Enhancing Web Privacy through Code-Aware Program Analysis

Abdul Haddi Amjad

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer and Information Sciences

Muhammad Ali Gulzar, PhD, Chair
Zubair Shafiq, PhD
Danfeng (Daphne) Yao, PhD
Na Meng, PhD
Bo Ji, PhD

February 13, 2026
Blacksburg, Virginia

Keywords: Internet, Web, Privacy, Program Analysis

Enhancing Web Privacy through Code-Aware Program Analysis

Abdul Haddi Amjad

(ABSTRACT)

Advertisers and tracking services (ATS) are pervasive on the modern web. To counter these practices, millions of users rely on privacy-enhancing technologies (PETs), such as ad-blockers, which primarily depend on filter lists composed of regex-based rules to block network requests associated with data exfiltration. However, this approach has become increasingly ineffective in the face of an ongoing arms race. ATS now employ obfuscation, encryption, and request multiplexing, embedding tracking logic within resources that also provide essential website functionality. We refer to these as mixed web resources, which place PETs in a fundamental dilemma: aggressively blocking them risks breaking websites, while allowing them undermines user privacy.

This thesis addresses this challenge by asking: *How can code-aware program analysis be leveraged to design privacy-enhancing technologies that effectively mitigate tracking while preserving essential web functionality?* This thesis makes three primary contributions. First, we introduce TrackerSift, a large-scale measurement framework that reveals mixed behavior in a substantial fraction of web resources. Second, we explore JavaScript blocking at multiple granularities and demonstrate that function-level blocking effectively mitigates ATS behavior in most mixed resources while preserving website functionality. Finally, building on these insights, we present NoT.JS, a code-aware, machine-learning–based system that accurately identifies tracking JavaScript functions and refactors code to selectively remove tracking logic

while maintaining legitimate functionality. Together, these contributions advance privacy-enhancing technologies by enabling principled mitigation of mixed web resources through code-aware program analysis.

Enhancing Web Privacy through Code-Aware Program Analysis

Abdul Haddi Amjad

(GENERAL AUDIENCE ABSTRACT)

Online advertising and tracking services (ATS) are a fundamental part of today's web, but they also raise serious concerns about user privacy. To protect themselves, millions of people rely on privacy-enhancing technologies (PETs), such as ad-blockers. These tools typically depend on simple filter-rule lists to block outgoing tracking requests. However, as ATS adapt, they employ increasingly sophisticated techniques in which tracking is hidden within mixed website resources that also provide essential functionality. Blocking these mixed resources outright can break websites, while allowing them can expose users to unwanted tracking.
This thesis explores how PETs can better navigate this trade-off. It asks how deeper code-level analysis can be used to identify and remove tracking behavior without disrupting how websites functionality. The thesis first measures how common these mixed web resources are, those that combine essential website functionality with hidden tracking, and shows that they are widespread across the web. It then demonstrates that analyzing website JavaScript code at a fine-grained, function-level granularity enables tracking behavior to be isolated from website functionality in mixed resources. Building on these insights, the thesis presents an automated machine-learning system that detects and removes tracking code from mixed resources while preserving legitimate website functionality. Overall, this thesis shows that understanding how ATS tracking is embedded within website code is essential for building more effective and reliable PETs, and highlights how code-aware analysis can help PETs keep pace with an evolving web.

Contents

1 Introduction **1**

1.1 Problem Formulation 2

1.2 Challenges of Program Analysis for Web JavaScript 3

1.3 Dissertation Key Insights 4

1.3.1 Identification of Mixed JS Scripts 6

1.3.2 Validation via Program Analysis of Mixed Scripts 8

1.3.3 Mitigating Tracking in Mixed JS Scripts 9

1.4 Contributions 10

2 Review of Literature **12**

2.1 Existing Countermeasures against Web Tracking. 12

2.2 Script-level Blocking against Web tracking 13

2.3 Function-level Blocking against Web tracking 14

2.4 Code Localization 15

3 Untangling Mixed Tracking and Functional Web Resources **16**

3.1 Introduction 16

3.2 TrackerSift Methodology 18

3.2.1 Domain Classification . . . 19
3.2.2 Hostname Classification . . . 19
3.2.3 Script Classification . . . 20
3.2.4 Function Classification . . . 20
3.3 Data . . . 21
3.4 Results . . . 23
3.4.1 Domain Classification . . . 24
3.4.2 Hostname Classification . . . 25
3.4.3 Script Classification . . . 26
3.4.4 Function classification . . . 26
3.5 Future Directions . . . 27
3.6 Summary . . . 31

4 Blocking JavaScript Without Breaking the Web 32

4.1 Introduction . . . 32
4.2 Motivation . . . 35
4.3 Methodology . . . 38
4.3.1 Phase I: Automated JS Blocking Analysis . . . 38
4.3.2 Phase II: Manual Inspection of JS Blocking . . . 42
4.3.3 Dataset . . . 45

4.4 Results . . . 47
4.4.1 Phase I: Large-scale JS Blocking Analysis . . . 47
4.4.2 Phase II: Visual Inspection of JS Blocking and Web Breakage . . . 56
4.5 Discussion . . . 60
4.6 Summary . . . 63

5 Blocking Tracking JavaScript at the Function Granularity 64

5.1 Introduction . . . 64
5.2 Threat-Model . . . 67
5.3 Design And Implementation . . . 69
5.3.1 NoT.JS's Chrome Instrumentation . . . 70
5.3.2 Graph Representation . . . 73
5.3.3 Feature Extraction . . . 77
5.3.4 Labeling . . . 78
5.3.5 Classification . . . 79
5.3.6 Surrogate Generation and Replacement . . . 79
5.4 Evaluation . . . 82
5.4.1 Accuracy Analysis . . . 82
5.4.2 Feature Analysis . . . 84
5.4.3 Robustness . . . 86

5.4.4 Comparison with Existing Countermeasures 89

5.4.5 Surrogate Generation and Replacement 89

5.5 Deployment 94

5.6 Summary 96

6 Discussion and Future Work **97**

6.1 Chrome V8 Parser-Based Replacement for Tracking JavaScript Functions . . 99

6.2 API Patching for Mixed Resources: Evaluating Viability and Challenges . . 100

7 Conclusion **101**

Bibliography **102**

Chapter 1

Introduction

Online advertising and tracking services (ATS) form the foundation of today's web platforms, providing a crucial channel for companies to reach and engage wider audiences. To monetize their large user bases, web platforms rely heavily on ATS services as a primary source of revenue. The effectiveness of these ads relies on a deep understanding of the target audience, achieved through behavioral profiling using JavaScript (JS) APIs [84]. These profiling techniques enable user tracking by extracting insights into demographics, locations, and behaviors to deliver personalized advertising. ATS significantly impacts millions of consumers globally [143], spanning various platforms including the web, mobile devices, smart appliances, and augmented/virtual reality devices. While users benefit from free web content with intermittent ads, their privacy is frequently compromised by a lack of transparency in data collection practices. Additionally, these ads can involve distressing images, frequent pop-ups, and content that causes discomfort [90, 198].

To address privacy concerns, millions of users rely on privacy-enhancing technologies (PETs), particularly ad-blockers with millions of downloads [143]. These tools aim to prevent the exfiltration of user data to ATS endpoints by intercepting network requests associated with tracking activities. PETs identify and block ATS endpoints by matching domains, host-names, URLs, and HTTP headers against manually curated filter lists [21, 22] composed of string-based filtering rules. Privacy engineers maintain and update these filtering rules to ensure their correctness and effectiveness. As of 2023, thousands of such filter lists exist,

supported by numerous tools and tailored for a wide range of purposes [54]. Despite their widespread adoption, PETs remain in an ongoing arms race with ATS, which employ evasion techniques such as domain rotation, script obfuscation, and parameter encryption to bypass traditional list-based defenses [76, 87, 114].

These circumvention techniques generally fall into two categories. The first involves frequently altering the network location of tracking resources, such as rotating domains or URLs. PETs respond by regularly updating filter lists, and recent research has explored automating the generation of new filtering rules [117, 122, 171, 173]. The second category involves mixing tracking functionality with legitimate website functionality and serving both from the same endpoint [94, 101]. This challenge is further intensified when trackers encrypt URLs, making it difficult for filter-list-based approaches to distinguish between benign and tracking-related traffic [24, 190]. As a result, content-blockers face a fundamental dilemma: blocking such requests risks breaking legitimate functionality, while allowing them enables continued privacy-invasive tracking.

1.1 Problem Formulation

To understand the problem, privacy risks arise at two distinct stages: (1) user data collection within the web, and (2) exfiltration of that data to ATS endpoints. Existing PETs primarily address the second stage by blocking network requests to known ATS endpoints using filter lists, as discussed earlier. However, mitigating privacy risks at the first stage, identifying and preventing user data collection itself, remains an open research challenge. A key problem is when websites request JavaScript files that serve mixed behavior: enabling essential functionality while simultaneously performing tracking. In such cases, PETs face a fundamental trade-off. Blocking the script would break legitimate website functionality,

while allowing it compromises user privacy. An alternative approach is to permit the script to execute but block subsequent data exfiltration attempts. Yet this approach is increasingly undermined by ATS techniques such as encrypted URLs and obfuscated request parameters, which evade traditional network-based appraoches. Consequently, PETs are unable to intervene effectively either at the time mixed JavaScript is loaded or when it later attempts to transmit collected identifiers. This limitation highlights the need for new approaches based on program analysis of JavaScript to analyze and mitigate tracking at its source.

1.2 Challenges of Program Analysis for Web JavaScript

Unlike traditional software, analyzing JavaScript in modern web applications poses far more complex challenges than a trivial program analysis. In conventional program analysis, developers have direct access to source code, execution environments, and program states, making it easier to reason about behavior and debug issues. In contrast, web JavaScript executes in highly dynamic and heterogeneous environments where it is deeply intertwined with HTML, CSS, and the DOM, and its behavior is shaped by user interactions, third-party scripts, and runtime modifications. Consequently, several key challenges arise that must be addressed to enable effective and reliable program analysis for web-based JavaScript:

1. **JS Integration with the Browser Environment:** Although tools exist to perform program analysis on JavaScript in isolation using environments such as Node.js, analyzing web-based JavaScript is significantly more complex. In the browser, JavaScript interacts extensively with the DOM and CSS to modify page structure and styling. Script behavior often depends on event handlers, asynchronous callbacks, and network responses. Accurately reasoning about such code requires capturing these interactions and unraveling complex browser APIs and execution contexts. Modeling this rich,

event-driven behavior at scale remains a major challenge for automated analysis tools.

2. **Obfuscation and Minification:** JavaScript deployed on the web is commonly minified or intentionally obfuscated to reduce file size and protect intellectual property. These transformations rename variables, flatten control structures, and alter code organization, obscuring the original program logic. Such changes make it difficult for static analysis tools to extract meaningful syntactic or semantic features. Consequently, understanding the true behavior of obfuscated scripts becomes a significant challenge.

3. **Dynamic Language Features:** JavaScript is inherently dynamic, supporting runtime code generation, reflection, and flexible typing. Functions and objects can be created, modified, or invoked dynamically during execution. Control flow and data flow can change based on user input or external resources. This unpredictability greatly complicates accurate program analysis and limits the effectiveness of traditional program analysis techniques.

1.3 Dissertation Key Insights

The central hypothesis of this dissertation is that *we can design a specialized, code-aware JavaScript analysis that can be leveraged to build a privacy-enhancing tool capable of mitigating tracking while preserving essential web functionality.* We demonstrate this by leveraging several foundational insights that enable fine-grained identification of tracking behavior within mixed scripts, precise differentiation between functional and privacy-invasive code, and targeted intervention at the level of JavaScript execution.

Attribution of JavaScript Behavior: A fundamental requirement for effective privacy analysis is the ability to accurately attribute runtime actions to the specific JavaScript code responsible for them. In modern web applications, user interactions, API calls, and network requests originate from multiple scripts executing concurrently. Without precise attribution, it is difficult to determine which components of a page are performing legitimate functions and which are engaging in tracking. Developing mechanisms to capture and attribute these events to their originating scripts is therefore a key insight that enables meaningful analysis and intervention.

Identification of Mixed-Purpose Scripts: A major obstacle for existing PETs is the prevalence of mixed-purpose JavaScript resources that combine essential website functionality with tracking behavior. Traditional blocking approaches treat scripts as indivisible units, leading to an inherent trade-off between privacy and usability. Identifying such mixed scripts is crucial for enabling more nuanced defenses that can distinguish between functional and privacy-invasive components. This insight forms the basis for moving beyond coarse-grained filtering toward more selective and context-aware mitigation strategies.

Execution Context Mapping through Program Analysis: Understanding complex JavaScript behavior requires transforming low-level runtime activity into structured and analyzable representations. By mapping dynamic execution traces into program representations such as enriched call graphs and contextual event flows, it becomes possible to reason about how data is collected and propagated within a script. This structured view enables the separation of tracking-related logic from functional logic, even in highly dynamic environments. Such execution context mapping is therefore a critical enabler for practical program analysis of web JavaScript.

