Principled Selective Ad Blocking with Taint-based Exfiltration Protection Policies

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science (Computer Science)

by

Ethan Lowman

May 2019

Abstract

We start from the premise that it is a legitimate business model for publishers to provide content in exchange for expecting readers to view advertisements. This implicit agreement, however, should not require privacy and security compromises, which many content consumers identify as a justification for aggressive ad blocking. Rather than escalate the arms race between ad blockers and publishers, we address the privacy and security issues of the advertising ecosystem. In particular, we address the risk of user data exfiltration in the browser by third-party advertising and analytics scripts. We construct a conservative exfiltration prevention policy based on taint checking. To test the feasibility of our approach, we conduct a large-scale web scan using our novel JavaScript dynamic analysis tool, which instruments embedded scripts. We find that many scripts violate our strict anti-exfiltration policy, but show how the most popular advertising and analytics scripts could be redesigned to comply with a strict security policy. This motivates a principled selective ad blocker which ensures privacy, while supporting the business model of advertiser-supported content. Whereas leading selective ad blockers are based on vague usability requirements set by parties with financial stake in which ads are whitelisted (a clear conflict of interest), our selective ad blocking policy would allow users to benefit from the security properties of ad blockers while allowing publishers to profit from privacy-preserving ads.

Approval Sheet

This thesis is submitted in partial fulfillment of the requirements for the degree of

Master of Science (Computer Science)

Ethan Lowman

Ethan Lowman

This thesis has been read and approved by the Examining Committee:

[illegible]

David Evans, Advisor and Committee Chair

[illegible]

Yuan Tian

[illegible]

Yonghwi Kwon

Accepted for the School of Engineering and Applied Science:

[illegible]

Craig H. Benson, Dean, School of Engineering and Applied Science

May 2019

To my many advisors and mentors.

Contents

List of Figures

Chapter 1

Introduction

Advertising and analytics scripts are typically hosted on servers operated by ad networks and embedded directly into web pages. This method is convenient for both ad networks and web publishers, since ad networks are able to deploy updates without requiring publishers to manually upgrade local copies. However, by using this technique, a site that embeds an ad is implicitly trusting the ad provider with all the sensitive data on that site.

Browsers execute third-party JavaScript directly included from a remote URL with the same browser-side privileges as first-party scripts. This is both a valuable feature and a major security liability. Using third-party scripts allows developers to add rich functionality to their websites (like ads or analytics). Loading scripts from a third-party server instead of self-hosting allows for cost savings on bandwidth and faster load times for users, since popular scripts are usually cached in the browser. However, if the provider of the script (e.g., a content delivery network or the ad network itself) is dishonest or compromised, it only takes a malicious update to a popular script to obtain unfettered client-side access to all the sites that use the script.

The costs of such a widespread attack can be devastating. The simplest attacks include reading and exfiltrating user data, including session credentials, from the sites that include the compromised ad script. Attackers also have opportunities for lateral movement from the host site. For example, an attacker could leverage the credibility of the host site to phish credentials for a different site, distribute malware, or exploit browser vulnerabilities to gain access to users' machines.

These attacks are not just theoretical. For example, in 2016, the websites of the New York Times, BBC, AOL, and NFL were compromised by an attack on an ad network and were used to distribute malware and ransomware [1]. Another recent study [2] found that the AdThink advertising platform was stealing user emails from autofilled login forms.

Some security-conscious users have adopted ad blocking software to address these privacy and security concerns, but ad blocking software has primarily gained its popularity due to the disruptive user experience of many ads. In 2014, 15.7 percent of U.S. internet users blocked ads on at least one device (including mobile devices). In 2018, that figure grew to 25.2 percent [3]. Widespread ad blocking, however, threatens the sustainability of many websites. According to one study, in 2017 the worldwide market value of display ads was an estimated 100 billion USD, and out of this amount, an estimated 42 billion USD was lost due to ad blocking [4]. These conditions have led to a new arms race of ad detection and obfuscation. In an effort to deescalate this arms race, selective ad blockers have emerged. Standards for what constitutes a "high-quality ad" have been developed, and selective ad blockers whitelist ads satisfying these standards. However, these standards are often subjective and the ad blockers often have financial stake in the interpretation of the criteria [5, 6], so this model leads to conflicts of interest.

Securing third-party JavaScript is a well-studied topic, but prior work has failed to produce practical mechanisms for controlling the risk of user data exfiltration. Proposed mechanisms [7, 8] for securing third-party advertisement scripts, in particular, suffer compatibility issues with many types of ads. Though universally blocking advertising and analytics scripts is a blunt and effective way to secure third party advertisements, we seek a compromise of security and utility that protects user privacy and publisher profitability. We aim to prevent malicious exfiltration in third-party advertisement and analytics scripts using a selective ad blocker with a conservative anti-exfiltration policy based on taint checking. Ultimately, we see such principled selective ad blockers as a solution to the ad blocker arms race, and an economically powerful force that could be leveraged to improve the security landscape of the advertising and analytics ecosystem.

Contributions. In Chapter 3 we introduce two variants of an anti-exfiltration policy based on taint checking. We design these policies with the goal of preserving the utility of most websites while providing substantial protection from malicious exfiltration of sensitive data.

In Chapter 4 we introduce a novel method of JavaScript instrumentation based on the Chrome DevTools Protocol, which allows us to test our policies on real-world websites. In contrast to existing JavaScript and browser instrumentation tools, which are typically intended for performance evaluation, our instrumentation approach provides simultaneous visibility of browser events and JavaScript events, and enables manipulation of the JavaScript runtime for rich prototyping. This tool has a wide variety of applications for researchers beyond this study. Many new browser features, which would ordinarily by costly and time consuming to prototype in large browser code bases, can be quickly prototyped and evaluating using our instrumentation tool.

In Chapter 5, we use our instrumentation tool in a large-scale web scan to analyze how both of our policies perform in the real world. We find that the most popular advertising and analytics scripts currently violate our strict anti-exfiltration policies. However, in Chapter 6, we show techniques that can be used to make benign advertising and analytics scripts comply with the anti-exfiltration policies. We estimate that if the top three advertising and analytics networks alone adopted these techniques, about 62% of the most popular web pages would be policy-compliant.

Chapter 2

Background

In this section, we provide background on embedded web scripts and ad blockers, and put our research in the context of related work.

2.1 Embedded Web Scripts

Web browser security is built on the same-origin policy [9], which provides isolation between the resources on different websites. Under the same-origin policy, a web page can only access a resource if the web page and the resource have matching *origins*. An origin is defined as the concatenation of a URL's scheme, host name, and port number (if specified). For instance, the web page https://advertiser.net/index.html cannot directly access data associated with https://example.com/example.html since the origins https://advertiser.net and https://example.com do not match.

However, it is a common practice to execute untrusted third-party JavaScript, like advertising or analytics scripts, in a document's top-level origin. For example, if https://example.com/ embeds a remote script from https://advertiser.net/, then the remote script executes with the origin https://example.com and has full access to resources for https://example.com which are otherwise protected by the same-origin policy. For example, https://advertiser.net/ could serve a script to https://example.com that reads and exfiltrates cookies for https://example.com.

Nikiforakis et al. [10] conducted a large-scale evaluation of remote JavaScript inclusions and found that about 70% of the Alexa top 10,000 websites include JavaScript from more than five unique remote hosts, with some of the top sites including JavaScript from hundreds of remote hosts. The authors also conducted a study of remote JavaScript inclusions over a 10 year period and found that the Alexa top 10,000 websites included an average of 2.1 new domains in the period of 2009-2010. Furthermore, the authors found that the most popular remote scripts come from only a handful of sources. Google alone is responsible for serving five of the top ten remote scripts among the Alexa top 10,000 sites, and the Google Analytics script is present on nearly 70% of these sites. Compromising this single script could enable client-side attacks against many of the most popular websites.

