. We also provided detailed taxonomy and case studies from our results.

In the third work, we demonstrated another area where applying the taint analysis engine that we built would prove fruitful. In this work, we studied the practice where web tracking is done via *first-party* cookies that are set by third-party JavaScript code, which we referred to as *external cookies*. We further extended the capabilities of our taint analysis engine to precisely track byte-level taint status for the string type, in order to differentiate external cookies from other traditional first-party cookies. The key contributions of this work also include: (i) a systematic method to automatically filter for external cookies that contain a unique tracking identifier, and (ii) a heuristic that matches together and sheds light on the relationship between entities originally set external cookies containing tracking IDs, and the entities that receive tracking information derived from such cookies. We used our system to crawl the Alexa top 10K websites and found that 97.75% of the websites have external cookies set on them, and that out of the 26,632 unique external cookies that were observed in the Alexa top 100K, 4,212 contain unique tracking IDs and can be used for persistent tracking of the user. We used our tracking ID detection algorithm to provide a complete taxonomy of the external cookies that we collected from our crawl of the Alexa top 10K, and our heuristic matching algorithm revealed a complex network of connections between setter and receivers of external cookies.

The browser instrumentations and large-scale studies presented in this dissertation hopefully prove that the privacy implications of the current web can be effectively measured and understood, and that the insights gained from this endeavour will help future efforts to improve privacy protection for the average web user.

BIBLIOGRAPHY

[Adb] *Adblock Plus*. `https://adblockplus.org/`. 2020.

[Afr18] Afroz, S. et al. "Exploring server-side blocking of regions". *arXiv preprint arXiv:1805.11606* (2018).

[Agt12] Agten, P. et al. "JSand: complete client-side sandboxing of third-party JavaScript without browser modifications". *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM. 2012, pp. 1–10.

[Ale] Alexa. *How are Alexa's traffic rankings determined?* `https://support.alexa.com/hc/en-us/articles/200449744-How-are-Alexa-s-traffic-rankings-determined-`.

[Bac] *Background Pages*. `https://developer.chrome.com/extensions/background_pages`. 2018.

[Ban10] Bandhakavi, S. et al. "VEX: Vetting Browser Extensions for Security Vulnerabilities." *USENIX Security Symposium*. Vol. 10. 2010, pp. 339–354.

[Bar10] Barth, A. et al. "Protecting Browsers from Extension Vulnerabilities." *NDSS*. 2010.

[Bau15] Bauer, L. et al. "Run-time Monitoring and Formal Analysis of Information Flows in Chromium." *NDSS*. 2015.

[Chra] *Blink*. `https://www.chromium.org/blink`. 2018.

[Blo] Blog, C. *Trustworthy Chrome Extensions, by default*. `https://blog.chromium.org/2018/10/trustworthy-chrome-extensions-by-default.html`.

[Bra20a] Brave. *PageGraph*. `https://github.com/brave/brave-browser/wiki/PageGraph`. 2020.

[Bra20b] Brave. *Secure, Fast & Private Web Browser with Adblocker*. `https://brave.com/`. 2020.

[Buk92] Bukh, P. N. D. *The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling*. 1992.

[Cat18] Catapult Project. *Web Page Replay*. `https://github.com/catapult-project/catapult/tree/master/web_page_replay_go`. 2018.

[Che18] Chen, Q. & Kapravelos, A. "Mystique: Uncovering Information Leakage from Browser Extensions". *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: ACM, 2018, pp. 1687–1700.

[Che21] Chen, Q. et al. “Detecting Filter List Evasion With Event-Loop-Turn Granularity JavaScript Signatures”. *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2021.

[Chrb] *Chrome - chrome.cookies*. `https://developer.chrome.com/extensions/cookies`. 2019.

[Con] *Chrome - Content Scripts*. `https://developer.chrome.com/extensions/content_scripts`. 2019.

[Chr18] Chrome Web Store. *Web of Trust*. `https://chrome.google.com/webstore/detail/wot-web-of-trust-website/bhmmomiinigofkjcapegjjndpbikblnp?hl=en-US`. 2018.

[Chrc] *Chromedriver - Webdriver for chrome*. `https://sites.google.com/a/chromium.org/chromedriver/`. 2019.

[Chrd] *chrome.storage*. `https://developer.chrome.com/extensions/storage`. 2018.

[Chre] *chrome.tabs*. `https://developer.chrome.com/extensions/tabs`. 2018.

[Chu15] Chudnov, A. & Naumann, D. A. “Inlined information flow monitoring for JavaScript”. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 629–643.

[Con18] Conrod, J. *A tour of V8: Crankshaft, the optimizing compiler*. `http://www.jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler`. 2018.