Code-Aware Refactoring for Privacy Preservation Rather than blocking entire scripts, a more effective approach is to modify JavaScript code to selectively remove or neutralize tracking functionality. Through targeted refactoring, it is possible to transform mixed-purpose scripts so that essential features continue to operate while privacy-invasive behavior is eliminated. This insight enables a new class of privacy-enhancing tools that operate at the level of code execution rather than network requests. Code-aware transformation thus provides a practical path toward mitigating tracking without sacrificing website functionality

Summary. We materialize the aforementioned foundational insights into a unified framework for identification, validation, and mitigation of tracking in mixed JavaScript. TrackerSift [81] identifies and quantifies the scope of the problem by systematically uncovering how prevalent mixed JavaScript scripts are across the web and how deeply tracking logic is embedded within them. Building on this, we conduct extensive validation empirical study [82] demonstrating that fine-grained program analysis can effectively isolate tracking behavior from functional code within mixed scripts. Finally, we operationalize these insights through the design and implementation of NOT.js [83], an end-to-end automated system that detects tracking logic and selectively refactors mixed scripts to preserve functionality while mitigating privacy risks. Together, these techniques validate the central hypothesis that code-aware analysis and transformation of JavaScript can enable practical, privacy-enhancing defenses beyond traditional network-based approaches. The following sections provide an overview of each component.

1.3.1 Identification of Mixed JS Scripts

In the first step, we investigate the prevalence of mixed JS scripts on the web. There has been anecdotal evidence showing that ATS increasingly circumvent PETs by embedding

both functional and tracking logic within the same JavaScript code [24, 94, 101, 190]. This practice makes traditional network-level filtering ineffective, as blocking such scripts would disrupt legitimate website functionality, while allowing them enables user data exfiltration. To systematically analyze this problem, we build TrackerSift, a system inspired by spectrum-based fault localization (SBFL) techniques [128]. In this framework, the software under analysis is web mixed JS code, while individual test cases correspond to network requests generated during webpage execution. Network requests labeled as tracking or functional using filter lists [21, 22] serve as failing and passing cases, respectively. To attribute executed behavior to specific lines of mixed JS code, we capture the call stack associated with each network request during webpage execution. By executing pages and observing which scripts are active for each labeled request, TrackerSift constructs execution spectra that associate JS code with observed tracking and functional behaviors.

TrackerSift is implemented as a Chrome extension that utilizes the DevTools protocol [50] to collect fine-grained execution traces. Through instrumentation of JavaScript built-in APIs, it records network requests along with their originating call stacks, enabling code-level attribution of tracking and functional behavior. Using this framwork across a large-scale crawl of 10K websites, TrackerSift quantifies the extent of mixed JS scripts on the web. Our results show that 135K scripts generate both tracking and functional network requests, confirming that mixed scripts are prevalent on the web. These mixed scripts are largely hosted on CDNs or consist of bundled JavaScript produced by build tools such as Webpack, which aggregate multiple components into a single deployable resource. These findings highlight the limitations of network-level blocking and motivate the need for fine-grained, code-aware analysis techniques. This work was published at ACM IMC 2021 [79] and presented at the Adblock Summit 2021 [14].

1.3.2 Validation via Program Analysis of Mixed Scripts

In the second step, we conduct a comprehensive validation study to demonstrate that tracking behavior embedded within mixed JS scripts can be precisely identified and isolated. First, we perform a longitudinal analysis to examine how the prevalence of mixed JS scripts has evolved over time. By re-crawling a large set of websites and comparing measurements from 2021 and 2022, we observe a clear upward trend: the proportion of mixed scripts increased by 14% within a single year. This result confirms that the problem is not only widespread but also actively growing, underscoring the urgency of developing more robust defenses against mixed JS scripts. Building on this observation, we extend the TrackerSift framework to analyze mixed scripts at a finer, function-level granularity. Instead of treating an entire JS script as a single unit, the enhanced framework inspects the internal execution of mixed JavaScript to pinpoint the specific JS functions responsible for triggering network requests.

To enable function-level granularity, we instrument JavaScript built-in call stack API [185] so that, in addition to capturing the script associated with a network request, we also capture the exact function within that script that initiated the request. This instrumentation allows us to construct execution spectra that record which individual functions in a mixed script are executed to trigger each network request. As a result, we can distinguish tracking requests from functional requests even when they originate from the same mixed script. Using this framework across a large-scale crawl of 10K websites, TrackerSift quantifies the extent of mixed JS scripts on the web and demonstrates that 93.2% of tracking activity within mixed scripts can be isolated at the function level, highlighting the effectiveness of fine-grained program analysis. Together, these findings validate that fine-grained identification of tracking code at the JS function level within mixed JS script is both feasible and necessary for enabling practical privacy-preserving interventions. This paper is published in PETS 2023

[82] and also featured in Adblock Summit 2022 [26].

1.3.3 Mitigating Tracking in Mixed JS Scripts

In the final step, we realize the mitigation step by developing NoT.js [83], an end-to-end system that detects and mitigates tracking within mixed JavaScript without disrupting legitimate functionality. So far, we attribute individual JS functions within a mixed script by observing the network requests they initiate, capturing the corresponding call stacks, and labeling those requests using filter lists. Building on this attribution, we aim to learn the behavioral characteristics of these JS functions so that tracking functions within mixed scripts can be automatically detected based on their code and execution features. To enable this, NoT.js employs browser instrumentation to capture rich dynamic execution context, including the call stack, heap state, and local variable values for each function invocation involved in tracking and functional behavior. While the static representation of a JavaScript function remains unchanged, its runtime behavior can vary significantly depending on the surrounding execution context. Capturing this context enables NoT.js to reason semantically about individual function executions and accurately differentiate between tracking and functional behavior. To enable fine-grained intervention, NoT.js encodes dynamic execution information into a rich graph-based representation that models relationships among scripts, functions, and runtime events. Using these representations, NoT.js trains a supervised machine learning classifier to detect tracking at the function-level granularity. The system then automatically refactor mixed JS scripts that selectively block only the execution of tracking functions while preserving functionality within the mixed JS script.

We evaluate NoT.js on the top 10K websites from the Tranco top-million list to assess its effectiveness, robustness, and usability. Our results show that NoT.js detects track-

ing JS functions with-in mixed scripts with 94% precision and 98% recall, outperforming prior state-of-the-art approaches by up to 40%. Incorporating dynamic execution context contributes a 29% improvement in F1-score, highlighting the importance of context-aware analysis. NoT.JS also remains resilient against common obfuscation techniques—including control-flow flattening, dead code injection, functionality mapping, and bundling—with only a 4% decrease in F1-score. Furthermore, the automatically generated surrogate scripts successfully block 84% of tracking function calls while causing no observable breakage on 92% of tested websites. This work was published at ACM CCS 2024 [83], received the Distinguished Artifact Award, was featured at the Adblock Summit 2023 [20], and was presented to the Google Privacy Sandbox team.

1.4 Contributions

Our contributions are as follows:

1. We present TrackerSift [81], the first framework for systematically identifying and quantifying mixed JS scripts on the web. TrackerSift adapts spectrum-based fault localization techniques to the domain of web tracking by treating JavaScript code as the software under analysis and network requests as labeled test cases. It leverages browser instrumentation to capture call stacks and execution spectra, enabling fine-grained attribution of tracking and functional behaviors within the same script. Through a large-scale study of 10K websites, TrackerSift reveals that mixed scripts are widespread and confirms the limitations of traditional network-level blocking.

2. We conduct the first longitudinal and function-level validation of mixed JS scripts [82]. Building on TrackerSift, we perform a longitudinal analysis demonstrating a 14%

increase in mixed scripts from 2021 to 2022, confirming that the problem is actively growing. We further extend TrackerSift to inspect mixed scripts at the JS function-level, showing that over 93% of tracking activity within mixed scripts can be isolated to specific functions. This work establishes that fine-grained program analysis can effectively separate tracking logic from functional logic, providing empirical evidence that selective mitigation is feasible without breaking legitimate functionality.

3. We introduce NoT.js [83], the first automated system for function-level mitigation of tracking in mixed JS scripts. NoT.js captures rich dynamic execution context—including call stacks, heap state, and local variable values—to learn the behavioral characteristics of tracking functions. It trains a supervised classifier to detect tracking at function-level granularity and automatically refactor mixed scripts that block only tracking functions while preserving functional code. Evaluated on the top 10K websites, NoT.js achieves 94% precision and 98% recall, improves F1-score by 29% through dynamic context, and blocks 84% of tracking function calls without causing breakage on 92% of websites. NoT.js was published at ACM CCS 2024, received the CCS Distinguished Artifact Award.

Chapter 2

Review of Literature

2.1 Existing Countermeasures against Web Tracking.

Privacy enhancing content blockers, such as uBlock Origin [27] and Brave [89], primarily rely on manually curated filter lists [21, 22, 23, 25, 89] to block tracking network requests at the client side. These filter lists contain URL- or domain-based rules to determine whether a network request should be allowed or blocked. Prior research shows that trackers are able to evade filter lists by changing their network location, such as the URL or domain [4, 140, 152, 183]. These evasions necessitate manual updates to the filter lists to accommodate the new network locations, leading to a perpetual arms race between the maintainers of filter lists and trackers [1, 2, 3, 5, 118, 179, 189].

To mitigate this issue, recent approaches, such as AdGraph [121], WebGraph [170], and PageGraph [174], use machine learning to automatically generate filter list rules, aiming at the identification of tracking network requests. These approaches adopt a graph-based representation of the webpage execution to classify network requests. Both WebGraph and PageGraph enhance AdGraph by incorporating additional features into their graph representation. For instance, WebGraph additionally captures storage accesses (*e.g.,* cookie read-/write), exhibiting superior performance against URL/domain-based evasion. While these approaches detect and block tracking network requests, their instrumentation is limited to the interactions between the initiator script and the tracking request, making it ineffective

in pinpointing the origin of tracking activity within mixed resources.

2.2 Script-level Blocking against Web tracking

Trackers/advertisers evade content blockers by utilizing a common network location to serve both tracking and non-tracking resources. For instance, trackers have started using Content Delivery Networks (CDNs) or engage in CNAME cloaking [103, 105, 106] to serve both tracking and non-tracking requests from a common network location. Content blockers are thus presented with a dilemma: either block the network request, potentially disrupting legitimate website functionality, or permit the network request, consequently letting go of tracking/advertising. To address this issue, a few content blockers such as uBlock Origin have introduced support for "scriptlets", which are custom JS snippets injected at runtime to substitute the code that initiates tracking/advertising network requests [65]. This scriptlet strategy is effective in countering tracking, even when originating from the same network location as non-tracking resources, as it eliminates tracking at its origin — well before a tracking network request is initiated. However, akin to filter lists, scriptlets are manually curated, rendering them challenging to scale across the entire web. At present, Brave Browser and uBlock Origin are capable of blocking a mere 27 scripts[1], while popular filter lists comprise over 6,000 exception rules designed to enable functionality-critical scripts [180]. Consequently, tens of thousands of privacy-invasive scripts remain unblocked.

Static or dynamic program analysis approaches have also been proposed to detect tracking/advertising JS code. Ikram *et al.* [116] employed machine learning to analyze syntactical and structural aspects of JS code, aiming to classify tracking scripts. However, their static analysis approach remains vulnerable to basic JS obfuscation techniques. Chen *et al.*

[1] https://github.com/gorhill/uBlock/tree/master/src/web_accessible_resources

[95] devised event-loop-based signatures based on dynamic code execution to detect tracking scripts. They found that some trackers bundle tracking and non-tracking code within a single script, posing a similar challenge when tracking and non-tracking resources are served from the same network location.

The increasing prevalence of mixed scripts, particularly those facilitated through bundling, poses a fundamental challenge to privacy-enhancing content blocking [8]. According to the Web Almanac, bundling is already a common practice on top-ranked websites; specifically, 17% of the top 1000 websites employ the Webpack JS bundler [19]. Furthermore, there has been a 5× increase in the downloads of prominent bundlers such as Webpack over the past five years [2]. Amjad *et al.* showed that the prevalence of mixed scripts on top 100K websites increased from 12.8% in 2021 [80] to 14.6% in 2022 [82].

2.3 Function-level Blocking against Web tracking

Prior research has proposed a function-level characterization of JS code. Smith *et al.* proposed SugarCoat [177] to systematically generate substitutes for scripts involved in tracking activities. SugarCoat relies on existing filter lists to identify tracking scripts for rewriting, rendering it ineffective against false negatives present in these filter lists. SugarCoat makes use of PageGraph [174] to pinpoint the code locations where scripts access the privacy-sensitive data, such as `document.cookie` and `localStorage`. Subsequently, these code locations, which can include JS functions, are replaced by benign mock implementations that are manually generated by developers. However, to date, only six mock implementations [3] are created for the designated APIs through developer assistance. This reliance on manual effort from developers limits SugarCoat's applicability to a large number of tracking scripts.

[2] https://npmtrends.com/webpack
[3] https://github.com/SugarCoatJS/sugarcoat/tree/master/mocks

2.4 Code Localization

The problem of localizing tracking-inducing code shares similarities with prior research on fault-inducing code localization. For example, spectra-based fault localization (SBFL) [74, 107, 127, 129, 162, 181] collect statement coverage profiles of each test, passing or failing, to localize the lines of code that are most likely to induce a test failure. Bela et al. [184] and Laghari et al. [134] presented a call frequency-based SBFL technique. Instead of coverage information, they use the frequency of method occurrence in the call stack of failing test cases. A method that appears more in the failing call stack of failing test cases is more likely to be faulty. In NoT.JS, methods responsible for more frequently initiating tracking requests than functional requests is classified as tracking. Abreu et al. [73] studied how accurate these SBFL techniques are, and their accuracy is independent of the quality of test design. Jiang et al. [126] used call stack to localize the null pointer exception, and Gong et al. [113] generated call stack traces to successfully identify 65% of the root cause of the crashing faults. One common limitation across most fault-localization approaches is that they require an extensive test suite capable of exercising faulty behavior, along with an instrumented runtime to collect statement-level coverage.