Web developers have a number of security features at their disposal that promise to help secure third-party script execution. For example Content Security Policy (CSP) [11] can be used to whitelist allowable scripts URLs. However, third parties can trivially serve a malicious script at a whitelisted URL. CSP or Subresource Integrity can be used to verify a script's hash before executing, but identifying a trusted hash by manual auditing is usually not feasible. Third-party scripts are often heavily obfuscated and "minified," or transformed to minimize file size. These transformations make large scripts very difficult to reverse engineer and audit. Additionally, ad networks may release new versions of a script without warning, so a benign update would be blocked due to a hash mismatch and require manual re-auditing.

Though Content Security Policy has widespread browser support, it has struggled to gain traction on the internet at large. Patil et al. [12] found in 2016 that only 27 of the Alexa top 100,000 websites used CSP, and only 20 of those sites actually enforced their policies. The remaining 7 sites operated in report-only mode. In 2016, Weichselbaum et al. [13] conducted an Internet-wide analysis of CSP use on approximately 100 billion pages from over 1 billion hostnames, and found CSP deployments on less than 2 million hosts with only 25,011 unique policies. Where CSP is used, it is often used improperly, giving a false sense of security. Weichselbaum et al. [13] found that 99.34% of unique CSP policies deployed offered no protection against XSS due to misconfiguration or whitelisting of web endpoints or scripts which contained CSP bypass vulnerabilities.

Browser features have failed to properly address the risk of exfiltration by third-party scripts since they primarily target cross-site scripting (XSS) and similar attacks. XSS attacks are one of the most common threats to confidentiality on the web: a script is maliciously injected into a victim page where it executes and exfiltrates sensitive data. Security features that protect against XSS are typically designed to prevent the injection step, whether by avoiding the injection vulnerability completely (e.g., by sanitizing input), or by preventing the injected script from running (e.g., with CSP). However, in a third-party script exfiltration attack, the victim site voluntarily injects the script. Thus, XSS-oriented browser features are of limited use for exfiltration attacks.

Acker et al. [14] show how CSP in particular has failed to address risk of exfiltration by neglecting some types of network requests which can be used as exfiltration channels. Additionally, content security policies operate on the destination of network requests, not their contents, and Chen et al. [15] show that browser-enforced exfiltration protection policies based on the destination of network requests are insecure, since it is often possible to publicly leak information through whitelisted channels (an attack called self-exfiltration).

A number of approaches to securely executing third-party scripts have emerged, but many of these are hard to deploy or do not support a full JavaScript feature set.

TreeHouse [16] is a JavaScript sandbox which uses Web Workers for isolation. The sandbox has a configurable policy that gives web developers fine-grained control over the execution of the scripts they include.

AdJail [8] specifically targets securing advertisements. Advertisement scripts run in iframe sandboxes [17] with an unprivileged origin, and DOM artifacts (e.g., ad images and formatting) are mirrored out of the sandbox. More complex functionality is implemented by passing messages between the host frame and the sandbox frame. A custom policy mediates access to privileged resources.

JavaScript in JavaScript [18] is a JavaScript runtime implemented in client-side JavaScript. Third party scripts executing in this runtime are isolated from the host runtime, and calls to privileged methods in the host runtime are mediated through a custom policy. This approach has significant performance overhead, since JavaScript in the sandbox runs twice as slow as native JavaScript.

TreeHouse, AdJail, and JavaScript in JavaScript are all defenses which rely on developers to specify custom policies. However, experience from CSP suggests that it is very difficult to get site owners to provide effective policies [12].

Caja [19] and AdSafe [7] take a completely different approach, employing server-side rewriting to transform scripts into safe versions that can be directly embedded. These approaches are hard to deploy since they require a script transformation server, and they do not support the full JavaScript feature set. For example, AdSafe does not allow access to Date or Math.random since their non-deterministic behavior is hard to statically analyze. More importantly, since their security models are not based on in-browser security primitives, mismatches between the server-side and client-side security models have led to many vulnerabilities in the past [20].

Akhawe et al. [20] introduced a primitive called a data-confined sandboxes (DCS), based on iframe sandboxes. postMessage calls are used to mediate interaction between the child iframe sandbox and the parent page. Data-confined sandboxes are the most promising approach to secure script execution since they are based on in-browser security primitives and require a minimal trusted compute base (TCB) for the mediation layer.

In this study, we introduce a mechanism to mitigate risk of malicious exfiltration by third-party scripts. To address the shortcomings of prior work, we propose an approach which: does not require individual sites to provide custom policies, enables full access to all JavaScript features, does not require costly script rewriting or server-side computing, and comprehensively protects explicit exfiltration channels.

2.2 Ad Blockers

An arms race is escalating between online publishers and ad blockers. As ad blocking software becomes better at detecting advertising and analytics scripts, publishers devise new ways to circumvent the blockers, since they rely on advertising revenue to operate [21].

Initiatives like the Coalition for Better Ads [22] aim to find a compromise between advertisers and users of the web. The Coalition for Better Ads is comprised of almost 50 companies, including Google, Facebook, the Washington Post, AppNexus, and Procter & Gamble. The organization has developed a set of criteria that describe the low-quality ad formats that frustrate users and motivated ad blockers in the first place. These formats include pop-up ads, auto-playing video ads with sound, prestitial ads with a countdowns (ads which appear before the main content has loaded), and large sticky ads.

The Google Chrome browser ships with a built-in selective ad blocker which blocks ads deemed unacceptable according to the standards set by the Coalition for Better Ads [5]. This presents a severe conflict of interest — Google operates an ad network, while also implementing a selectively blocking competitor's ads based on standards it helped set. Additionally, the standards for "acceptable" ads include room for interpretation, so enforcing them involves subjective judgment which could be biased.

AdBlock Plus, a popular ad blocker, earns most of its revenue by accepting payment from companies to whitelist their ads, provided they meet the Acceptable Ads standard [6]. The standard are set by a committee of independent members, and ads which do not meet the standard are not whitelisted, but the specific criteria also leave room for interpretation [23].

Selective ad blockers de-escalate the ad blocker arms race, but selective ad blockers based on subjective criteria such as user experience qualities have been problematic. Therefore, we propose a selective ad blocker based on objective security properties of advertising and analytics scripts: scripts that exhibit risk of exfiltrating sensitive data are blocked, and all others are allowed. We show that in the short term, our ad blocker would offer strong privacy protection to users while blocking most of the ads blocked by a universal ad blocker. However, as ad providers move to offer scripts with better security design (e.g., using data-confined sandboxes), the ad blocker would permit these newly-secure ads while maintaining the same privacy protection properties.

Chapter 3

Exfiltration Policy

Our goal is to prevent third-party scripts from exfiltrating sensitive user data from the browser. We introduce a simple data exfiltration protection policy based on taint checking. The policy prevents untrusted scripts that have accessed potentially sensitive data (via *tainting APIs*) from accessing explicit exfiltration channels (via *exfiltration APIs*). The policy specification follows:

> We keep track of whether scripts are trusted or untrusted by adding them to the trusted group or the untrusted group. Each group is initialized to the empty set at the start of a page load.
>
> The untrusted script group is initially marked untainted.
>
> When a script is parsed, we evaluate a trust policy to determine whether it should be trusted. If the script is trusted, it is added to the trusted group; otherwise, it is added to the untrusted group.
>
> On each call to a tainting API, if the call stack contains any script in the untrusted group, the untrusted group is marked as tainted.
>
> Calls to exfiltration APIs are blocked if the call stack contains one or more tainted, untrusted scripts.

In this section, we motivate each part of this policy: the design of script trust policies, the types of APIs that return potentially confidential data, which APIs are exfiltration channels, and how we implement taint-checking.