[Gaca] *Cookies and User Identification*. `https://developers.google.com/analytics/devguides/collection/analyticsjs/cookies-user-id`. 2020.

[Gacb] *Custom dimensions & metrics*. `https://support.google.com/analytics/answer/2709828?hl=en`. 2020.

[(DA09] (DAA), D. A. A. *Self Regulatory Principles for Online Behavioral Advertising*. `https://digitaladvertisingalliance.org/sites/aboutads/files/DAA_files/seven-principles-07-01-09.pdf`. 2009.

[DG12] De Groef, W. et al. “FlowFox: a web browser with flexible and precise information flow control”. *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 748–759.

[Det18] Detectify Labs. *Chrome Extensions - AKA Total Absence of Privacy*. `https://labs.detectify.com/2015/11/19/chrome-extensions-aka-total-absence-of-privacy/`. 2018.

[Dha09] Dhawan, M. & Ganapathy, V. “Analyzing information flow in JavaScript-based browser extensions”. *Computer Security Applications Conference, 2009. ACSAC’09. Annual*. IEEE. 2009, pp. 382–391.

[Disa] *Disconnect*. `https://disconnect.me/`.

[Dje10] Djeric, V. & Goel, A. “Securing script-based extensibility in web browsers”. *Proceedings of the USENIX Security Symposium*. 2010.

[Doc] *Docker*. `https://www.docker.com/`. 2019.

[Ele] *EasyList*. `https://easylist.to/`. 2020.

[EFF] EFF. *Privacy Badger*. `https://www.eff.org/privacybadger`.

[V8e] *Embedder’s Guide*. `https://github.com/v8/v8/wiki/Embedder’s-Guide#contexts`. 2018.

[Enc10] Enck, W. et al. “TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones”. *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. 2010.

[Eng16] Englehardt, S. & Narayanan, A. “Online Tracking: A 1-million-site Measurement and Analysis”. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, 2016, pp. 1388–1401.

[Eng15] Englehardt, S. et al. “Cookies that give you away: The surveillance implications of web tracking”. *Proceedings of the 24th International Conference on World Wide Web*. 2015, pp. 289–299.

[Fas19] Fass, A. et al. “HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs”. 2019.

[Fou18] Fouad, I. et al. *Tracking the Pixels: Detecting Web Trackers via Analyzing Invisible Pixels*. 2018. arXiv: `1812.01514`.

[Fra18] Franken, G. et al. “Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies”. *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 151–168.

[Gar17] Garimella, K. et al. “Ad-blocking: A study on performance, privacy and counter-measures”. *Proceedings of the 2017 ACM on Web Science Conference*. ACM. 2017, pp. 259–262.

[Gho] *Ghostery*. `https://www.ghostery.com/`.

[Giu12] Giuffrida, C. et al. “Memoirs of a browser: A cross-browser detection model for privacy-breaching extensions”. *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM. 2012, pp. 10–11.

[Goo18] Google. *Chrome V8*. `https://developers.google.com/v8/`. 2018.

[Gug15] Gugelmann, D. et al. “An automated approach for complementing ad blockers’ blacklists”. *Proceedings of the Symposium on Privacy Enhancing Technologies (PETS)* (2015).

[Guh11] Guha, A. et al. “Verified security for browser extensions”. *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE. 2011, pp. 115–130.

[Hil] Hill, R. *uBlock Origin*. `https://github.com/gorhill/uBlock`.

[How18] How-to Geek. *Warning: Your Browser Extensions Are Spying On You*. `https://www.howtogeek.com/180175/warning-your-browser-extensions-are-spying-on-you/`. 2018.

[Chrf] *How to step through your code*. `https://developers.google.com/web/tools/chrome-devtools/javascript/step-code`. 2020.

[Ikr17] Ikram, M. et al. “Towards seamless tracking-free web: Improved detection of trackers via one-class learning”. *Proceedings of the Symposium on Privacy Enhancing Technologies (PETS)* (2017).

[Inv16] Invernizzi, L. et al. “Cloak of visibility: detecting when machines browse a different web”. *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 743–758.

[Iqb20] Iqbal, U. et al. “Adgraph: A graph-based approach to ad and tracker blocking”. *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2020.

[Jan10] Jang, D. et al. “An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications”. *Proc of the 17th ACM Conf on Computer and Communications Security CCS*. Vol. 10. 2010, pp. 43–51.

[Chrg] *JavaScript APIs*. `https://developer.chrome.com/extensions/api_index`. 2018.

[Cra] *jQuery Prepending Protocol*. `https://github.com/jquery/jquery/blob/2d4f53416e5f74fa98e0c1d66b6f3c285a12f0ce/test/data/jquery-1.9.1.js#L8088`. 2018.