Chapter 3

Untangling Mixed Tracking and Functional Web Resources

3.1 Introduction

Background & Motivation. Privacy-enhancing content blocking tools such as AdBlock Plus [10], uBlock Origin [9], and Brave [12] are widely used to block online advertising and/or tracking [112, 142, 148]. Trackers have engaged in the arms race with content blockers via counter-blocking [153, 159] and circumvention [77, 136]. In the counter-blocking arms race, trackers attempt to detect users of content blocking tools and give them an ultimatum to disable content blocking. In the circumvention arms race, trackers attempt to evade filter lists (e.g., EasyList [21], EasyPrivacy [22]) used to block ads and trackers, thus rendering content blocking ineffective. While both arms races persist to date, trackers are increasingly employing circumvention because counter-blocking efforts have not successfully persuaded users to disable content blocking tools [97, 161, 167].

Limitations of Prior Work. Trackers have been using increasingly sophisticated techniques to circumvent content blocking [77, 88, 136]. At a high level, circumvention techniques can be classified into two categories. One type of circumvention is achieved by frequently changing the network location (e.g., domain or URL) of advertising and tracking resources. Content blocking tools attempt to address this type of circumvention by updating filter lists

promptly and more frequently [119, 175, 186]. The second type of circumvention is achieved by mixing up tracking resources with functional resources, such as serving both from the same network endpoint (e.g., first-party or Content Delivery Network (CDN)) [77, 96, 102]. Content blocking tools have struggled against this type of circumvention because they are in a no-win situation: they risk breaking legitimate functionality as collateral damage if they act and risk missing privacy-invasive advertising and tracking if they do not. While there is anecdotal evidence, the prevalence and modus operandi of this type of circumvention has not been studied in prior literature.

Measurement & Analysis. This paper aims to study the prevalence of mixed resources, which combine tracking and functionality, on the web. We present TrackerSift to conduct a large-scale measurement study of mixed resources at different granularities starting from network-level (e.g., domain and hostname) to code-level (e.g., script and method). TrackerSift's hierarchical analysis sheds light on how tracking and functional web resources can be progressively untangled at increasing levels of finer granularity. It uses a localization approach to untangle mixed resources beyond the script-level granularity of state-of-the-art content blocking tools. We show how to classify methods in mixed scripts, which combine tracking and functionality, to localize the code responsible for tracking behavior. A key challenge in adapting software fault localization approaches to our problem is to find a rigorous suite of test cases (i.e., inputs labeled with their expected outputs) [129]. We address this challenge by using filter lists [21, 22] to label tracking and functional behaviors during a web page load. By pinpointing the genesis of a tracking behavior even when it is mixed with functional behavior (e.g., method in a bundled script), TrackerSift paves the way towards finer-grained content blocking that is more resilient against circumvention than state-of-the-art content blocking tools.

Results. Using TrackerSift, our measurements of 100K websites show that 17% of the

69.3K observed domains are classified as mixed. The requests belonging to mixed domains are served from a total of 26.0K hostnames. TrackerSift classifies 48% of these hostnames as mixed. The requests belonging to mixed hostnames are served from a total of 350.1K (initiator) scripts. TrackerSift classifies 6% of these scripts as mixed. The requests belonging to mixed scripts are initiated from a total of 64.0K script functions. TrackerSift classifies 9% of these script functions as mixed. Our analysis shows that the web resources classified as mixed by TrackerSift are typically served from CDNs or as inlined and bundled scripts, and that blocking them indeed results in breakage of legitimate functionality. While mixed web resources are prevalent across all granularities, TrackerSift is able to attribute 98% of the script-initiated network requests to either tracking or functional resources at the finest function-level granularity.

Our key contributions include (1) **large-scale measurement and analysis** of the prevalence of mixed web resources; and (2) a **hierarchical localization approach** to untangle mixed web resources.

3.2 TrackerSift Methodology

In this section, we describe the design of TrackerSift to untangle mixed web resources. TrackerSift conducts a hierarchical analysis of web resources to progressively localize tracking resources at increasingly finer granularities if they cannot be separated as functional or tracking at a given granularity. TrackerSift needs a test oracle capable of identifying whether a web page's behavior (e.g., network requests) is tracking or functional. TrackerSift relies on filter lists, EasyList [21] and EasyPrivacy [22], to distinguish between tracking and functional behavior.

As also illustrated in Figure 3.1, we next describe TrackerSift's hierarchical analysis at in-

creasingly finer granularities of domain, hostname, script, and method.

3.2.1 Domain Classification

As webpage loads, multiple network requests are typically initiated by scripts on the page to gather content from various network locations addressed by their URLs. We capture such script-initiated requests' URLs and apply filter lists to label them as tracking or functional. We then extract the domain names from request URLs and pass the label from URLs to domain names. For each domain, we maintain a tracking count and functional count. All the domains that are classified as tracking or functional are set aside at this level. The rest representing mixed domains serving both tracking and functional requests are further examined at a finer granularity. For instance, in Figure 3.1, the domain `ads.com` and `news.com` serve solely tracking and solely functional content, respectively. The domain `example.com` serves both and thus needs analysis at a finer granularity.

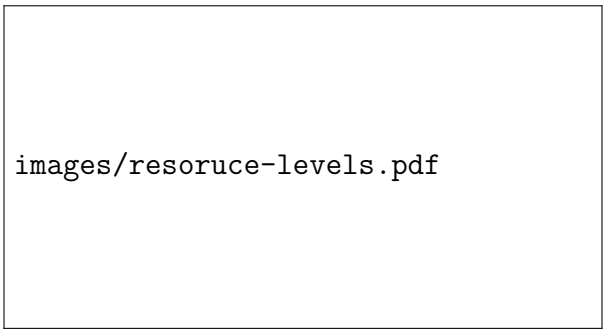


Figure 3.1: This toy example (not based on real data) illustrates how TrackerSift progressively classifies tracking (red) and functional (green) resources. For mixed resources (yellow), it proceeds to a finer granularity for further classification.

3.2.2 Hostname Classification

At the domain level, we find the requests served by mixed domains and extract their hostnames. We increment the tracking and functional count for each hostname within mixed domains based on the corresponding request's label. The hostnames serving both tracking and functional requests are further analyzed at a finer granularity, while the rest are clas-

sified as either tracking or functional. In Figure 3.1, `example.com` was previously classified as mixed and therefore, all hostnames belonging to `example.com` need to be examined. We classify `ad.example.com` and `maps.example.com` as tracking and functional, respectively. In contrast, `cdn.example.com` is mixed and thus needs analysis at a finer granularity.

3.2.3 Script Classification

We locate the script initiating the request to a mixed hostname and label it as either functional or tracking, reflecting the type of request they initiate. Like other levels, we measure the count of tracking and functional requests launched from each script and redistribute those into functional, tracking, and mixed scripts, where mixed scripts will be further analyzed at a finer granularity. In Figure 3.1, `sdk.js`, `clone.js`, and `stack.js` all initiate requests to the mixed hostname `cdn.example.com`. We classify `sdk.js` and `stack.js` as tracking and functional, respectively. Since `clone.js` requests both tracking and functional resources, it needs analysis at a finer granularity.

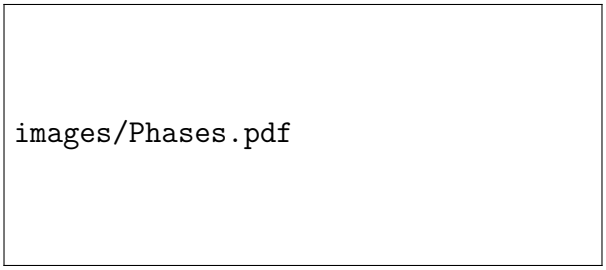


Figure 3.2: TrackerSift's web crawling and labeling

3.2.4 Function Classification

We analyze the corresponding requests for each mixed script and locate the initiator JavaScript methods of each request. We then measure the number of tracking and functional requests initiated by each of the isolated methods. In the final step, we classify the methods into functional, tracking, and mixed. In Figure 3.1, for the mixed script `clone.js`, we classify `m1()` as tracking and `m3()` as functional. Since `m2()` requests both tracking and functional

Table 3.1: Classification of requests at different granularities

Granularity	Tracking (Count)	Functional (Count)	Mixed (Count)	Separation Factor (%)	Cumulative Separation Factor (%)
Domain	755,784	566,810	1,129,109	54%	54%
Hostname	161,604	106,542	860,963	24%	65%
Script	235,157	490,295	135,511	84%	94%
Method	23,819	74,223	37,469	72%	98%

resources, it is classified as mixed.

3.3 Data

In this section, we describe TrackerSift's browser instrumentation that crawls websites and labels the collected data. Note that TrackerSift's hierarchical analysis is post hoc and offline. Thus, it does not incur any significant overhead during page load other than the browser instrumentation and bookkeeping for labeling.

Crawling. We used Selenium [62] with Chrome 79.0.3945.79 to automatically crawl the landing pages of 100K websites that are randomly sampled from the Tranco top-million list [164] in April 2021. Our crawling infrastructure, based on a campus network in North America, comprised of a 13-node cluster with 112 cores at 3.10GHz, 52TB storage, and 832GB memory. Each node uses a Docker container to crawl a subset of 100K webpages. The average page load time (until `onLoad` event is fired) for a web page was about 10 seconds. Our crawler waits an additional 10 seconds before moving on to the next website. Note that the crawling is stateless, i.e., we clear all cookies and other local browser states between consecutive crawls.

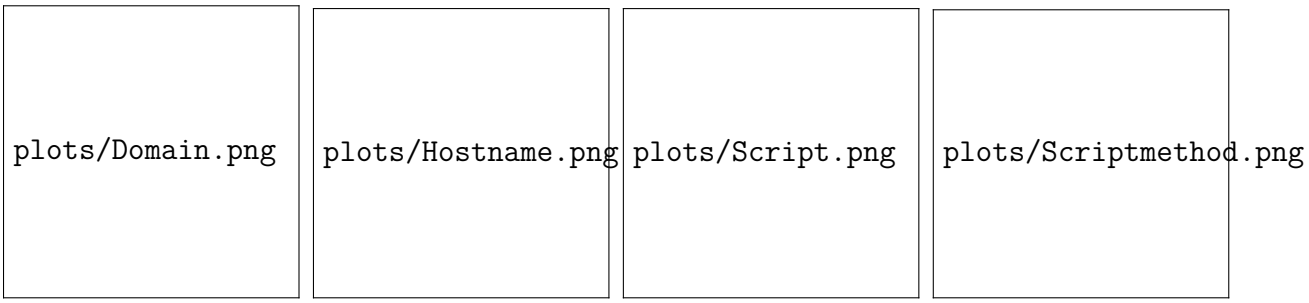
As shown in Figure 3.2, our crawler was implemented as a purpose-built Chrome extension that used DevTools [13] API to collect the data during crawling. Specifically, it relies on

Table 3.2: Classification of resources at different granularities

Granularity	Tracking (Count)	Functional (Count)	Mixed (Count)	Separation Factor (%)
Domain	6,493	50,938	11,861	83%
Hostname	4,429	9,248	12,383	52%
Script	194,156	134,726	21,168	94%
Method	17,940	40,500	5,579	91%

two `network` events: `requestWillBeSent` and `responseReceived` for capturing relevant information for script-initiated network requests during the page load. The former event provides detailed information for each HTTP request such as a unique identifier for the request (`request_id`), the web page's URL (`top_level_url`), the URL of the document this request is loaded for (`frame_url`), requested resource type (`resource_type`), request header, request timestamp, and a `call_stack` object containing the initiator information and the stack trace for script-initiated HTTP requests. The latter event provides detailed information for each HTTP response, such as response headers and response body containing the payload.

Labeling. We gather authoritative source labels by applying filter lists to the crawled websites. Filter lists are not perfect (e.g., they are slow to update [175] and are prone to mistakes [77]) but they are the best available source of labels. We use two widely used filter lists that target advertising (EasyList [21]) and tracking (EasyPrivacy [22]). These filter lists mainly build of regular expressions that match advertising and/or tracking network requests. As shown in Figure 3.2, network requests that match EasyList or EasyPrivacy are classified as tracking, otherwise they are classified as functional. Note that we maintain the call stack that contains the ancestral scripts that in turn triggered a script-initiated network request (e.g., `XMLHTTPRequest` fetches). For asynchronous JavaScript, the stack track that preceded the request is prepended in the stack. Thus, for script-initiated network requests, we ensure that if a request is classified as tracking or functional, its ancestral scripts in the



(a) domain (b) hostname (c) script URL (d) script method

Figure 3.3: Distribution of resources at increasingly finer granularities. Y-axis shows the count of unique (a) domains, (b) hostnames, (c) scripts, and (d) script methods. X-axis represents the common logarithmic ratio of the number of tracking to functional requests. Interval (-∞,-2] is classified as functional (green), (-2,2) is classified as mixed (yellow), and [2,∞) is classified as tracking (red).

stack are also classified as such. Since network requests that are not script-initiated can not be trivially classified as tracking or functional, we exclude them from our analysis.

3.4 Results

Classifying Mixed Resources. We compute the logarithmic ratio of the number of tracking to functional network requests to quantify the mixing of tracking and functional resources.

$$ratio = \log\left(\frac{\#\ of\ tracking\ requests}{\#\ of\ functional\ requests}\right) \tag{3.1}$$

At each granularity, we classify resources with the common logarithmic ratio less than -2 as functional because they triggered 100× more functional requests than tracking requests. Similarly, we classify resources with the common logarithmic ratio of more than 2 as tracking because they triggered 100× more tracking requests than functional requests. The resources with the common logarithmic ratio between -2 and 2 are classified as mixed. We analyze the suitability of the selected classification threshold using sensitivity analysis later in Section

3.5.