3.1 Trust Policy

Trust policies classify scripts as trusted or untrusted. We construct trust policies using a table of URL predicates, which are functions that accept a script URL as a parameter and return a boolean, indicating whether the URL is matched. We use three types of URL predicates:

1. *Always-true predicate* — This predicate always returns true, matching all script URLs.
2. *Hostname predicate* — This predicate returns true if the script URL has a given scheme, hostname, and port number (if specified). For instance, the hostname predicate defined by the scheme "https", hostname "example.com" and undefined port matches the scripts https://example.com/ads.js but not https://example.com:8080/ads.js or https://example.org/ads.js. Note that the (scheme, hostname, port) tuple defines the origin of a URL. However, since the origin of a script at runtime depends on how it is loaded, not just on its URL, we refer the tuple as a *hostname* to avoid confusion with the origin the script is running under.
3. *Filter list predicate* — This predicate is defined by a list of wildcard patterns or regular expressions in the format of an AdBlock filter list [24]. The predicate returns true if the remote script URL matches at least one of the globs/regular expressions in the pattern list.

We build trust policies by composing these predicates into tables of rules. Each row consists of a URL predicate and a flag that indicates whether scripts matching the predicate should be trusted. The first row to match a script's source URL determines whether that script is trusted. To build a whitelist policy, we place trusted predicates at the top of the table followed by a default untrusted always-true predicate. Similarly, to build a blacklist policy, we place untrusted predicates at the top of the table followed by a default trusted always-true predicate. With this table structure, more complex polices can be built by interleaving trusted and untrusted predicates.

A script's URL alone is not sufficient to determine whether it should be trusted. We also need to consider how the script is loaded. If an untrusted script embeds a new script, the new script must also be considered untrusted. Even if the embedded script was benign, individually-benign code can be composed to create malicious behavior (similar to attacks based on ROP gadgets [25]). The complete algorithm for evaluating a trust policy on a new script is given in Algorithm 1.

Algorithm 1: Evaluating a trust policy

```
Data: Script parsed event $E$ and untrusted script group $U$
Result: true if the new script is trusted, false otherwise
if initiator of event $E$ is a script then
    for script ID $c$ in initiator call stack do
        if $c \in U$ then
            return false
if newly parsed script is a remote script then
    $url \leftarrow$ URL of newly parsed script
    for ($shouldTrust$, $predicate$) in predicate table do
        if $predicate(url)$ then
            return $shouldTrust$
return true
```

The trusted and untrusted script groups are initialized to empty sets for each new page. Each newly parse script is added to either the untrusted or trusted script sets based on the outcome of Algorithm 1.

We refer to scripts that are embedded in the original HTML response document as *statically-loaded scripts.* We refer to scripts created by other scripts at runtime as *dynamically-loaded scripts.* Examples of dynamically-loaded scripts include ⟨`script`⟩ elements inserted into the DOM by JavaScript, or scripts executed by the `eval` function.

We say a script has a *trusted initiator* if it is either present in the original document (a statically-loaded script) or if it is created dynamically by a fully-trusted call stack (every script on the stack is trusted). According to Algorithm 1, scripts with untrusted initiators are always untrusted. Remote scripts with trusted initiators are only trusted if the predicate table classifies the URL as trusted. Inline scripts with trusted initiators are always trusted. This design decision is based on the assumption that inline scripts included in the original response document are usually small enough to manually audit.

We introduce and explore the implications of two simple anti-exfiltration policies:

1) Hostname Policy. This whitelist policy trusts all inline scripts with trusted initiators. We trust remote scripts with trusted initiators whose hostnames matche the origin of the containing document. All other scripts are untrusted.

Hostname Policy Predicate Table

Rule	Trust	Predicate
1	✓	HostnamePredicate⟨document.origin⟩
2	×	TruePredicate

2) Ad Blocker Policy. This blacklist policy trusts all scripts except for those loaded from URLs that would be blocked by an ad blocker. We use the EasyList filter list [26] to determine which scripts to distrust. EasyList, originally designed for the Adblock Plus ad blocker, is a list of URL patterns designed to comprehensively match advertising and analytics scripts on the web.

Ad Blocker Policy Predicate Table

Rule	Trust	Predicate
1	×	FilterListPredicate⟨EasyList⟩
2	✓	TruePredicate

The widespread use of ad blockers shows that most websites can operate effectively without advertising and analytics scripts, so blocking functionality in advertising and analytics scripts that violates the exfiltration policy will disrupt user experience at most as much as a standard ad blocker. Therefore, with some cosmetic document cleanup in the event of policy violations (e.g., removing partially initialized ads), this policy can provide strong security benefits to the user with a clear upper bound on its impact on user experience. Additionally, this policy encodes a good heuristic for the risk level of a third-party script: advertising and analytics scripts have historically been especially prone to malicious activity.

3.2 Tainting APIs

We call APIs which can read sensitive state *tainting APIs*. Tainting APIs include all APIs protected by the same-origin policy. This includes (but is not limited to) APIs that interact with the DOM elements, Local Storage, Session Storage, IndexedDB, Navigator, and History.

Notably, we consider all APIs that interact with persistent state to be tainting APIs. The policy state (which includes the sets of trusted/untrusted scripts) is reset every page load, so we do not know the provenance of persisted data at the start of a page load. If data store APIs were not tainting APIs, an adversary could persist confidential data without violating the policy, then read it from the data store API and exfiltrate it without violation on the next page load. Other APIs, like window.name, persist across page loads, and can therefore be used as ad-hoc data stores, so we must consider these tainting APIs as well [15].

For our feasibility study, we developed a static list of 465 tainting APIs by enumerating the methods in the JavaScript prototype of classes which operate on sensitive data. These classes are: Attr, CacheStorage, CharacterData, Document, Element, History, HTMLElement, IDBFactory, Navigator, and Storage. We also hand-select sensitive attributes of window, including window.name. The full list of tainting APIs is available online [27]. Production-ready implementations of this policy should use the browser's built-in security model used for the same-origin policy to determine whether an API is sensitive, rather than using a static list.

Many websites store custom state in the JavaScript global scope (for example in an object window.applicationState). If adversarial scripts could read data in these objects without being tainted, the scripts could trivially exfiltrate it without violating the policy. Therefore, a production-ready implementation of our policy would need to taint scripts that access these non-default global objects. For the purposes of our feasibility study, we do not consider these targeted attacks on global application state. We don't believe this affects our experimental results because we assume the third-party scripts we studied were benign.

3.3 Exfiltration

Akhawe et al. [20] identified three types of data exfiltration channels:

1) Network Channels. The simplest exfiltration technique is sending the data over the network to store it somewhere an adversary can read it. Many JavaScript APIs enable network requests. For example, the XHR (XMLHttpRequest) API allows the client to send generic HTTP requests, and read their responses (subject to the same-origin policy). Other APIs, such as the DOM, can send network requests to load resources such as images. Network requests can also be made without JavaScript code (e.g., via references to network resources in CSS stylesheets) leading to scriptless exfiltration attacks [28].

Data exfiltration via network channels occurs when confidential data is embedded in the outgoing requests. Despite the simplicity of the attack, it is difficult to determine whether a given network request sent by an untrusted script is actually exfiltrating data. Inspecting the request body for presence of confidential data is not possible, since the adversary could obfuscate or encrypt the data, or send the data in many requests, one bit at a time. Even if the request body is static, binary data can be exfiltrated via timing channels. For example, an adversary can exfiltrate one bit per clock tick by sending a request on the tick to transmit a 1-bit and sending no request on the tick to transmit a 0-bit.

Additionally, Chen et al. [15] caution against browser-enforced exfiltration protection policies based on the destination of network requests. Adversaries can often exfiltrate data by sending it to a local endpoint on the victim site which makes it accessible to the adversary on the site at a later time. For example, an advertising script on a blogging website could exfiltrate user data by posting it in a public comment on the same site. These first-party exfiltration channels are widespread: in 2012, Chen et al. [15] found that at least one self-exfiltration channel existed in each of the Alexa top 100 websites.