[Mdna] *JSON.stringify()*. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify`. 2020.

[Jue19a] Jueckstock, J. & Kapravelos, A. “VisibleV8: In-browser Monitoring of JavaScript in the Wild”. *Proceedings of the ACM Internet Measurement Conference (IMC)*. 2019.

[Jue19b] Jueckstock, J. et al. “The Blind Men and the Internet: Multi-Vantage Point Web Measurements”. *arXiv preprint arXiv:1905.08767* (2019).

[Kap13] Kapravelos, A. et al. “Revolver: An Automated Approach to the Detection of Evasive Web-based Malware”. *Proceedings of the USENIX Security Symposium*. 2013.

[Kap14] Kapravelos, A. et al. “Hulk: Eliciting malicious behavior in browser extensions”. *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 641–654.

[Kri09] Krishnamurthy, B. & Wills, C. “Privacy diffusion on the web: a longitudinal perspective”. *Proceedings of the 18th international conference on World wide web*. ACM. 2009, pp. 541–550.

[Kub] *Kubernetes - Production-Grade Container Orchestration*. `https://kubernetes.io/`. 2019.

[Lau17] Lauinger, T. et al. “Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web”. 2017.

[Lek13] Lekies, S. et al. “25 Million Flows Later - Large-scale Detection of DOM-based XSS”. *20th ACM Conference on Computer and Communications Security Berlin 4.11.2013*. 2013.

[Lek15] Lekies, S. et al. “The Unexpected Dangers of Dynamic JavaScript”. *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015, pp. 723–735.

[Li18] Li, B. et al. “JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions.” *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2018.

[Li12] Li, Z. et al. “Knowing your enemy: understanding and detecting malicious web advertising”. *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 674–686.

[Li07] Li, Z. et al. “SpyShield: Preserving privacy from spy add-ons”. *Recent Advances in Intrusion Detection*. Springer. 2007, pp. 296–316.

[May12] Mayer, J. R. & Mitchell, J. C. “Third-party web tracking: Policy and technology”. *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 413–427.

[Mel18] Melicher, W. et al. “Riding out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting” (2018).

[Chrh] *Message Passing*. `https://developer.chrome.com/extensions/messaging`. 2018.

[Mey10] Meyerovich, L. A. & Livshits, B. “ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser”. *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 481–496.

[Mir18] Miroglio, B. et al. “The effect of ad blocking on user engagement with the web”. *Proceedings of the 2018 World Wide Web Conference*. International World Wide Web Conferences Steering Committee. 2018, pp. 813–821.

[Mit] *Mitmproxy Project*. `https://mitmproxy.org/`. 2019.

[Moz] Mozilla. *webRequest.filterResponseData() - Mozilla | MDN*. `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/filterResponseData`.

[Moz18] Mozilla. *Modifying a web page*. `https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Modify_a_web_page`. 2018.

[Moz18] Mozilla. *SpiderMonkey*. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`. 2018.

[Moz20] Mozilla. *Event loop*. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Event_loop`. 2020.

[Fira] *Mozilla - Firefox 63 Lets Users Block Tracking Cookies*. `https://blog.mozilla.org/security/2018/10/23/firefox-63-lets-users-block-tracking-cookies/`. 2018.

[Nik12] Nikiforakis, N. et al. “You are what you include: large-scale evaluation of remote javascript inclusions”. *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 736–747.

[Pan15] Pan, X. et al. “I do not know what you visited last summer: Protecting users from third-party web tracking with trackingfree browser”. *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*. 2015.

[PC 18] PC Magazine. *’Web Of Trust’ Browser Extension Cannot Be Trusted*. `http://www.pcmag.com/news/349328/web-of-trust-browser-extension-cannot-be-trusted`. 2018.

[Firb] *PCMag - Firefox 22 to Disable Third-Party Cookies by Default*. `https://www.pcmag.com/news/308420/firefox-22-to-disable-third-party-cookies-by-default`. 2013.

[Pol11] Politz, J. G. et al. “ADsafety: Type-based Verification of JavaScript Sandboxing”. *Proceedings of the 20th USENIX Conference on Security*. SEC’11. San Francisco, CA: USENIX Association, 2011, pp. 12–12.

[Puj15] Pujol, E. et al. “Annoyed users: Ads and ad-block usage in the wild”. *Proceedings of the 2015 Internet Measurement Conference*. ACM. 2015, pp. 93–106.

[Red] *Redis*. `https://redis.io/`. 2019.

[Roe12] Roesner, F. et al. “Detecting and defending against third-party tracking on the web”. *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 12–12.