Results Summary. Table 3.1 summarizes the results of our crawls of the landing pages of 100K websites. Using the aforementioned classification, we are able to attribute 54% of the 2.43 million script-initiated network requests to tracking or functional domains. The remaining 46% (1129K) of the 2.43 million requests attribute to mixed domains that are further analyzed at the hostname-level. We are able to attribute 24% of the requests from mixed domains to tracking or functional hostnames. The remaining 76% (860K) of the requests attribute to mixed hostnames that are further analyzed at the script URL-level. We are able to attribute 84% of the requests from mixed hostnames to tracking or functional script URLs. The remaining 16% (135K) of the requests attribute to mixed script URLs that are further analyzed at the script method-level. We are able to attribute 72% of the requests from mixed script URLs to tracking or functional script methods. This leaves us with less than 2% (37K) requests that cannot be attributed by TrackerSift to tracking or functional web resources and require further analysis.

Next, we analyze the distribution of the ratio of tracking to functional requests by web resources at different granularities of domain, hostname, script URL, and script method in Figure 3.3. Table 3.2 shows the breakdown of web resources classified as tracking, functional, and mixed at different granularities.

3.4.1 Domain Classification

2451K requests in our dataset are served from a total of 69,292 domains (eTLD+1). Figure 3.3b shows three distinct peaks: [2, ∞) serve tracking requests, (-∞, -2] serve functional requests, and (-2, 2) serve both tracking and functional requests. We can filter 31% of the requests by classifying 6,493 domains that lie in the [2, ∞) interval as track-

ing. Notable tracking domains include `google-analytics.com`, `doubleclick.net`, and `googleadservices.com`, `bing.com`. We can filter 23% of the requests by classifying 50,938 domains that lie in the (-∞, -2] interval as functional. Notable functional domains include CDN and other content hosting domains `twimg.com`, `zychr.com`, `fbcdn.ne`, `w.org`, and `parastorage.com`. However, 46% of requests are served by 11,861 mixed domains that lie in the (-2, 2) interval. These mixed domains cannot be safely filtered due to the risk of breaking legitimate functionality, and not filtering them results in allowing tracking. Notable mixed domains include `gstatic.com`, `google.com`, `facebook.com`, `facebook.net`, and `wp.com`.

3.4.2 Hostname Classification

1129K requests belonging to mixed domains are served from a total of 26,060 hostnames. Figure 3.3b shows three distinct peaks representing hostnames that serve tracking, functional, or both tracking and functional requests. We can filter 14% of the requests by classifying 4,429 hostnames that lie in the [2, ∞) interval as tracking. We can filter 9% of the requests by classifying 9,248 hostnames that lie in the (-∞, -2] interval as functional. However, 76% of the requests are served by 12,383 hostnames that lie in the (-2, 2) interval are classified as mixed. Again, these mixed hostnames cannot be safely filtered due to the risk of breaking legitimate functionality, and not filtering them results in allowing tracking. Take the example of hostnames of a popular mixed domain `wp.com`. The requests from `wp.com` are served from tracking hostnames such as `pixel.wp.com` and `stats.wp.com`, functional hostnames such as `widgets.wp.com` and `c0.wp.com`, and mixed hostnames such as `i0.wp.com` and `i1.wp.com`.

3.4.3 Script Classification

860K requests belonging to mixed hostnames are served from a total of 350,050 initiator scripts. Figure 3.3c again shows three distinct peaks representing scripts that serve tracking, functional, or both tracking and functional requests. We can filter 27% of the requests by classifying 194,156 scripts that lie in the [2, ∞) interval as tracking. We can filter 57% of the requests by classifying 134,726 scripts that lie in the (-∞, -2] interval as functional. However, 16% of the requests are served by 21,168 scripts that lie in the (-2, 2) interval are classified as mixed. These mixed scripts cannot be safely filtered due to the risk of breaking legitimate functionality, and not filtering them results in allowing tracking. For example, let's analyze the initiator scripts of a mixed hostname `i1.wp.com`. The requests to this hostname are the result of different initiator scripts on the webpage `www.ibn24.tv`. Specifically, a tracking request to `i1.wp.com` is initiated by the script `show_ads_impl_fy2019.js` and a functional request to `i1.wp.com` is initiated by the script `jquery.min.js`. As another example, on the webpage `somosinvictos.com`, both tracking and functional requests to `i1.wp.com` are initiated by the mixed script `lazysizes.min.js`. Note that the scripts classified as tracking initiate requests to well-known advertising and tracking domains. For example, script `uc.js` served by `consent.cookiebot.com` initiated requests to googleadservices.com, doubleclick.net, and amazonadsystem.com.

3.4.4 Function classification

135K requests belonging to mixed scripts are served from a total of 64,019 script methods. Figure 3.3d again shows three distinct peaks representing methods that serve tracking, functional, or both tracking and functional requests. We can filter 17% of the requests by classifying 17,940 methods that lie in the [2, ∞) interval as tracking. We can filter 55%

of the requests by classifying 40,500 methods that lie in the (-∞, -2] interval as functional. However, 28% of the requests are served by 5,579 methods that lie in the (-2, 2) interval are classified as mixed. These mixed methods cannot be safely filtered due to the risk of breaking legitimate functionality, and not filtering them results in allowing tracking. For example, let's analyze script methods for a mixed script `tfa.js` on the webpage `hubblecontacts.com`. While both tracking and functional requests are initiated by the script, the tracking request was initiated by *get* method, and the functional request was initiated by *X* method. As another example, let's analyze script methods for a mixed script `app.js` on the webpage `radioshack.com.mx`. In this case, both tracking and functional requests are initiated by the mixed script method *Pa.xhrRequest*.

3.5 Future Directions

In this section, we discuss some case studies, opportunities for future work, and limitations.

Circumvention strategies. There are two common techniques for mixing tracking and functional resources.

(1) *Script inlining:* Despite potential security risks, publishers are willing to inline external JavaScript code snippets (as opposed to including external scripts using the `src` attribute) for performance reasons as well as for circumvention [135, 158]. For example, we find that the Facebook pixel [16] is inlined on a large number of websites to assist with targeting Facebook ad campaigns and conversion tracking.

(2) *Script Bundling:* Publishers also bundle multiple external scripts from different organizations with intertwined dependencies for simplicity and performance reasons. JavaScript bundlers, such as webpack [67] and browserify [48], use dependency analysis to bundle mul-

tiple scripts into one or a handful of bundled scripts. For example, `pressl.co` serves a script `app.*.js` that is bundled using the webpack [67]. This bundled script includes the aforementioned Facebook pixel and code to load functional resources from a first-party hostname. Existing content blocking tools struggle to block inlined and bundled tracking scripts without the risk of breaking legitimate site functionality. Finer-grained detection by TrackerSift presents an opportunity to handle such scripts by localizing the methods that implement tracking.

Threshold sensitivity analysis. We set the classification threshold to a symmetric value of (-2,2) for classifying mixed resources in Equation 4.1. To assess our choice of the threshold, we analyze the sensitivity of script classification results in Figure 3.4. Similar trends are observed for domain, hostname, and method classification. The plot shows the percentage of scripts classified as mixed as we vary the threshold from 1 to 3 in increments of 0.1 Note that the curve plateaus around our selected threshold of 2. Thus, we conclude that our choice of the threshold is stable and reasonably separates mixed resources from tracking and functional resources.

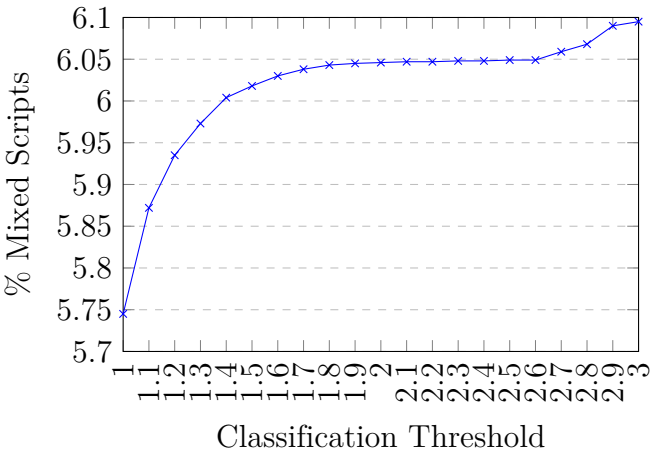


Figure 3.4: Sensitivity analysis of the classification threshold (default is -2 and 2) by studying the proportion of mixed scripts as a function of varying thresholds. The X-axis represents the threshold buckets. For example, 1.5 represents (-1.5, 1.5).

Breakage analysis. We conducted manual analysis to assess whether blocking mixed resources results in breakage of legitimate functionality. To assess functionality breakage, we load a random sample of websites with (treatment) and without (control) blocking mixed scripts as classified by TrackerSift. We label breakage as: *major* if the core functionality

Table 3.3: Manual analysis of breakage caused by blocking mixed scripts on randomly selected 10 websites.

Website	Mixed Script	Breakage	Comment
caremanagementmatters.co.uk	`jquery.min.js`	Minor	scroll bar and two widgets missing
gratis.com	`main.js`	Major	page did not load
forevernew.com.au	`require.js`	Major	multiple page banners missing
flamesnation.ca	`player.js`	Minor	video pop missing
biba.in	`MJ_Static-Built.js`	Major	page did not load
ecomarket.ru	`2.0c9c64b2.chunk.js`	Major	page did not load
peachjohn.co.jp	`jquery-1.11.2.min.js`	Major	navigation and scroll bar missing
shoobs.com	`widgets.js`	None	no visible functionality breakage
editorajuspodivm.com.br	`jquery.js`	Major	navigation and scroll bar missing
resourceworld.com	`jquery.min.js`	Major	navigation bar and images missing

such as search bar, menu, images, and page navigation is broken in treatment but not in control; *minor:* if the secondary functionality such as comment/review sections, media widgets, video player, and icons is broken in treatment but not in control; and *none:* if the core and secondary functionalities of the website are same in treatment and control. Note that we consider missing ads as no breakage. Table 3.3 shows our breakage analysis on a representative sample of 10 websites. We note major or minor breakage in all except one case. Thus, we conclude that mixed web resources indeed cannot be safely blocked by existing content blocking tools.

Blocking mixed scripts. When TrackerSift classifies a mixed script with different tracking and functional methods, we can simply remove tracking methods to generate a surrogate script that can then be used to shim the mixed script at runtime. Existing content blockers such as NoScript, uBlock Origin, AdGuard, and Firefox SmartBlock use surrogate scripts to block tracking by mixed scripts while avoiding breakage [11, 15, 18, 145]. However, these surrogate scripts are currently manually designed [17]. TrackerSift can help scale up the process of generating surrogate scripts by automatically detecting and removing tracking

Figure 3.5: Call stack analysis for the requests *ads-2* and *nonads-2* that can not be separated at method level i.e. *m2*. Call stack is analyzed to identify the first point of divergence i.e *track.js t* and it could be removed to block the tracking request.

methods in mixed scripts. Note that removing tracking methods is tricky because simply removing them risks functionality breakage due to potential coverage issues of dynamic analysis. To mitigate this concern, we plan to explore a more conservative approach using a guard—a predicate that blocks tracking execution but allows functional execution. Such a predicate has a similar structure to that of an `assertion`. We envision using classic invariant inference techniques [109, 160] on a tracking method's calling context, scope, and arguments to generate a program invariant that holds across all tracking invocations. If an online invocation satisfies the invariant, the guard will block the execution. A key challenge in this approach is collecting the context information, e.g., program scope, method arguments, and stack trace, for each request initiated by the mixed method at runtime. We plan to address these challenges in leveraging TrackerSift for generating safe surrogate scripts in our future work.

Blocking mixed methods. Our analysis shows that TrackerSift's separation factor is 91% even at the finest granularity. This leaves 5.6K mixed methods that cannot be safely blocked. One possible direction is to apply TrackerSift in the *context* of a mixed method initiating a request. We can define *context* as calling context, program scope, or parameters to the mixed method. In the case of calling context, we can perform a call stack analysis that takes a snapshot of a mixed method's stack trace when the method initiates a tracking or functional

request. We hope to see distinct stack traces from tracking and functional requests by a mixed method. We can consolidate the stack traces of a mixed method and locate the point of divergence, i.e., a method in the stack trace that only participates in tracking requests. We hypothesize that removing such a method will break the chain of methods needed to invoke a tracking behavior, thus removing the tracking behavior.

Figure 3.5 illustrates our proposed call stack analysis using a toy example. It shows the snapshot of stack traces of requests `nonads-2` and `ads-2`. These requests are initiated by a mixed method `m2()` on the webpage. The two stack traces are merged to form a call graph where each node represents a unique script and method, and an edge represents a caller-callee relationship. The yellow color indicates that a node participates in invoking both tracking and functional requests. `t` in `track.js` is the point of divergence since it only participates in the tracking trace. Therefore, `t` is most likely to originate a tracking behavior which makes it a good candidate for removal.

3.6 Summary

We presented TrackerSift, a hierarchical approach to progressively untangle mixed resources at increasing levels of finer granularity from network-level (e.g., domain and hostname) to code-level (e.g., script and method). We deployed TrackerSift on 100K websites to study the prevalence of mixed web resources across different granularities. TrackerSift classified more than 17% domains, 48% hostnames, 6% scripts, and 9% methods as mixed. Overall, TrackerSift was able to attribute 98% of all requests to tracking or functional resources by the finest level of granularity. Our results highlighted opportunities for finer-grained content blocking to remove mixed resources without breaking legitimate site functionality. NoT.JS can be used to automatically generate surrogate scripts to shim mixed web resources.