Considering the complexity of analyzing request bodies, we conservatively assume that any network request initiated by a script is an exfiltration attempt.

2) Client-Side Cross-Origin Channels. Browser features like `window.location`, `window.open`, and `postMessage` allow for a constrained form of cross-origin communication. For example, the `postMessage` API could be used to send data to a different web page, out of the scope of the active policy. Therefore, we consider calling these APIs possible exfiltration attempts.

3) Storage Channels. Instead of immediately exfiltrating confidential data over the network, a tainted script may first store it in persistent storage. A script running later on the same origin could then retrieve this data and exfiltrate it using a network channel. For the same reasons we don't attempt to analyze network request bodies, we do not analyze the data stored in persistent stores to determine whether it's confidential. Instead, we conservatively consider all APIs that access persistent storage to be tainting APIs.

We call APIs which are capable of exfiltrating data *exfiltration APIs.* To cover all possible exfiltration methods, we consider all APIs that can generate network requests, or that allow for cross-origin communication, to be exfiltration APIs.

Covert Channels. Our policy blocks many covert exfiltration channels, including timing based network exfiltration, self-exfiltration, and some basic scriptless attacks, but we recognize that some more complicated covert channels still exist. For example, an adversary may be able to construct a storage exfiltration channel using the CPU cache, storing bits by priming the cache, and reading data without using tainting APIs by timing cache probes. We consider such covert channel attacks out of scope and focus on explicit exfiltration.

3.4 Taint Checking

We have defined how scripts become tainted (by calling tainting APIs) and which functionality should be blocked for tainted scripts (calling exfiltration APIs). To complete our policy definition, we define how we propagate taint.

An ideal taint-tracking model would be based on a script's information flow graph. The tainted state would follow edges in the data-flow and control-flow graph, so only code paths that actually have access to sensitive data would be tainted. Propagating taint only along data flow dependencies can result in under-tainting (i.e., sensitive data might reach a section of untainted code). Propagating taint along control-flow dependencies (by allowing the program counter to become tainted) can lead to an explosion of tainting, and potentially over-tainting (i.e., tainting code that does not have access to the sensitive data).

Regardless of how taint is propagated, tainting across information flow edges requires static or dynamic analysis of the JavaScript engine internals. If our policy were enforced in a browser, we could not afford the significant run-time overhead from dynamic analysis or latency from static analysis, so we opt for simple conservative tainting policy: once a single untrusted script is tainted, we consider all untrusted scripts tainted.

This tainting policy is conservative and coarse, potentially resulting in false positives where a script becomes tainted even though it does not actually have access to sensitive data. However, as we show in Chapter 5 and Chapter 6, this simple policy can be effective in practice.

```
<html>
<head>
<!-- Script 1 -->
<script src="/script.js"></script>

<!-- Script 2 -->
<script src="https://example-cdn.com/jquery.js"></script>

<!-- Script 3 -->
<script>
    window.onload = function() {
        // Script 4
        let s1 = document.createElement("script");
        s1.src = "/lib.js";
        document.head.appendChild(s1);

        // Script 5
        let s2 = document.createElement("script");
        s2.src = "https://example.com/ads.min.js";
        document.head.appendChild(s2);
    };
</script>
</head>
<!-- Script 6 -->
<body onclick="console.log(1)">
</body>
</html>
```

Figure 3.1: Trust Policy Case Study Code

3.5 Illustrative Example

The example web page in Figure 3.1 shows four statically loaded scripts (Script 1, 2, 3, and 6) and two dynamically loaded scripts (Script 4 and 5).

According to the hostname trust policy, Script 1 is trusted since it is a static remote script with the same hostname as the main document. Script 6 is also trusted since it is a static inline script. Script 2 is untrusted since it is served from a different hostname than the main document. Script 3 is trusted since it is an static inline script. Script 4 is trusted since it is a remote script created dynamically in a trusted call stack (Script 3 is trusted) and the hostname of the script matches the main document's. Finally, Script 5 is dynamically-inserted and will be untrusted since its hostname does not match the main document's hostname.

According to the ad blocker trust policy, Scripts 1 and 2 are trusted since they are static

Script	Hostname	Filter List
1	✓	✓
2		✓
3	✓	✓
4	✓	✓
5		
6	✓	✓

✓: Script is trusted under this policy

Figure 3.2: Trust Policy Case Study

remote scripts with URLs that do not match EasyList patterns. Script 3 and 6 are both trusted since they are static inline scripts. Script 4 is trusted since it is a remote script created dynamically in a trusted call stack (Script 3 is trusted) and the script URL does not match an EasyList pattern. Script 5 is untrusted since the URL path `/ads.min.js` matches an EasyList pattern. Any scripts that Script 5 creates would also be untrusted, since they would be created from an untrusted call stack.

These trust policy results are summarized in Figure 3.2.

Chapter 4

Policy Implementation

To evaluate how exfiltration policies behave on real websites, we build a proof-of-concept implementation. We base the prototype on the Chromium browser, the open-source counterpart to Google Chrome. Though most prior browser security research uses the Mozilla Firefox browser, we choose Google Chrome since it is the most popular browser worldwide (over 60% market share in February 2019) [29].

A production implementation of our anti-exfiltration policy would be embedded directly in the browser, facilitating a level of performance and security that is not possible without controlling the browser implementation. However, modern browsers are large and highly complex codebases (Chromium has millions of lines of of C++), so the browser development workflow is not ideal for research. Slow builds and limited documentation make iterative development cumbersome. Since our focus is on designing and testing a security policy, not on implementing a production-ready feature, we developed a system that allows us to rapidly prototype browser security policies with minimal modifications to Chromium.

Our implementation approach is based on the insight that it is possible to manually evaluate many browser security policies by stepping through JavaScript code using the browser's built-in debugger. The Chrome Remote Debugger, included in Chromium, allows a separate process to programmatically control and instrument many parts of the browser, including the JavaScript runtime and debugger, via the Chrome DevTools Protocol [30].

Existing JavaScript instrumentation tools did not fit our needs, since they are primarily intended for runtime performance analysis (e.g., Google's Web Tracing Framework [31]). The existing JavaScript instrumentation tools we evaluated did not operate on page-level networking activity or allow us to easily control the execution of JavaScript in real time, but the Chrome Remote Debugger does both.

We implemented our policy by writing a standalone tool that controls the Chrome Remote Debugger. We call the tool the *Remote Instrumentation Client*, or the *RIC*. The RIC is implemented in about 2,500 lines of code, mostly written in Go. The RIC approach enabled us to make modifications to Chromium only where absolutely necessary (only about 30 lines of C++), saving us massive amounts of development time. Code for the instrumentation tool and the customized Chromium patches are available online [32].

4.1 Remote Instrumentation Client Design

Figure 4.1 shows a high level view of the RIC architecture. Each component is explained in detail in the following sections.

The RIC command line tool manages the full lifecycle of the Chromium browser, including launching, attaching to the Chrome Remote Debugger, loading and instrumenting a web page, and cleanly exiting. Optionally, the RIC can run Chromium in headless mode, without a graphical user interface (implemented using `xvfb-run` [33] due to bugs in Chromium's built-in headless mode).

The RIC opens an HTTP connection to the Chrome Remote Debugger server and uses the `godet` client library [34] to interact with the Chrome DevTools Protocol. The RIC supports instrumenting multiple simultaneous tabs and windows, making it suitable for running both automated and interactive user studies.

The Chrome DevTools Protocol divides functionality into *domains* (e.g., Network, Debugger, etc.) and supports two modes of interaction with each domain: *commands* and *events*. Both commands and events are asynchronous, but we found no evidence of commands or events being delivered to the RIC out of order in a way that affects policy violations.