[Roy07] Roy, C. K. & Cordy, J. R. “A Survey on Software Clone Detection Research”. *Queen's School of Computing, Technical Report* (2007).

[SR21] Sanchez-Rola, I. et al. “Journey to the Center of the Cookie Ecosystem: Unraveling Actors' Roles and Relationships”. *IEEE Symposium on Security and Privacy*. IEEE. 2021.

[Sax10] Saxena, P. et al. “A symbolic execution framework for JavaScript”. *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE. 2010, pp. 513–528.

[Sem] *Semantic Signatures*. `https://github.com/semantic-signatures/semantic-signatures`. 2020.

[Sen] *Sentinel - the artificial intelligence ad detector*. `https://adblock.ai/`.

[Shu18] Shuba, A. et al. “NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking”. *Proceedings of the Symposium on Privacy Enhancing Technologies (PETS)* (2018).

[Sko19] Skolka, P. et al. “Anything to Hide? Studying Minified and Obfuscated Code in the Web”. *The World Wide Web Conference*. 2019, pp. 1735–1746.

[Sny16] Snyder, P. et al. “Browser feature usage on the modern web”. *Proceedings of the 2016 Internet Measurement Conference*. ACM. 2016, pp. 97–110.

[Sny17] Snyder, P. et al. “Most websites don't need to vibrate: A cost-benefit approach to improving browser security”. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 179–194.

[Sta17] Starov, O. & Nikiforakis, N. “Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions”. *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2017, pp. 1481–1490.

[Sta19] StatCounter. *Browser Market Share Worldwide*. `http://gs.statcounter.com/browser-market-share`. 2019.

[Sto14] Stock, B. et al. “Precise client-side protection against DOM-based Cross-Site Scripting”. *USENIX Security Symposium*. 2014.

[Sto17] Storey, G. et al. “The future of ad blocking: An analytical framework and new techniques”. *arXiv preprint arXiv:1705.08568* (2017).

[Disb] *Tracking protection lists*. `https://disconnect.me/trackerprotection`. 2020.

[Tra19] Tramèr, F. et al. “AdVersarial: Defeating Perceptual Ad Blocking”. *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2019.

[Tsc18] Tschantz, M. C. et al. “A bestiary of blocking: The motivations and modes behind website unavailability”. *8th USENIX Workshop on Free and Open Communications on the Internet (FOCI 18)*. 2018.

[Mdnb] *Using HTTP cookies*. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies`. 2020.

[V8 18] V8 Project. *Launching Ignition and TurboFan*. `https://v8project.blogspot.com/2017/05/launching-ignition-and-turbofan.html`. 2018.

[V8b] *V8BindingDesign.md*. `https://cs.chromium.org/chromium/src/third_party/WebKit/Source/bindings/core/v8/V8BindingDesign.md`. 2018.

[VA11] Van Acker, S. et al. “WebJail: least-privilege integration of third-party components in web mashups”. *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM. 2011, pp. 307–316.

[Vog07] Vogt, P. et al. “Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.” *NDSS*. Vol. 2007. 2007, p. 12.

[Wan12] Wang, J. et al. “An empirical study of dangerous behaviors in firefox extensions”. *Information Security* (2012), pp. 188–203.

[Jav] *Web Technology Surveys - Usage of JavaScript for websites*. `https://w3techs.com/technologies/details/cp-javascript/all/all`. 2020.

[Web20] WebKit. *Intelligent Tracking Prevention*. `https://webkit.org/blog/7675/intelligent-tracking-prevention/`. 2020.

[Web18] WebProNews. *The Chromium-Powered Opera Is Finally Here*. `https://www.webpronews.com/the-chromium-powered-opera-is-finally-here/`. 2018.

[Wei11] Weinberg, Z. et al. “I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks”. *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE. 2011, pp. 147–161.

[Wei18] Weissbacher, M. *These Chrome extensions spy on 8 million users*. http://mweissbacher.com/blog/2016/03/31/these-chrome-extensions-spy-on-8-million-users/. 2018.

[Wei17] Weissbacher, M. et al. “Ex-Ray: Detection of History-Leaking Browser Extensions”. *Annual Computer Security Applications Conference*. ACM publishing. 2017.

[Chri] *What are extensions?* https://developer.chrome.com/extensions. 2018.

[Gacc] *What are the values in _ga cookie?* https://stackoverflow.com/questions/16102436/what-are-the-values-in-ga-cookie. 2020.

[Yue09] Yue, C. & Wang, H. “Characterizing insecure javascript practices on the web”. *Proceedings of the 18th international conference on World wide web*. ACM. 2009, pp. 961–970.