Chapter 4

Blocking JavaScript Without Breaking the Web

4.1 Introduction

JavaScript is often used to provide rich user experiences on the web. The volume of JavaScript on the web has steadily increased over the years. The median web page load today ships 500+ kilobytes of JavaScript [188]. While some of it is used to implement various libraries and frameworks (*e.g.,* jQuery, React), almost half of it is third-party scripts that implement advertising and tracking services. The research community is concerned about the negative impact of JavaScript on performance [91, 133, 187], security [100, 111, 144, 197], and privacy [108, 123, 130, 147, 148].

Due to these concerns, there is a small but active community of web users who want to use the web without JavaScript. In fact, all major browsers now provide a native way for users to block all JavaScript [47]. Moreover, users can employ browser extensions such as NoScript [39] that block all scripts – except those from a trusted source. HTML5 now also supports the `noscript` element that allows web developers to gracefully support such browsers that do not support scripting [57].

While blanket JavaScript blocking does alleviate these concerns, it inevitably breaks the

legitimate website functionality. The privacy community has developed content-blocking tools that selectively block tracking resources (*e.g.,* scripts) on a webpage. Privacy-enhancing content blockers, such as uBlock Origin [9], block network requests to known trackers by matching request URLs with manually curated filter lists [22, 31].

Since these privacy-enhancing content blockers are now used by more than one-third of web users [46, 141], there are strong financial incentives for web developers to evade content blockers. The typical evasion strategy is to manipulate the URLs, *e.g.,* change the URL path or hostname such that filter lists are no longer effective [77, 120]. This has led to an arms race where filter lists must be promptly updated in response to such evasion attempts [96, 137, 178]. Filter list curators have also made a concerted effort to selectively block the underlying scripts from downloading or execution that are responsible for initiating tracking requests. In response, a new evasion strategy has emerged where web developers attempt to mix tracking and functional code in the same script (*e.g.,* JS bundling [96]). Privacy-enhancing content blockers risk breaking a webpage if they block such scripts or compromise user privacy if they do not.

Privacy-enhancing content blockers aim to eliminate tracking while preserving website functionality. However, if they are forced to choose — *e.g.,* when tracking and functional code is mixed —they always prioritize functionality preservation. This is because most users tend to disable privacy-enhancing content blockers if they break legitimate website functionality. Recent research [81, 178] has shown that many websites now mix functional and tracking code that renders privacy-enhancing content blocking useless.

In this paper, we conduct a first-of-its-kind empirical investigation of JS blocking. To this end, we quantitatively and qualitatively evaluate the impact of different granularities of JS blocking on 100K websites. Our goal is to assess whether it is feasible to eliminate tracking effectively while preserving website functionality at different granularities of JS

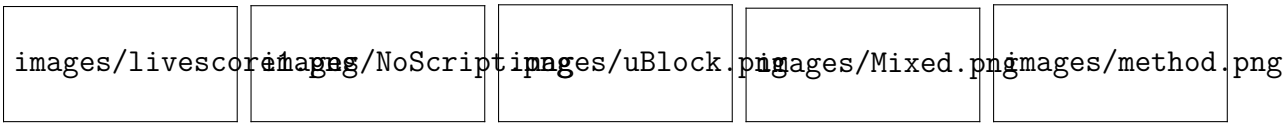
code *i.e.,* script and method. Beyond blanket JS blocking, we first investigate selective blocking of tracking scripts as well as mixed scripts. We further expand our investigation to the effectiveness of method-level blocking.

Our large-scale automated analysis of 100K websites reaffirms that blanket JS blocking indeed eliminates tracking, but it also breaks website functionality on approximately two-thirds of the tested websites. We then show that selective blocking of tracking scripts mitigates tracking without degrading website functionality, but there remains a significant fraction of scripts that mix tracking and functional behavior. Specifically, we find that 14.6% of the scripts exhibit both tracking and functional (*i.e.,* mixed) behavior. We then adapt Spectra-based fault localization (SBFL), a popular faulty code localization technique, to further localize tracking to the constituent methods of these mixed scripts. We find that method-level blocking of tracking methods significantly reduces website breakage while providing the same level of tracking prevention.

We also qualitatively analyze a sample of 383 websites under different JS blocking configurations for functionality breakage. We characterize functionality into four components *e.g.,* navigation, single sign-on, appearance, and additional functionality, and quantify breakage on 3-levels (none, minor, and major). Our evaluation shows that method-level JS blocking is far better at preserving functionality while achieving a similar level of tracking prevention. Specifically, we find that script-level JS blocking results in 3.8× major breakage and 1.5× minor breakage as compared to method-level JS blocking.

We summarize our key findings and contributions below:

- We find that method-level JS blocking is able to prevent tracking on par with script-level JS blocking while improving functionality preservation by 3.8× major breakage and 1.5× minor breakage.



(a) Control (b) NoScript (c) uBlock Origin (d) Mixed (e) Method

Figure 4.1: The snapshot of `livescore.com` with (a) control-setting (no content blocker), (b) NoScript (default setting), (c) uBlock Origin (default setting), (d) mixed script blocked (`_app-*.js`), and (e) JS method blocked (method in `_app-*.js`).

- By comparing two web crawls conducted one year apart, we find a 14% increase in the number of websites that employ mixed scripts on 100K websites.
- Even at the method-level granularity, there remain 6% mixed methods that combine tracking and functionality and require even deeper program analysis for effective blocking without breaking functionality.
- The data set crawled for this study offers a full-scale view of JS code integration on today's websites, presenting a detailed lineage of tracking, functional, and mixed JS code units across 100K websites.

4.2 Motivation

In this section, we present a case study to illustrate the tradeoff between tracking prevention and functionality breakage.

No JS blocking. Let's take the example of `livescore.com`, a top-10 ranked sports website [56]. We first load the homepage of `livescore.com` in a stock Chrome browser without any JavaScript intervention. Loading this webpage results in 294 network requests in 11 seconds, including 83 requests to fetch scripts and 175 requests initiated by these scripts. For motivation, consider two of these scripts that initiate network requests to *known*[1] tracking endpoints: `gtm.js` served by `googletagmanager.com` and `_app-*.js`

[1]See, for example, Disconnect tracking protection list [52]

Figure 4.2: Steps for localizing tracking and functional JS code using Spectra-based fault localization. ❶ shows the two network requests on `intuit.com`. Filter lists are used to label requests in ❷. Spectra-based fault localization is used to classify resources based on participation, as shown in ❸ and ❹.

served by `livescore.com`. `gtm.js` sends network requests to `googleadservices.com` and `google-analytics.com`. `_app-*.js` sends network requests to `doubleclick.net`. Upon careful inspection, we find that `_app-*.js` also sends a network request to `livescore.com /api/announcements/` that includes known tracking cookies such as `_gads` [93, 154]. While both scripts are responsible for network requests to tracking endpoints, `_app-*.js` is a mixed script that seems to implement both legitimate website functionality (*e.g.,* add media, populate game statistics) and tracking. Figure 4.1 (a) shows the homepage of `livescore.com` in the control configuration (without any blocking).

Blanket JS blocking. The naive way is to block all JS on `livescore.com` at the page load time. This capability is available in all major browsers [47]. While this approach blocks all the aforementioned tracking requests, it also completely breaks the website functionality. `livescore.com` becomes unusable and in fact notifies the user[2] that JS needs to be enabled for the website to display correctly. NoScript [39] also blocks all JS on `livescore.com`, including `gtm.js` served by

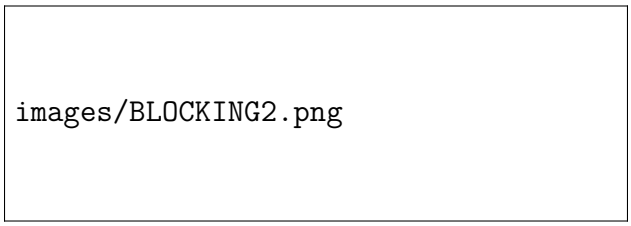


Figure 4.3: Illustration of the breakage metrics for automated JS blocking. Request count (❶) and HTML of website (❷) are compared with control configuration.

[2]The notice on livescore.com states: "Your browser is out of date or some of its features are disabled, it may not display this website or some of its parts correctly. To make sure that all features of this website work, please update your browser to the latest version and check that Javascript and Cookies are enabled."

`googletagmanager.com` and `_app-*.js` served by `livescore.com`. This again completely breaks the website functionality. Figure 4.1 (b) shows the homepage of `livescore.com` when NoScript [39] is used.

Selective JS blocking. We next use a tracker blocking tool, called uBlock Origin [9], on `livescore.com`. Note that these tracker blocking tools do not specifically target JS. Instead, they use a curated filter list to block network requests to known tracking endpoints that may incidentally include network requests to fetch JS. Thus, compared to blanket JS blocking, uBlock Origin aims to block all network requests to known tracking endpoints while allowing other network requests. After loading `livescore.com` with uBlock Origin installed, we observe that `gtm.js` is blocked, thus eliminating all subsequent tracking network requests from `gtm.js`. However, instead of blocking `_app-*.js`, uBlock Origin blocks the network request to `doubleclick.net` while it allows the network request `livescore.com/api/announcements/` containing tracking cookies. Figure 4.1 (c) shows the homepage of `livescore.com` when uBlock Origin [9] is used. Although there is no website breakage, uBlock Origin has essentially decided not to block `_app-*.js` to avoid website breakage even though it results in tracking requests. As we elaborate later, trackers have been increasingly putting tracker blocking tools in such a bind.

Tracking and Mixed JS blocking. To understand why uBlock Origin chose not to block `_app-*.js`, we next use uBlock Origin but also configure it to block `_app-*.js`. As shown in Figure 4.1 (d), this leads to a major functionality breakage on `livescore.com`; the navigation button, game statistics, and the featured news section are not rendered correctly. Put simply, there is a no-win situation when it comes to `_app-*.js`. Blocking it results in website breakage, and not blocking it results in tracking.

Method-level JS blocking. Recent work [81, 178] has applied dynamic analysis to identify tracking methods in mixed scripts manually. Our analysis of network requests initiated by `_app-*.js` shows that the tracking requests were initiated by the method shown in Listing 4.1. As shown in Figure [56] (e), when this method in `_app-*.js` is blocked (*e.g.,* it is renamed such that all calls to this method are invalidated), the entire webpage renders completely while all tracking requests are also blocked. It is noteworthy that manually refactoring mixed scripts is not feasible at scale. Therefore, only a handful of mixed scripts have been refactored in prior work [61].

```
- u = function(e) {
+ donotExecuteMe = function(e) {
      ...
      return fetch(e).then(c.cg).then(
      (function(e){return e || {}}))
```

Listing 4.1: JS method `u` that initiates tracking requests in script `_app-*.js`. We replace this method name with `donotExecuteMe`.

4.3 Methodology

This section describes our methodology for automated analysis of JS blocking on 100K webpages (Phase I) and manual inspection of JS blocking on 383 websites (Phase II).

4.3.1 Phase I: Automated JS Blocking Analysis

Figure 4.2 shows our automated JS blocking analysis pipeline comprising a JS collection step and JS code localization step. Figure 4.3 shows our JS blocking impact analysis step.

JavaScript Corpus Collection. We crawl landing pages of 100K randomly sampled websites from Tranco top-million list [164] using a custom-built Chrome extension. We spend 20 seconds on a page, exceeding the median `onLoad` time by 13.5 seconds on average. This

allows us to capture the vast majority of the content fetched, which is consistent with over 90% of all webpages [59]. Nonetheless, we measure the impact of increasing the crawl time to 90 seconds on 200 web pages randomly sampled from 100K. We notice average differences of 2% and 5.2% in tracking and functional requests, respectively, causing an insignificant impact on our findings. Thus, we set the crawl time to 20 seconds.

For each webpage, our crawler outputs a JSON file that maps each network request to its initiator script and method (step ❶). We then label each network request and its initiator code (*e.g.,* JS script and methods) as tracking or functional using filter lists [22, 31] (step ❷). We use EasyList [31] and EasyPrivacy [22] that are used by existing content blockers such as uBlock Origin [9], Brave [12], and Adblock Plus [6]. These filter lists only do binary classification and tend to classify mixed resources as functional to avoid website breakage. This is an inherent limitation of filter lists that our work aims to highlight in the context of JavaScript blocking.

Localizing Tracking and Functional JS Code. Next, we classify each script and method using spectra-based "fault" localization (SBFL) [**?** **?**]. SBFL requires a set of failing and passing test cases. For every test, it simply collects the list of code units that participated in the test execution. Based on the test output, it labels the participating code units as either passing or failing. Finally, it compares the participation of code units in passing and failing tests and assigns a *score* to them.

We adapt SBFL to localize tracking code units (*i.e.,* scripts, methods). Instead of test cases, we analyze each network request and the methods and scripts in the call stack trace of the network request. For example, Figure 4.2-❶ shows two network requests on `intuit.com`. We use filter lists (step ❷) to classify a request (and its call stack) as tracking (*i.e.,* failed test case) and functional (*i.e.,* passed test case). We then calculate "tracking score" (Eq 4.1) for each code unit (*i.e.,* script or method) based on its participation in the call stack

ID	Level	JS block	Blocked Tracking	Annotated Mixed	Entity Functional
CTRL	None	None	✗	✗	✗
ALL	script	Blanket	✔	✔	✔
TS	script	Selective	✔	✗	✗
MS	script	Selective	✗	✔	✗
TMS	script	Track & Mixed	✔	✔	✗
TM	method	Method	✔	✗	✗

Table 4.1: Six different JS blocking configurations. ✗ represents an unblocked entity, and ✔ represents a blocked entity.

trace of tracking and functional requests, as shown in step ❸. The script `utag.js` initiates 132 tracking requests and 160 functional requests. In this script, method `loader` initiates 131 tracking requests and 1 functional request. Method `fireCORS` initiated 159 functional and 1 tracking request. Figure 4.2 demonstrates the calculation of the tracking score on the webpage in step ❹.

$$tracking\ score = \log\left(\frac{number\ of\ tracking\ requests}{number\ of\ functional\ requests}\right) \tag{4.1}$$

We classify code units that participate 100× times more in tracking than functional (*i.e.,* tracking score of > 2) as tracking. We classify code units that participate 100× times more in functional than tracking (*i.e.,* tracking score of < -2) as functional. This threshold is determined experimentally in prior work [81]. The code units that fall in neither category are classified as mixed. The localization step results in a list of tracking, functional, and mixed JS methods and scripts. In this example, script `utag.js` is classified mixed, method `fireCORS()` is functional, and method `loader()` is tracking.