We built a web interface for the RIC that allows us to visualize policy state and analyze event logs. The web interface also includes a number of interactive test cases for our taint analysis and exfiltration analysis. We can check that the policy is implemented correctly by visiting each test case page and verifying the policy state is correct. In Figure 4.2, we see a screenshot of part of the web interface showing two passing test cases for the strict hostname policy. The first correctly yields no policy violation, and the second correctly yields a policy violation because an untrusted script accessed cookies before sending an XHR to a different hostname.

The RIC can be configured via command-line flags to take a screenshot and upload results to Amazon S3 at the end of an instrumentation run.

Since our goal is to understand how real-world web traffic aligns with our policies, we do not actually block violations of the policy in our experiments. Instead, we simply log the violating event and continue to execute the instrumented script. Blocking the first violating network request could change the subsequent script behavior, but for the purposes of analyzing policy violations, the behavior up until the first violation is identical whether we enforce the policy by blocking or not.

4.2 Instrumenting Tainting APIs

The RIC must instrument use of tainting APIs in order to properly taint the untrusted scripts. One possible approach would be to use the Debugger.stepInto command and the Debugger.paused event to programmatically step through every line of JavaScript and detect use of an API based on the name of the function where the debugger is paused. However, an API may have aliases so this method is not robust.

To pause when an API is called, we need a way to set a breakpoint *inside* of that API (i.e., on its first line). However, JavaScript breakpoints cannot be set in the native C++ implementations of built-in browser APIs. Furthermore, we need to also support APIs which are attribute-based rather than functional (e.g., the window.name attribute), so there may not always be a function in which to set a breakpoint.

Our solution is to inject *API shims*, which allow us to intercept API calls. Before script execution, we overwrite APIs with wrapper functions, or shims. The shims are implemented using dynamic accessor properties (getter and setter functions) which run instrumentation code and call the underlying APIs.

To shim APIs implemented as methods or attributes of classes, we overwrite the corresponding attribute of the class prototype. For example, to shim the geolocation property of the Navigator API, we use Object.defineProperty to override Navigator.prototype.geolocation with a dynamic accessor property. The getter of this accessor property pauses the debugger, which sends a Debugger.paused event and passes execution control to the RIC. After recording the API access and potentially tainting the untrusted scripts, the RIC sends a Debugger.resume command to the debugger to resume execution. The shim resumes and calls the original API implementation.

To shim APIs that are not part of a class prototype, such as the window.name attribute, we simply set our dynamic accessor property shim directly on the object instance (in this case, directly on window).

We base our implementation of API shims on the break-on-access debugging tool [35]. We make no effort to prevent adversaries directly targeting RIC from disabling or bypassing shims. Implementing instrumentation in JavaScript rather than in native browser code saves time for prototyping, but is difficult to make secure since the policy code shares the runtime with malicious code. Browser vendors are best-suited to build secure policy mechanisms in native code once a prototype has been validated. Our threat model of the API shims reflects the intended use of the RIC as an instrumentation and prototyping tool without targeted attackers, rather than a hardened policy enforcer.

Pausing and resuming the debugger when tainting APIs are accessed can substantially degrade runtime performance. As a performance optimization, we observe that accessing a tainting API from a trusted call stack will never taint untrusted scripts, and we forgo the pause/resume round trip in this case. To implement this optimization, the RIC writes the current list of trusted script IDs to an object in the JavaScript runtime each time the debugger is paused (using Debugger.setVariable). The shim determines whether the call stack is trusted by comparing the list of trusted script IDs to the output of console.stackTrace(). console.stackTrace is a JavaScript builtin function we implemented in the V8 engine which returns an array of script IDs corresponding to each frame of the call stack.

This optimization significantly improves the performance of the RIC, especially on complex trusted scripts. We also took care to tune performance of our injected JavaScript shims. For example, we initially used a JavaScript forEach loop to iterate through the stack trace, but discovered through profiling that a regular for loop was significantly faster.

4.3 Instrumenting Exfiltration APIs

Recall that exfiltration APIs include both networking and non-networking APIs. The Chrome DevTools Protocol facilitates intercepting all network activity, and includes stack traces of the initiator script (if applicable), so rather than shimming networking APIs, we instrument network exfiltration attempts using the built-in Network.requestWillBeSent event. We acknowledge that the Chrome DevTools Protocol does not correctly attribute network requests sent from CSS generated by JavaScript to the script, so we fail to properly detect scriptless exfiltration. A real world implementation would remedy this issue with correct attribution.

For non-networking exfiltration APIs, such as window.open, we intercept and log them by creating exfiltration API shims.

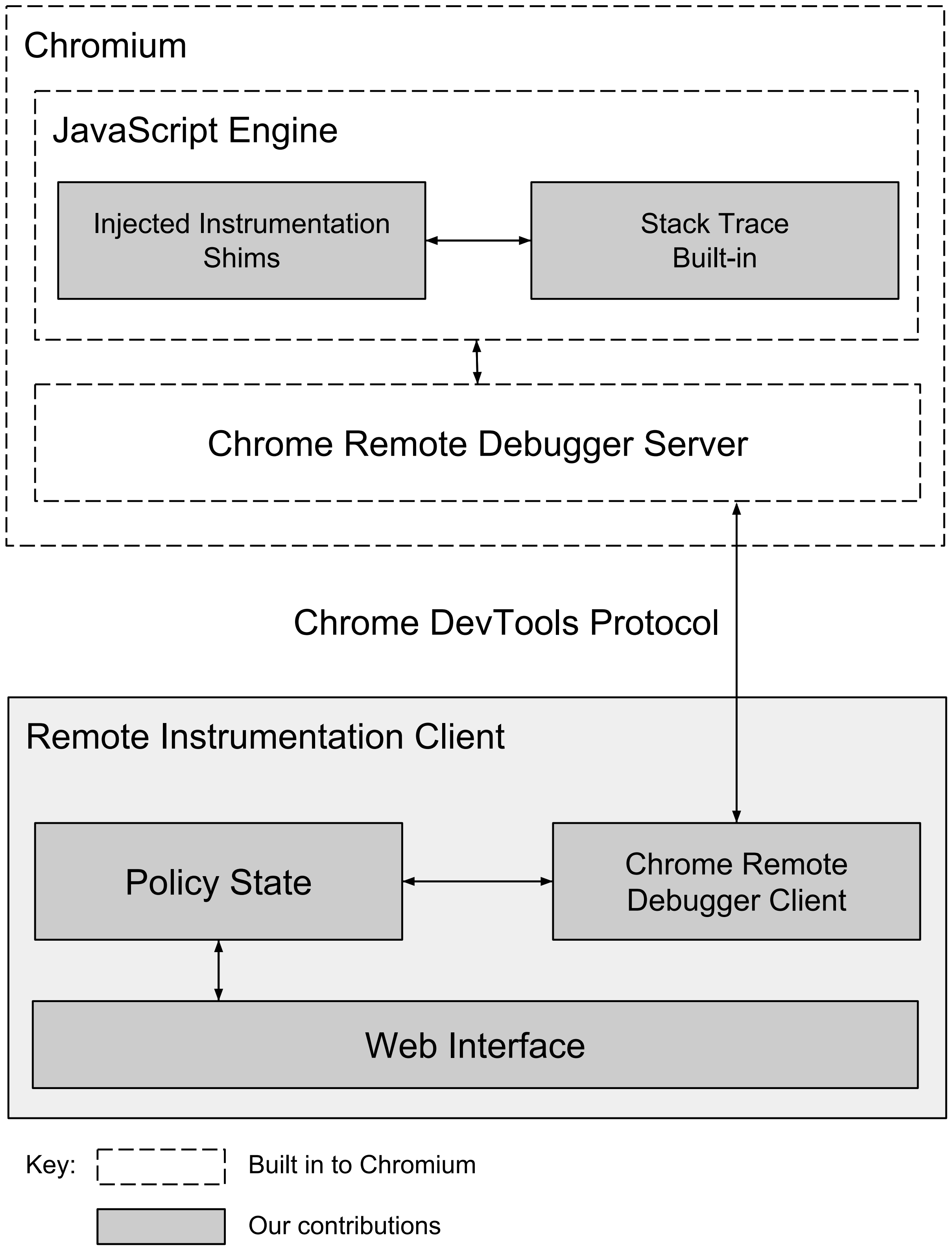
Exfiltration API calls are not blocked, even if the case of a policy violation, since our goal is to experimentally test the impact of our policies rather than to actually implement a policy. Policy violations are analyzed after the fact from log files.