JS Blocking Impact Analysis. To measure the impact of blocking JS code units, our custom-built Chrome extension loads every page from the 100K websites and blocks the associated tracking JS script or method from the list of labeled methods and scripts. It blocks

Script Domain	Script	Method	Websites (%)
google-analytics.com	analytics.js	wd	38%
google-analytics.com	analytics.js	ta	25%
facebook.net	fbevents.js	c	19%
googlesyndication.com	sodar2.js	Ma	11%
twitter.com	widget.js	i.e	7%

Table 4.2: Top JS methods found on the maximum number of websites in control configuration.

the JS scripts from loading in the browser, similar to existing content blockers. To block a script method, it simply replaces the method name with `doNotExecuteMe` to redirect its invocations, as shown in Listing 4.1. Renaming the method name may cause a MethodNotFound exception that terminates the tracking thread in a webpage's JS execution as intended. We conduct this experiment on the same 100K webpages in six parallel configurations shown in Table 4.1. These configurations are illustrated in the `livescore.com` case study and inspired by unique JS blocking strategies that are mostly in practice or proposed by prior work. Control configuration (`CTRL`) is used to localize JS code units (scripts and methods) using the aforementioned SBFL technique and for breakage comparison in the later subsection. In `ALL`, all scripts (tracking, mixed, and functional) are blocked to evaluate blanket JS blocking. This configuration represents NoScript, which blocks all scripts by default. In `TS`, tracking scripts are blocked to evaluate selective JS blocking. This configuration represents the majority of content blockers such as uBlock Origin [9], Brave [12], and Adblock Plus [6] that use EasyList [31] and EasyPrivacy [22]. In `MS`, mixed scripts are blocked to see its adverse consequence on functionality. In `TMS`, tracking and mixed scripts are blocked to evaluate tracking and mixed JS blocking. `TMS` is the optimum choice for content blockers in tracking prevention, but it risks functionality breakage, as shown in Section 4.2. Finally, we compare the results of `TMS` with `TM`, where we block tracking methods (all located in tracking and mixed scripts) to evaluate method-level JS blocking.

In CTRL configuration, we have websites that do not crash. However, website crashes and breakages may still occur in the blocking configurations due to blocking. Website breakage is a subjective metric that requires a visual inspection, which is not feasible on 100K webpages. Therefore, we discuss two metrics that are correlated with website breakage [137].

Tracking and Functional request count. Network requests fetch critical functional resources like scripts, images, and other media as well as JS scripts and images that perform tracking activity. We use the number of tracking and functional requests as a measure of tracking and functional activity on a webpage. We compare these numbers with the control configuration (CTRL) to get the missing requests, as shown in Figure 4.3-❶. This metric helps in collecting non-visual breakage clues. For example, we do not see any visual breakage on website *poshmark.ca* after blocking mixed script *sdk.js?hash=**. Instead, we observe two missing requests, one that sets the cookie and the other functional request that redirects the login button.

HTML of websites. We scan the HTML tags with `src` attributes on a webpage to estimate visible functional deterioration. These HTML tags include `<img>`, `<video>`, and `<iframe>`. Each tag has a source, `src`, attribute that specifies the URL of a resource file. We compare the missing tags in our experiments with the control configuration (CTRL), as shown in Figure 4.3-❷. Note that if the attribute of a missing URL belongs to the functional request in the control configuration (CTRL), then it is classified as functional breakage.

4.3.2 Phase II: Manual Inspection of JS Blocking

Data Sampling

Manually inspecting 100k websites is time-consuming and practically infeasible. We randomly sample 500 websites from the top 100K websites used in Phase I. We exclude dupli-

Blocking Configuration	Total Network Requests			Script-Initiated Network Requests			Total Scripts	Total JS Methods
	Tracking	Functional	Total	Tracking	Functional	Total		
CTRL	1,175,033	4,279,844	5,454,877	953,931	882,111	1,836,042	256,042	366,025
ALL	265,101	3,248,767	3,513,868	177,352	315,378	492,730	91,984	137,006
TS	355,169	4,049,340	4,404,509	248,103	820,428	1,068,531	164,670	239,960
MS	1,012,708	3,916,499	4,929,157	815,553	684,084	1,499,637	227,658	323,174
TMS	349,888	3,887,372	4,237,260	245,389	657,361	902,750	155,810	224,681
TM	348,135	4,115,351	4,463,486	243,002	749,238	991,240	164,543	233,927

Table 4.3: Characteristics of the crawled dataset across six blocking configurations.

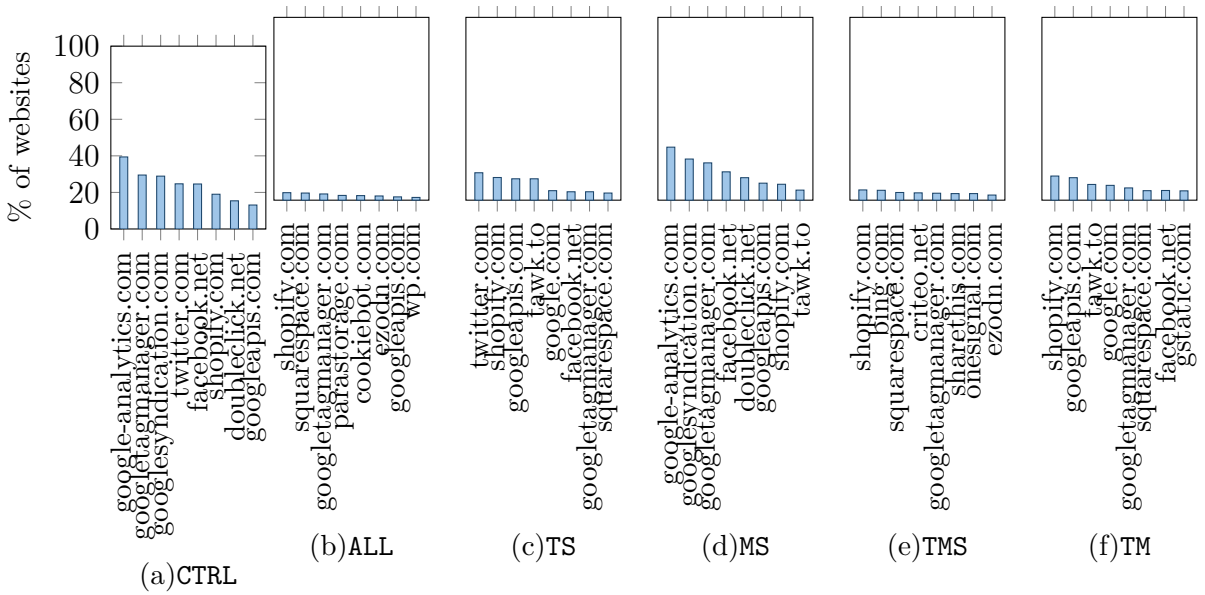


Figure 4.4: The top domains of request-initiating scripts across six blocking configurations. X-axis shows the top domains of the request-initiating scripts, and Y-axis shows the % of websites.

cate websites and websites with the same second-level domains (SLD), but different top-level domains (TLD) *e.g.,*`google.com.uk` and `google.com`. We excluded a total of 117 websites and manually inspected 383 websites, which is a statistically significant sample size for 100K websites with ± 5% margin of error [63].

Manual Inspection

Two testers independently inspected 383 websites. Inspecting six configurations for each website manually and in parallel is prohibitively expensive. Therefore, we choose the three

most important configurations *i.e.,* `CTRL` (for comparison), `TMS` (tracking and mixed JS blocking), and `TM` (method-level JS blocking). To assist inspection, our study platform launches three independent instances of Chrome (`CTRL`, `TMS`, and `TM` from Table 4.1) displayed adjacent to each other. Each tester spent at least 5 minutes inspecting the three windows, scrolling each page end to end, and clicking on different webpage components. The two testers spent a total of 85 hours manually inspecting the websites and documenting their findings according to the following rubric. They report visual and functional differences in the following four categories and use a 3-level breakage scale (*i.e.,* no breakage, minor breakage, and major breakage). Any disagreements were discussed and resolved by consensus.

- **Navigation.** Website navigation contains lists of links to internal webpages. It typically consists of a menu or navigation bar that contains links to various sections of the website, such as the homepage, products or services, about us, and contact. Minor breakage involves non-functional navigation links, abnormal styling layouts, or missing icons. These issues can be frustrating for users and may make it difficult to navigate the website. Major breakage involves more serious issues, such as the navigation button not being operational or the navigation bar not appearing at all. This type of breakage can significantly impact the website's usability.

- **Single sign-on (SSO).** Website SSO allows users to sign in using credentials from services such as Google and Facebook. Minor breakage typically involves issues such as non-functional SSO services, unresponsive login buttons, or missing login options. For example, if the Google SSO service is not functioning, users may be unable to sign in to the website using their Google account. Major breakage involves more serious issues, such as the missing SSO service or the failure of all SSO options. This type of breakage can significantly impact the website's usability.

- **Appearance.** This category includes the appearances of media elements, the scrolling behavior of websites, and the HTML element. We exclude advertisements when inspecting appearance-based breakage. Minor breakage involves missing media resources, unstyled HTML, or jittery/unsmooth page scrolling experience. Major breakage involves all the media resources missing altogether or an unscrollable page.

- **Additional functionality.** Anything that does not fall into the mentioned categories is added to this category, such as dark mode, website settings, and chatbot. Minor breakage entails abnormal behavior or non-responsive feature. Major breakage includes page crashes and missing components.

4.3.3 Dataset

This section summarizes the characteristics of dataset crawled across six blocking configurations. Table 4.3 lists the total network requests and script-initiated requests in six configurations over 100K websites and the JS scripts and methods that initiate those requests. In control configuration (`CTRL`), out of 5.45 million requests, 22% of the requests are tracking, leaving the remaining 78% as functional. 34% of the total requests are initiated by JS scripts. In script-initiated requests, 52% are tracking, and the remaining 48% are functional. These script-initiated requests are initiated by 366K JS methods inside 256K scripts.

Figure 4.4 shows the top domains of the scripts that initiate network requests. In control configuration (`CTRL`), 39% of websites initiate requests from the script served by `google-analytics.com`, 30% of websites initiate requests from the script served by `googletagmanager.com`, and 29% of websites initiate requests from the script served by `googlesyndication.com`.

Our baseline JS blocking configuration is `ALL` in which all tracking, mixed, and functional scripts are blocked. Note that a small number of scripts may still load in `All` if such scripts

were previously not observed during the localization step in Section 4.3.1.

When tracking JS scripts are blocked (`TS` configuration), the majority of tracking script domains disappear, including `google-analytics.com`. We observe a relatively lower occurrence of script domains in `TMS` than `TM` because `TMS` blocks all tracking and mixed scripts that include all tracking methods and some functional methods. Whereas in `TM`, only tracking methods are blocked. For example, due to the mixed nature of scripts from `facebook.net`, scripts from `facebook.net` appear in `TM`, but not in `TMS`.

```
wd = function(a, b, c, d) {
    var e = 0.XMLHttpRequest;
    if (e) return 1;
    var g = new e;
    if (("withCredentials" in g)) return 1;
    a = a.replace(/^http:/, "https:");
    g.open("POST", a, 0);
    g.withCredentials = 0;
    g.setRequestHeader("Content-Type", "text/plain");
    g.onreadystatechange = function() {
  ...
ta = function(a) {
    var b = M.createElement("img");
    b.width = 1;
    b.height = 1;
    b.src = a;
    return b}
```

Listing 4.2: Methods `wd` and `ta` in `analytics.js` served by google-analytics.com are present on 38% and 25% of 100K websites, respectively.

Table 4.2 shows the top five request-initiating JS methods across 100k websites. Method `wd`

in script `analytics.js` is served by `google-analytics.com`. It appears in 38% of the 100K websites where it sets up a request and its header using XMLHttpRequest [68] API, shown in Listing 4.2. Method `ta` in script `analytics.js` is served by `google-analytics.com`. It appears in 25% of the websites where it adds the `<img>` tag with a specific source given as a parameter to the function, shown in Listing 4.2. Both of these methods are classified as tracking in the localization step in Section 4.3.1.

4.4 Results

This section presents the results of our empirical investigation of different types of JS blocking listed in Table 4.1.

4.4.1 Phase I: Large-scale JS Blocking Analysis

We aim to address the following research questions in our analysis of JS blocking.

1. How resilient is website functionality against blanket JS blocking (`ALL`)?

2. How effective is selective script-level JS blocking in tracking prevention and functionality preservation (`TS` and `MS`)?