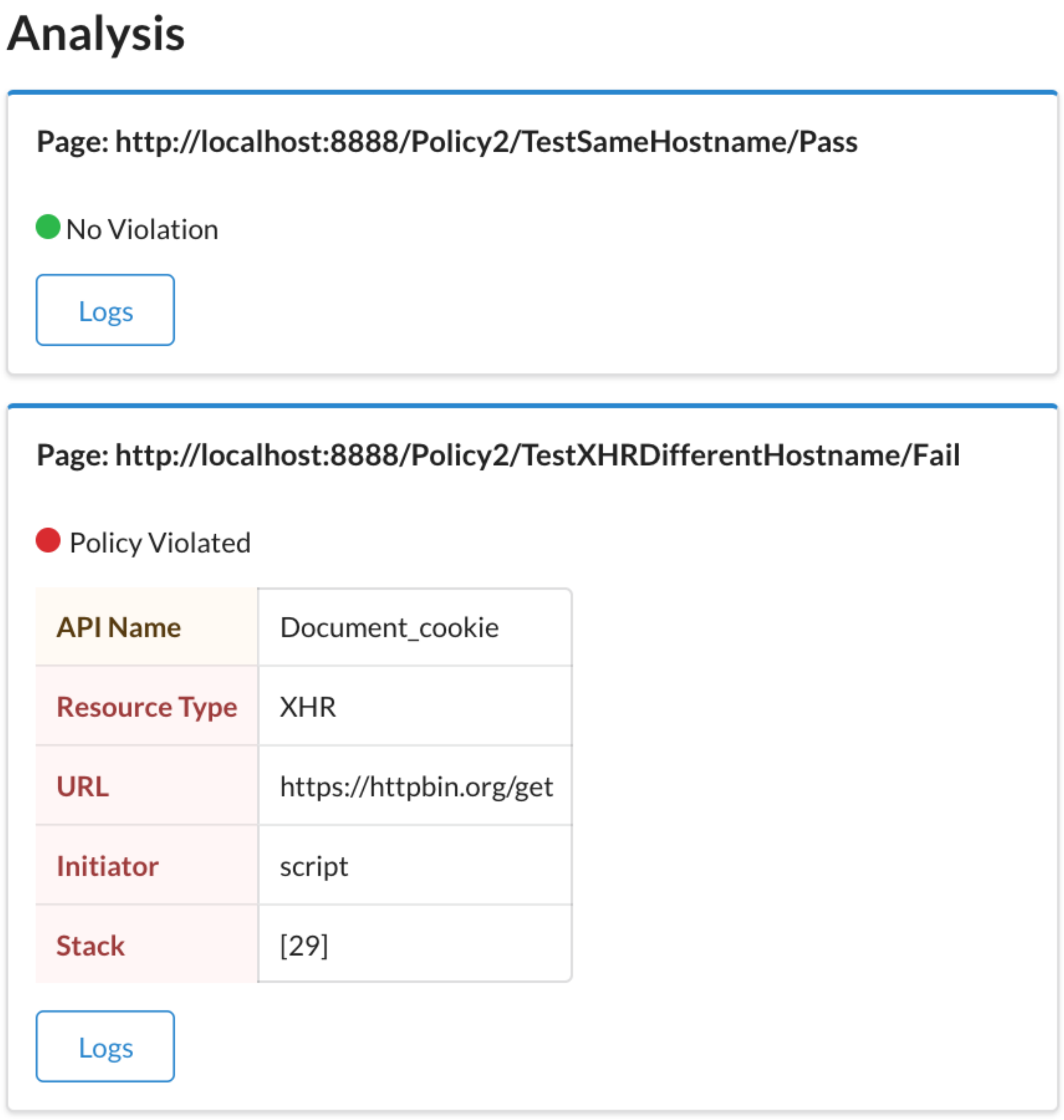
4.4 Trust Policy Evaluation

The RIC makes creating new trust policies and URL predicates simple, facilitating rapid experimentation. The filter list predicate in the ad blocker trust policy was implemented using Mézard's Go library to parse and match against EasyList patterns [36]. The trust policy version in use (i.e., hostname or ad blocker policy) is selected using a command line flag.

The RIC follows Algorithm 1 to evaluate trust policies. The RIC handles Debugger. scriptParsed events and evaluates the trust policy to determine whether the script should be trusted. If a script was created by an existing script (e.g., by inserting a ⟨script⟩ element, or running eval), the initiator call stack is embedded in the Debugger.scriptParsed event.

Figure 4.1: RIC Architecture





Analysis

Page: http://localhost:8888/Policy2/TestSameHostname/Pass

● No Violation

Logs

Page: http://localhost:8888/Policy2/TestXHRDifferentHostname/Fail

● Policy Violated

API Name	Document_cookie
Resource Type	XHR
URL	https://httpbin.org/get
Initiator	script
Stack	[29]

Logs

Figure 4.2: Part of the web interface for visualizing policy state

Chapter 5

Evaluation

We used our RIC tool in a large-scale web scan to evaluate the extent to which popular websites comply with our anti-exfiltration policies.

5.1 Methodology

Web Crawl. We built a representative sample of the most popular pages on the internet by crawling sites from the Tranco list [37]. The Tranco list is a ranking of the most popular sites on the internet which addresses issues with traditional rankings like Alexa: the Tranco list is designed to be resistant to adversarial manipulation and to be stable over time. We used the Tranco list[1] created on March 4, 2019. For the top 500 sites in this list, we used a Chromium browser programmatically operated by Selenium [38] to crawl each domain. The crawler, written in Go, discovered new pages by following links (anchor tags). Only links pointing to URLs within same domain were followed. We crawled each domain three levels deep, or until 100 URLs were found for each domain (whichever condition was met first). Finally, we randomly sampled five URLs from each domain's list of discovered URLs. This yielded a total of 2500 web pages.

[1]Available at https://tranco-list.eu/list/2549

The crawl was run from an IP address in Charlottesville, VA, which had a small impact on the URLs discovered. For instance, craigslist.org redirected to charlottesville.craigslist.org based on the location estimated from the crawler's IP.

Instrumentation. For each anti-exfiltration policy, we ran the RIC on each of the 2500 web pages found in the crawl, logging all policy state, but not blocking policy violations.

We ran the instrumentation jobs on sixteen t2.2xlarge Amazon Web Services EC2 instances running Ubuntu 18.04 in the us-east-1 region. Instance provisioning and job scheduling was implemented using Ansible [39] and a simple xargs script.

In a real-world implementation, the policy mechanism would be in effect until the user closes the browser window. However, JavaScript on web pages can execute indefinitely so for the purposes of this study, we had to choose when to end instrumentation. We collected instrumentation data until the browser fired a load lifecycle event (via the Chrome DevTools Protocol event Page.lifecycleEvent). Thus, we evaluated the policy until the browser considers the page fully loaded, but we recognize that this may result in missing policy violations that occur after the load event. We do not believe this has affected the interpretability of our results.

5.2 Results

We found that most web pages with ads violate both policies. However, understanding the nature of the violations informs us of how websites need to change to comply with the ad blocking policy and benefit from its strong security and privacy properties.

We analyze the experimental results for both policies:

Hostname Policy. For the hostname policy, 1772 pages were instrumented without timeout or other unexpected error. Of these, 1565 pages contained untrusted scripts. On the pages with untrusted scripts, 1509 of the untrusted script groups were tainted by accessing tainting APIs. A total of 1431 pages had untrusted scripts that called exfiltration APIs after being tainted, triggering a policy violation. Altogether, about 81% of the instrumented pages had policy violations. Of the pages with third-party scripts, 91% had policy violations.

Figure 5.1 shows the types of policy violations that occurred with the hostname policy. No policy violations occurred due to non-networking exfiltration APIs (such as window.name). DOM operations (particularly directly on the Document object) were the most common tainting APIs. We found that scripts were the most common exfiltrating network resource type because third-party scripts often load their own dependencies by inserting script tags.

Ad Blocker Policy. For the ad blocker policy, 1882 pages were instrumented without timeout or other unexpected error. Of these, 1193 pages contained untrusted scripts. On the pages with untrusted scripts, 1122 of the untrusted script groups were tainted by accessing tainting APIs. A total of 1038 pages had untrusted scripts that called exfiltration APIs after being tainted, triggering a policy violation. Altogether, about 55% of the instrumented pages had policy violations. Of the pages with ad/analytics scripts, about 87% had policy violations.

The top 10 most common APIs that tainted untrusted scripts were all APIs in the Document object, including Document.cookie, Document.referrer, Document.URL, and other DOM APIs.

Figure 5.2 shows the types of policy violations that occurred with the ad blocker policy. No policy violations occurred due to non-networking exfiltration APIs. We found network requests for images were the most common exfiltrating network resource type in policy violations (present on 461 pages) because analytics scripts collect data using requests to image resources. The Google Analytics collection endpoints alone accounted for 263 of these image-based violations, since the Google Analytics script uploads data by embedding it in the query string of an image URL. Network requests for script resources resulted in policy violations on 353 pages, since many advertising and analytics scripts are embedded as small scripts which dynamically load larger scripts. Scripts that violated the policy were all tainted by DOM operations.

Tainting API Class	Exfiltrating Network Resource Type								
	Document	Font	Image	Media	Script	Stylesheet	XHR	Other	**Total**
Attr	1	0	3	0	7	0	4	0	15
Document	38	107	314	3	613	33	233	38	1379
Element	0	0	0	0	5	0	3	0	8
HTMLElement	0	2	0	0	3	0	0	0	5
History	0	0	0	0	11	0	1	0	12
Navigator	0	1	6	0	5	0	0	0	12
Total	39	110	323	3	644	33	241	38	1431

Figure 5.1: Types of Policy Violations for Hostname Policy

Tainting API Class	Exfiltrating Network Resource Type								
	Document	Font	Image	Media	Script	Stylesheet	XHR	Other	**Total**
Attr	0	1	3	0	2	0	1	1	8
Document	33	16	432	1	327	6	142	21	978
History	0	0	0	0	17	0	0	0	17
Navigator	0	0	26	0	7	0	0	2	35
Total	33	17	461	1	353	6	143	24	1038

Figure 5.2: Types of Policy Violations for Ad Blocker Policy

Analysis. We conclude that inferring trust of scripts based on their hostname is not a feasible policy in most cases. Most sites have policy violations and trusting only scripts with hostnames matching the site origin is too restrictive to be compatible with most popular websites. We found that only about 12% of the pages studied (207 pages) hosted all script resources on their trusted first-party domain (e.g., wikipedia.org hosts all script resources on the same domain).

The ad blocker policy showed slightly lower rates of violation than the strict hostname policy, meaning compatibility with most popular websites is low unless changes are made. However, we found that most policy violations are caused by just a few scripts. Of the pages we studied, about 17% included untrusted scripts exclusively from the three most common advertising and analytics networks (Google, Scorecard Research, and Quantserve). If these three networks alone restructured their advertising scripts in a way that was compliant with anti-exfiltration policies, the percentage of policy-compliant pages would increase from about 45% to about 62%.

Chapter 6

Adapting to Anti-Exfiltration

In this section, we describe techniques advertising and analytics providers could use to comply with the anti-exfiltration policy.

Honest advertising and analytics scripts legitimately need some types of sensitive data from tainting APIs (for example, a tracking ID from a storage API, tracking tokens from the URL query string, or the referrer URL in document.referrer). Our recommended approach for accommodating these needs without policy violations is based on data-confined sandboxes [20]. We construct a very small, easily-auditable trusted computing base which reads this required data from tainting APIs and transmits it to the untrusted script running inside an iframe sandbox.

Analytics Example. A simple example of a secure analytics script implementation is shown in Figure 6.1. A small easily-auditable, trusted inline script reads and sends data to an analytics.net script in an iframe via postMessage. The page is contained in an iframe with the sandbox features allow-scripts and allow-same-origin, which allows scripts in the iframe to execute in the analytics.net origin. Since the analytics script does not directly access the tainting APIs, it is not tainted and therefore does not violate the anti-exfiltration policy.

Some sites, like www.dropbox.com, already practice a similar technique to securely use untrusted analytics scripts. Dropbox runs Google Analytics on a separate subdomain,

```
<html>
<head>
<script>
    window.onload = function() {
        let a = document.getElementById("a");
        a.contentWindow.postMessage({
            clientID: "AJFKWOJSNPKDLINH",
            referrer: document.referrer,
            url: document.URL,
            ...
        }, "https://analytics.net");
    };
</script>
</head>
<body>
    <iframe
        src="https://analytics.net/iframe.html"
        id="a"
        sandbox="allow-scripts allow-same-origin"
        style="display: none;"
    ></iframe>
</body>
</html>
```

Figure 6.1: Simple Secure Analytics Implementation

marketing.dropbox.com, to prevent the script from accessing confidential user data on the www.dropbox.com origin. To send a referrer to Google Analytics, www.dropbox.com encodes the referrer in the query string to the marketing subdomain (i.e., https://marketing.dropbox.com?referrer=⟨referrer⟩). The main site safely loads this URL in a iframe sandbox. A script on marketing.dropbox.com shims the referrer from the query string into document.referrer for Google Analytics script to process. This ad hoc approach to securely sending a referrer to an analytics provider could be avoided if the analytics scripts were designed using the techniques we propose.

Display Ad Example. A simple example of a secure image-based display ad implementation is shown in Figure 6.2. An untrusted script inserting an image into the DOM will always result in a policy violation, since manipulating the DOM to insert the ⟨img⟩ element taints the script, and the subsequent network request to fetch the image violates the exfiltration policy. Our solution is to place ads in sandboxed iframes which are configured using a small,

```
<html>
<head>
<script>
    window.onload = function() {
        let a = document.getElementById("a");
        a.contentWindow.postMessage({
            clientID: "AJFKWOJSNPKDLINH",
            url: document.URL,
        }, "https://ads.net");
    };
    window.addEventListener("message", function(event) {
        if (event.origin != "https://ads.net") return;
        let a = document.getElementById("display-ad");
        a.src = event.data.url;
    }, false);
</script>
</head>
<body>
    ...
    <iframe
        src="https://ads.net/iframe.html"
        id="display-ad"
        sandbox="allow-scripts allow-same-origin"
    ></iframe>
    ...

    <iframe
        src="https://ads.net/ads.js"
        id="a"
        sandbox="allow-scripts allow-same-origin"
        style="display: none;"
    ></iframe>
</body>
</html>
```

Figure 6.2: Simple Secure Display Ad Implementation

easily-auditable trusted script. A client ID and a page identifier are sent to the ad network via a message to an sandboxed web page hosted by the ad provider. The ad provider sends a message back to the main web page with a the URL of an ad to display. The containing page sets the source of the iframe to that URL, which securely loads the ad.