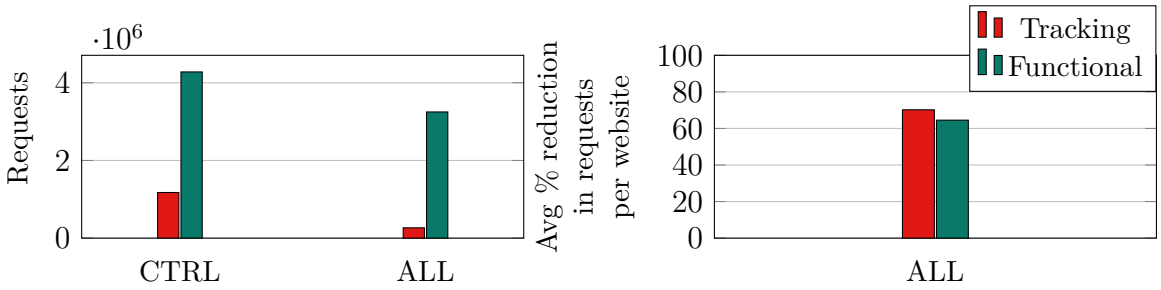
3. How common is it for website developers to mix tracking and functionality in the same script?

4. How effective is method-level JS blocking in tracking prevention and functionality preservation (`TMS` and `TM`)?

RQ1: Blanket JS Blocking

We first study the naive approach to JS blocking by blocking all JS scripts (`ALL` configuration in Table 4.1). Specifically, we block all 256K scripts on 100K webpages and compare the breakage metrics (*i.e.,* network request count and HTML resource count) with the control (`CTRL`). Given blanket JS blocking, we expect a sharp drop in the number of tracking or functional requests. Figure 4.5 (a) shows that 22% of functional requests and 76% of tracking requests remain after blocking all JS scripts (`ALL`). Note that a few requests are initiated by the scripts previously not captured in the localization step in Section 4.3.1 and hence, were not blocked in blanket JS blocking (`ALL`) configuration. Figure 4.5 (b) presents the average percentage of reduction in request count per webpage. On average, per webpage, the tracking and functional request count decrease by 70% and 65%, respectively. This shows that webpages today can retain one-third of functionality even with extreme blocking strategies. Another observation is that the tracking reduction per webpage is higher than functional reduction, which means that many webpages often sacrifice tracking but attempt to retain functionality.

To map this behavior per webpage, we find the number of webpages with different levels of request reduction for both tracking and function. Figure 4.6 illustrates the result. We find that the majority of the webpages (57%) have either less than 10% request reduction or more than 90% request reduction in both tracking and functional. This result shows both (1) high resilience against tracking reduction and functional breakage due to anti-content blocking strategies such as loading resources by changing network endpoints [77, 136], and also (2) low resilience where blocked scripts are critical for a functioning webpage [81, 178]. Further inspection of HTML DOM elements reveal that 191K functional HTML tag sources are missing from 100K webpages when `ALL` scripts are blocked, reflecting severe functionality loss. Table 4.4 shows the breakdown of the category of these missing sources. In `ALL` configuration,



(a) Number of requests (b) % Reduction

Figure 4.5: (a) compares the request count of control configuration with blanket JS blocking (`ALL`). (b) shows average % reduction in request per website for blanket JS blocking (`ALL`).

71K functional `<img>` tags, 21K functional `<iframe>` tags, and 100K functional `<script>` tags are missing.

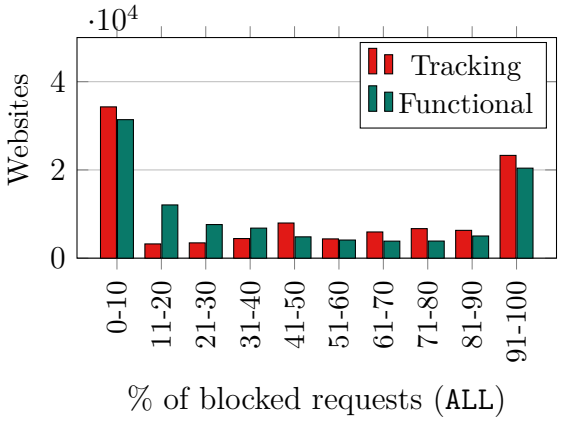


Figure 4.6: The % of blocked request in blanket JS blocking configuration (`ALL`).

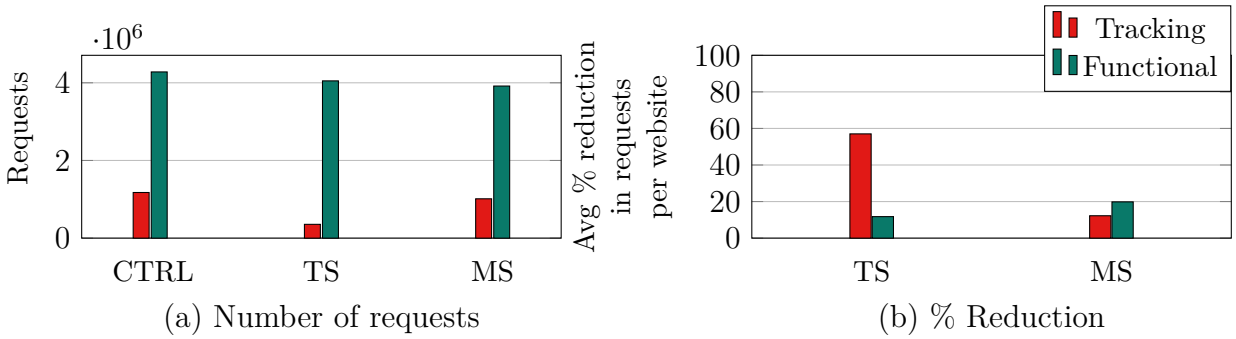
Tag Category	Blanket JS Blocking (ALL)
`<image>`	70600
`<video>`	5
`<iframe>`	21052
`<script>`	100278
`<source>`	39

Table 4.4: Missing HTML tags whose URLs are classified as functional in blanket JS blocking (`ALL`).

Takeaway. Two-thirds (66%) of the webpages experience a significant functionality breakage when blanket JS blocking is employed.

RQ2: Effectiveness of Selective JS Blocking

Since Blanket JS blocking is ineffective, we study the effectiveness of selective JS blocking by blocking tracking scripts (`TS` configuration in Table 4.1). Later, we block mixed scripts (`MS` configuration in Table 4.1) to see its adverse effects on functionality.



(a) Number of requests (b) % Reduction

Figure 4.7: (a) compares the request count of control configuration with selective JS blocking (`TS` and `MS`). (b) shows average % reduction in request per website for selective JS blocking (`TS` and `MS`).

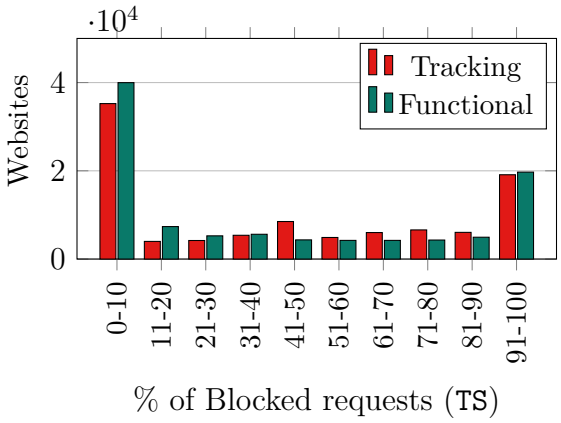


Figure 4.8: The % of blocked request in selective JS blocking configuration (`TS`).

Tag Category	Tracking JS Blocked (TS)	Mixed JS Blocked (MS)
`<image>`	12607	20035
`<video>`	0	0
`<iframe>`	11774	14682
`<script>`	21650	37197
`<source>`	23	37

Table 4.5: Missing HTML tags whose URLs are classified as functional in selective JS blocking (`TS` and `MS`).

Blocking Tracking Scripts. In this experiment, we block 93K tracking scripts (`TS`) from 256K JS scripts across 100K live webpages and investigate its impact on tracking mitigation and functional breakage. Figure 4.7 (a) reports that 95% of functional requests persist, whereas 30% of tracking requests manage to survive. Figure 4.7 (b) shows an average reduction in requests per webpage. In the case of `TS`, we observe a 57% reduction in tracking requests and an 11% reduction in functional requests per webpage on average. Measurement with HTML tag metric in Table 4.5 shows that blocking tracking JS scripts (`TS`) results in 46K missing functional sources across 100K webpages. In `TS` configuration, 13K functional `<img>` tags, 12K functional `<iframe>` tags, and 22K functional `<script>` tags are missing.

Blocking Mixed Scripts. In this experiment, we block only mixed JS scripts (`MS`). We expect a decrease in both functionality and tracking, as mixed scripts represent both. Figure 4.7 (a) visualizes these results. Overall, we see 86% of tracking and 92% of functional requests. This observation is consistent with other HTML tag metric in Table 4.5. In `MS` configuration, 20K functional `<img>` tags, 15K functional `<iframe>` tags, and 37K functional `<script>` tags are missing. Figure 4.9 show visual breakage on `pressl.co` due to blocking mixed JS scripts that eliminate tracking at the cost of critical functional breakage.

We further ask *Do all webpages react similarly when tracking scripts are blocked?* Our goal is to unfold the resilience of different webpages with blocked tracking scripts (`TS`).

Figure 4.8 measures the distribution of webpages across different levels of functional breakage and tracking mitigation from blocking tracking scripts. 39K webpages experience less than 10% functional deterioration, and 35K webpages experience less than 10% tracking mitigation.

The left of the bar chart represents webpages that heavily employ mixed scripts, making JS script blocking ineffective. 19K webpages are only left with greater than 90% functionality deterioration and tracking mitigation, representing the class of webpages relying less on mixing scripts and thus are susceptible to JS script blocking. Although JS script blocking is effective on a few webpages, it does not apply to a significant proportion of webpages that employ mixed scripts. Therefore, we must address the tracking behavior concealed in mixed scripts.

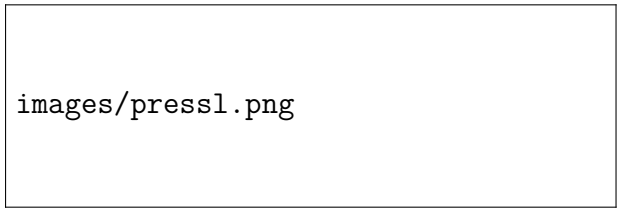


Figure 4.9: Visual impact of blocking mixed JS script. The left side shows a normal website, whereas the right side shows a breakage due to blocking.

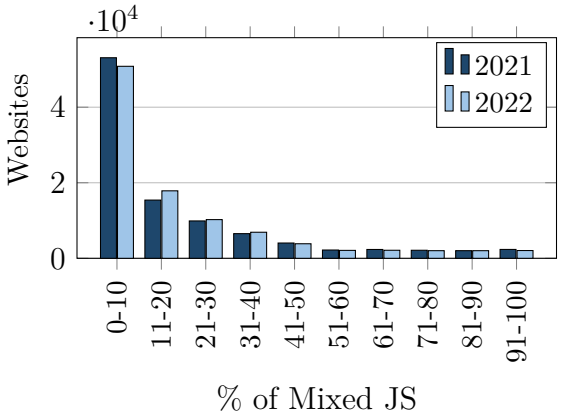


Figure 4.10: Comparison of % mixed JS scripts when tracking score is in [-2,2] for web corpus collected in 2021 and 2022.

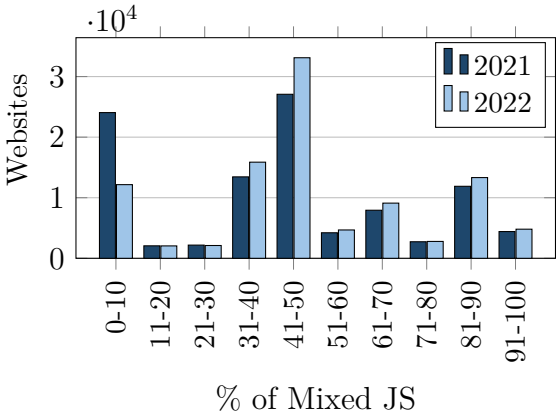


Figure 4.11: Comparison of % mixed JS scripts without any threshold on tracking scores for web corpus in 2021 and 2022.

Takeaway. To maximize tracking prevention while minimizing functional breakage, mixed scripts need to be inspected at a finer granularity.

RQ3: Prevalence of Mixed Scripts

A trivial way for web developers and trackers to bypass filter lists is by mixing functional behavior with tracking in a single script. Privacy-enhancing content blockers, such as uBlock Origin, cannot afford to break the webpage and have no choice but to allow such scripts to load in the browser. To gather concrete evidence on the prevalence of this practice, we first conduct a longitudinal experiment on the frequency of mixed JS scripts over the past two years (2021 and 2022) on 100K webpages. In 2021, we crawled 100K webpages and classified the collected JS code using the SBFL-inspired approach from Section 4.3. We repeat the same experiment in 2022 on the same 100K webpages.

Figure 4.10 shows the result of the experiment. The x-axis represents the percentage of scripts that are mixed, ranging from 0 to 100 in 10 bins each of size 10. The y-axis represents the number of webpages in each bin. In 2021, 15% of webpages out of 100K have between

11% to 20% of scripts that were mixed. This number increases to 18% in 2022. Overall, in 2021, out of 220K JS scripts, 28K are mixed JS scripts, making it 12.8%, whereas, in 2022, 37.5K out of 256K JS scripts are mixed, making it 14.6%. There is 14% increase in the number of websites employing mixed scripts over 100K websites, as compared to last year. For example, on the website `kixie.com`, we observe a new mixed JS script `20564323.js` in 2022, initiating HubSpot analytics code along with the functional code that redirects the `Try Kixie Free` button. We also find that the change in total script count corroborates the general belief that JS scripts across the web have increased marginally since 2021 [188].