To make a script compliant with the anti-exfiltration policy, the script provider should port it to the data-confined sandbox design. This is accomplished by implementing a restricted mediation API over the postMessage channel to interface the embedded sandbox with the parent page. With this design, all sensitive data not explicitly offered through the mediation API is isolated from the third-party script by the same-origin policy. Akhawe et al. [20] showed that a variety of real-world codebases can be ported to use data-confined sandboxes with minimal code changes, so keeping the TCB small should be feasible for most advertising and analytics scripts. Ideally, the script provider should author the small mediation TCB with public oversight (e.g., by making it open source) to facilitate accountability and auditability. Publicly audited mediation API TCBs are whitelisted by hash in the browser's anti-exfiltration policy, so fetching sensitive data that has been explicitly allowed does not result in policy violations.

With this design, the third-party script author sets the sensitive data access policy it must adhere to by publishing a mediation API. Since site owners don't have to provide their own policies, the adoption issues CSP has faced [13] are not applicable.

Chapter 7

Limitations and Future Work

Our methodology does not evaluate the policy on website functionality which requires user interaction, such as filling out a form or logging in. A user study would help evaluate policy compatibility with interactive features, but the strong security properties of the policies hold regardless of whether we consider interactive functionality.

Additionally, our feasibility study methodology does not detect scriptless attacks, since the Chrome Remote Debugger does not properly attribute the script initiator of network events that occur within CSS. However, the approach we demonstrate in Chapter 6 effectively prevents scriptless attacks. Future work should include studying how scriptless exfiltration channels affect policy compliance.

Our proposed secure advertisment and analytics script design does not easily support rich DOM operations without incorporating a large trusted compute base. Client-side techniques for safely evaluating the results of DOM operations, like DOMPurify's approach using HTML5 template tags [40], can be used to implement these rich DOM operations, but using a library for this requires trusting another third-party library. Ideally, we would use built-in browser features to limit the TCB. We recommend that untrusted third-party scripts attempt to accomplish all DOM interactions inside visible iframe sandboxes so DOM artifacts can be directly and safely included.

Finally, other future directions of study might include exploring low-overhead fine-grained taint tracking techniques to make the exfiltration policies more precise and reduce over-tainting.

Chapter 8

Conclusion

Advertisements are critical to the business models of many websites today. However, third-party advertising and analytics scripts present a serious threat to user privacy when they are directly included in web pages that contain sensitive user data. Ad blockers control the risk of malicious third-party advertising and analytics scripts by preventing them from running at all, but an arms race is escalating between publishers and ad blockers. In this study, we focus on limiting the risk of malicious exfiltration by third-party advertising and analytics scripts while de-escalating the ad blocker arms race.

We propose two policies that would help prevent malicious exfiltration in third-party scripts: a hostname policy and an ad blocker policy. We built a novel, general-purpose JavaScript instrumentation tool to help us test these policies in the wild. In a large scale study of the 500 most popular websites, we find the hostname policy to be too restrictive to maintain the utility of most sites. However, the ad blocker policy shows promise since only a few top ad providers need to move to a policy-compliant ad format for a majority of the web to benefit from improved security. We showed how these ad providers can easily make their scripts policy-compliant by using data-confined sandboxes along with trusted sandbox mediation scripts which are small enough to be publicly audited and whitelisted by the policy.

In the same way that traditional ad blockers have led to selective ad blockers which incentivize higher quality ad formats, a security-oriented selective ad blocker based on our anti-exfiltration policy gives advertisers financial incentive to comply with anti-exfiltration rules. As advertisers move toward more secure practices, the policy will progressively allow these secure ads. Users benefit from the ad blocker's privacy protection and publishers of policy-compliant ads benefit from restored ad revenue.

Bibliography

[1] Alex Hern. Major sites including New York Times and BBC hit by ‘ransomware’ malvertising. https://www.theguardian.com/technology/2016/mar/16/major-sites-new-york-times-bbc-ransomware-malvertising, 2016.

[2] Steven Englehardt Gunes Acar and Arvind Narayanan. No boundaries for user identities: Web trackers exploit browser login managers. https://freedom-to-tinker.com/2017/12/27/no-boundaries-for-user-identities-web-trackers-exploit-browser-login-managers/, 2017.

[3] eMarketer. Ad blocking user penetration rate in the united states from 2014 to 2020. *Statista - The Statistics Portal*, 2018.

[4] OnAudience. Comparison of display advertising market value and ad blocking losses worldwide in 2016 and 2017 (in billion u.s. dollars). *Statista - The Statistics Portal*, 2017.

[5] Google. Building a better world wide web. https://blog.chromium.org/2019/01/building-better-world-wide-web.html, 2019.

[6] eyeo GmbH. About adblock plus. https://adblockplus.org/en/about, 2019.

[7] Douglas Crockford. AdSafe. http://www.adsafe.org/, 2019.

[8] Mike Ter Louw, Karthik Thotta Ganesh, and VN Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *USENIX Security Symposium (SSYM)*, 2010.

[9] Mozilla. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, 2019.

[10] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747. ACM, 2012.

[11] Mozilla. Content Security Policy (CSP). https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP, 2019.

[12] Kailas Patil and Braun Frederik. A measurement study of the content security policy on real-world applications. *International Journal of Network Security*, 2016.

[13] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1376–1387. ACM, 2016.

[14] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Data exfiltration in the face of csp. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 853–864. ACM, 2016.

[15] Eric Y Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. In W2SP, 2012.

[16] Lon ingram and Michael Walfish. Treehouse: Javascript sandboxes to help web developers help themselves. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 153–164, Boston, MA, 2012. USENIX.

[17] Mozilla. ¡iframe¿: The Inline Frame element. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe, 2019.

[18] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. JavaScript in JavaScript (js. js): Sandboxing third-party scripts. In *Presented as part of the 3rd {USENIX} Conference on Web Application Development (WebApps 12)*, pages 95–100, 2012.

[19] Google. Caja. https://developers.google.com/caja/, 2017.

[20] Devdatta Akhawe, Frank Li, Warren He, Prateek Saxena, and Dawn Song. Data-Confined HTML5 Applications. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security – ESORICS 2013*, pages 736–754, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[21] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, NDSS 2018, pages 18–21, 01 2018.

[22] Coalition for Better Ads. Coalition for better ads. https://www.betterads.org/, 2018.

[23] eyeo GmbH. Acceptable ads criteria. https://acceptableads.com/en/about/criteria, 2019.

[24] eyeo GmbH. Writing adblock plus filters. https://adblockplus.org/filters, 2018.

[25] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 27–38, New York, NY, USA, 2008. ACM.

[26] EasyList. Easylist - overview. https://easylist.to/, 2019.

[27] Ethan Lowman. Tainting APIs. https://github.com/eal5ub/web-exfiltration/blob/master/js/tainting-apis.txt, 2019.

[28] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 760–771. ACM, 2012.

[29] StatCounter. Browser market share worldwide. http://gs.statcounter.com/browser-market-share, 2019.

[30] ChromeDevTools. Chrome devtools protocol viewer. https://chromedevtools.github.io/devtools-protocol/, 2019.

[31] Google. tracing-framework. https://google.github.io/tracing-framework/, 2019.

[32] Ethan Lowman. eal5ub/web-exfiltration. https://github.com/eal5ub/web-exfiltration, 2019.

[33] Canonical Ltd. Ubuntu Manpage: xvfb-run. https://manpages.ubuntu.com/manpages/trusty/man1/xvfb-run.1.html, 2019.

[34] Raffaele Sena. raff/godet. https://github.com/raff/godet, 2018.

[35] Paul Irish. paulirish/break-on-access. https://github.com/paulirish/break-on-access, 2018.

[36] Patrick Mézard. pmezard/adblock. https://github.com/pmezard/adblock, 2019.

[37] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, February 2019.

[38] seleniumHQ. Selenium - Web Browser Automation. https://www.seleniumhq.org/, 2019.

[39] Red Hat. Ansible is Simple IT Automation. https://www.ansible.com/, 2019.

[40] Cure53. cure53/DOMPurify. https://github.com/cure53/DOMPurify, 2019.