While investigating selective JS blocking, we also find deterioration in the functionality when only tracking scripts are blocked (`TS`). Naturally, we ask *why does blocking tracking scripts (TS) result in functional deterioration?* We suspect that such an issue may arise due to the narrow threshold on SBFL's tracking score. JS code units (*i.e.,* scripts, methods) with > 2 score are annotated as purely tracking. Functional behavior in tracking scripts can also exist due to the dynamic nature of webpages. Between the tracking score measurement and blocking experiments, the script may have changed, or the webpage deliberately refactors the script slightly for reasons such as JS obfuscation [156, 176] or minification [151]. For better threshold selection, we must answer *what are the consequences of widening the tracking score threshold?* We conduct a brief sensitivity analysis on the tracking score's threshold. Figure 4.11 shows the new distribution when the threshold is set to maximum. We find that 46% of the webpages have more than 50% of their scripts mixed with at least one tracking or functional request, further reducing the applicability of JS script blocking and showing the extent of this problem. Our investigation in RQ3 highlights the following trade-off. We either sacrifice functionality when blocking mixed JS scripts or let go of privacy. If functional preservation is critical, we forego opportunities to block numerous tracking activities.

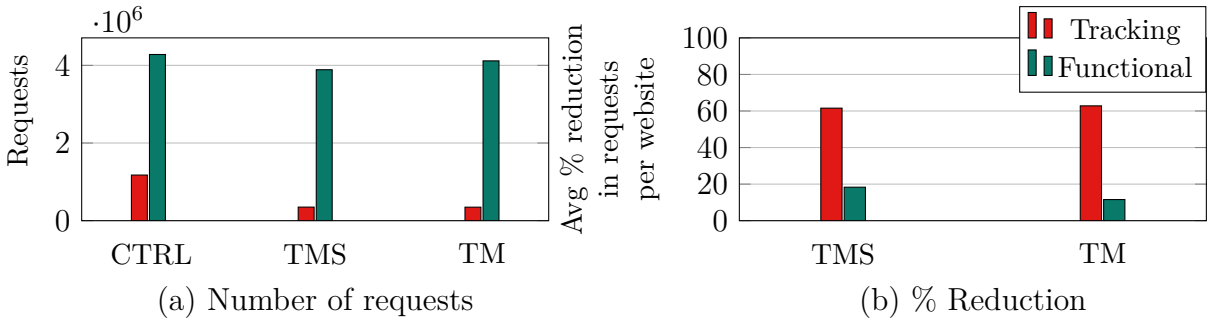
Takeaway. Websites are increasingly employing sophisticated code refactoring techniques (*e.g.,* inlining or bundling) to mix tracking code with functional code, making existing content-blocking techniques ineffective.

RQ4: Fine-Grained JS Blocking

In RQ4, we assess the benefits of performing JS blocking at the method-level. Our hypothesis is that blocking tracking JS method will provide higher precision in tracking prevention, leading to significantly lower functional breakage than JS script-level blocking. In our first experiment, we compare the effectiveness of method-level JS blocking (`TM`) against tracking and mixed JS blocking (`TMS`).

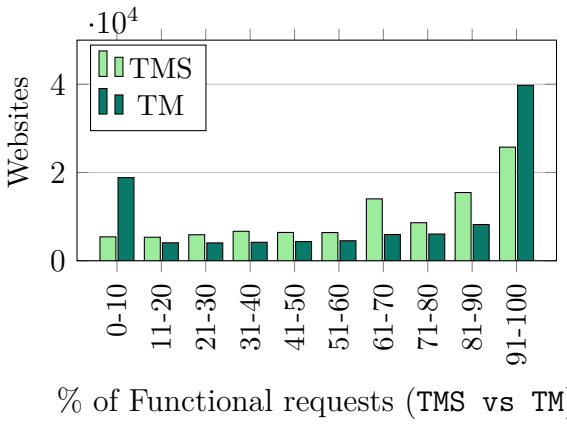
We combine results from blocking both tracking and mixed scripts (`TMS`) as the baseline because all tracking methods are either located in tracking scripts or mixed scripts. Blocking a tracking JS method (`TM`) may eliminate the tracking behavior of a mixed script or a tracking script.

Figure 4.12 summarizes these results. Both baseline tracking and mixed JS blocking (`TMS`) and method-level JS blocking (`TM`) reduce the tracking requests by 71% and block on average 62% of the tracking requests per page. The two configurations cover most of the tracking requests among themselves, and blocking them will yield the same result. More surprisingly, we see an improvement in total functionality retention when blocking method-level (`TM`) *i.e.,* a 6% total improvement, whereas the average functional request breakage per page decreases by 7%. On evaluating HTML, JS method-level blocking(`TM`) retains approximately 2X more functional HTML tag sources, such as images and scripts, than blocking tracking and mixed JS scripts (`TMS`), as shown in Table 4.6. For example, in Figure 4.14, we visually inspect `deeretnanews.com` to find functional media breakage in `TMS` configuration that loads



(a) Number of requests (b) % Reduction

Figure 4.12: (a) compares the request count of control configuration with tracking and mixed (`TMS`) and method-level JS blocking (`TM`). (b) shows average % reduction in request per website for tracking and mixed (`TMS`) and method-level JS blocking (`TM`).



% of Functional requests (`TMS vs TM`)

Figure 4.13: The % of functional requests in tracking and mixed (`TMS`) JS blocking and method-level JS blocking (`TM`). A higher % of functional requests is desirable.

Tag Category	Tracking & Mixed JS Blocked (TMS)	Tracking JS Methods Blocked (TM)
`<image>`	30512	17524
`<video>`	0	2
`<iframe>`	18362	14035
`<script>`	56852	30011
`<source>`	37	35

Table 4.6: Missing HTML tags whose URLs are classified as functional in tracking and mixed (`TMS`) and method-level (`TM`) JS blocking.

normally in `TM` configuration.

We further investigate *how much functional breakage does each webpage face with method-level blocking (`TM`) compared to the baseline* `TMS`*?* Figure 4.13 sheds more light on the functional request count between two blocking granularities. With method-level JS blocking (`TM`), 40% webpages have less than 10% functional breakage (preserved more than 90% functional requests). In comparison, tracking and mixed JS blocking (`TMS`) leads to around 25% of webpages in this category.

We observe two classes of webpages: (1) webpages that decouple functionality and tracking more prominently at the method-level and hence, are less prone to functional breakage, and (2) webpages that tightly integrate tracking code with functional, which is harder to separate even at the method-level and thus results in high functional breakage when such methods are blocked. Further investigation on the number of such mixed methods finds that 6% of 366k JS methods integrate tracking with functional code.

Takeaway. Nearly 40% of the webpages implement functional and tracking code in a modularized fashion. Blocking tracking methods in such webpages shows improved tracking prevention and reduced functional breakage as compared to script-level blocking. The rest of the webpages demand increasing the granularity (*i.e.,* statement-level) or incorporating more sophisticated dynamic analysis.

4.4.2 Phase II: Visual Inspection of JS Blocking and Web Breakage

In Phase II, we perform a qualitative study to validate our quantitative findings with an in-depth visual inspection of sampled websites, as described in Section 4.4. We seek to answer the following research questions:

5. Does our manual inspection validate that method-level JS blocking is more effective than JS blocking?

6. Is method-level JS blocking the most effective in minimizing breakage while preventing tracking?

7. Can webpages withstand the removal of tracking methods?

RQ5: Validating the effectiveness of method-level JS blocking.

Figure 4.15 summarizes the results of investigating true functional breakage on 383 websites, measured according to four established metrics (*i.e.,* navigation, SSO, appearance, and others) and three levels of breakage. The X-axis represents the percentage of websites with functional breakage.

Overall, there is an evident decline in the number of broken websites, for both major and minor breakage, when JS method-level blocking is used instead of tracking and mixed JS blocking. These results validate the findings of quantitative analysis in RQ4. In tracking and mixed JS blocking (`TMS`), 68 websites have minor breakage and 118 websites have major breakage, whereas, in method-level JS blocking, 45 websites have minor breakage, and 29 websites have major breakage. Most of the breakages were observed in additional feature categories, comprising broken widgets (*e.g.,* chatbots and feedback) and malfunctioning home buttons.

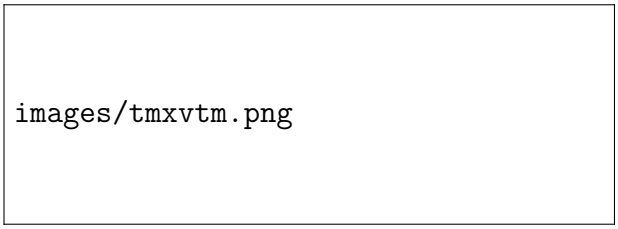


Figure 4.14: Image compares the functional breakage in tracking and mixed JS blocking (right) as compared to method-level JS blocking (left), which loads the website `deeretnanews.com` normally.

`Washingtonpost.com` (ranked 9^{th} in news and media publisher category in USA [66]) is one of the 383 sampled websites. It suffers a crash (a major breakage) in tracking and mixed scripts JS blocking (`TMS`). On the contrary, the website is completely functional and tracking-free at method-level JS blocking (`TM`).

Similarly, on `tenki.jp` (ranked 4^{th} in the streaming and online TV category in Japan [64]), manual inspection reveals a missing Twitter widget and a Twitter button in tracking and

mixed scripts JS blocking (TMS).

These breakages are documented as minor breakages. However, in method-level JS blocking (TM), all tracking advertisements are blocked, and both the button and widget appear correctly and are functional, similar to the control experiment (CTRL). The website ndtv.com (rank 5^{th} in the news and media category in India [134]) renders multiple advertisements in the control experiment (CTRL). Website completely crashes in tracking and mixed scripts JS blocking (TMS), whereas, in method-level JS blocking, it renders normally without any advertisement.

We also argue that minor improvements can make a difference in many websites. For example, website gamestop.com (rank 9^{th} in the gaming category in USA [55]) shows 37.5% breakage in tracking and mixed scripts JS blocking (TMS) whereas shows only 12.5% breakage at method-level JS blocking(TM).

At TMS, we see unexpected white spaces on the top of the website, a minor breakage in the appearance category. The webpage's home button also causes the website to crash, a major breakage recorded in additional functionality. However, in TM, we only see an unexpected white space on the website, a minor breakage in the appearance category. These results also affirm that the breakage metrics (network request and media resources) used in Phase I are effective measures of breakage.

RQ6: Is method-level blocking most effective in reducing breakage and eliminating tracking?

Although method-level JS blocking (TM) performs significantly better than tracking and mixed JS blocking (TMS), there are cases where we observe little or no improvement. This is mainly because of 6% methods still show mixed behavior *i.e.,* include tracking and functional

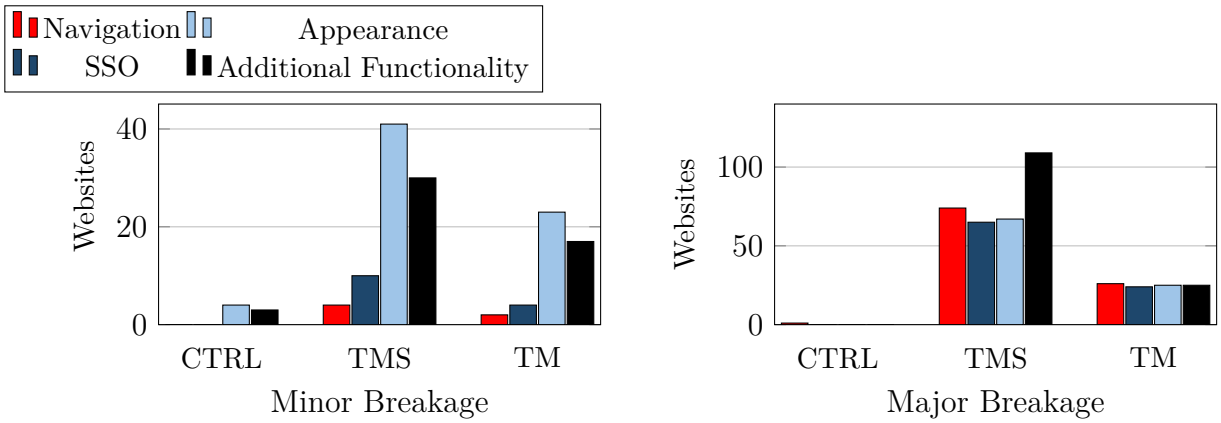


Figure 4.15: (Left) Comparison of "minor" breakage in tracking and mixed JS blocking (`TMS`) vs method-level JS blocking (`TM`) among 383 sampled websites. (Right) Comparison of "major" breakage.

code. `Elpais.com` (currently ranks 2^{nd} in the news and media publisher category in Spain [53]) fails to load a single resource in tracking and mixed scripts JS blocking (`TMS`). However, in method-level JS blocking (`TM`), it causes the navigation bar to be unresponsive, a minor breakage due to the mixed method `e.loadInternal` in script `provider.hlsjs.js`.

RQ7: Can webpages sustain simply removing the tracking JS method?

On 100K webpages, we have found that webpages in their vanilla form have 1.32 severe errors on average. Severe error refers to three main compile-time errors in JavaScript: syntax errors, runtime errors, and logical errors. Errors are common in JS and do not always impact functionality. Compared to other software, webpages can withstand many runtime issues, such as network error, JS script not found, and JS script syntax errors that can arise from diverse host environments. In our experiments, we block JS tracking method by simply renaming the method, which may lead to MethodNotFound error. Replacing a method name and redirecting its invocation may generate additional errors. However, such errors do not affect the website's functionality, as they only terminate the tracking-inducing thread in

the JS process.

4.5 Discussion

In this section, we present the key takeaways of our empirical investigation, highlight the key challenges of effective JS blocking, and offer future ideas for dynamic analysis-based fine-grained JS blocking.

JS blocking at finer granularity. While blocking JS tracking methods is beneficial, we still observe that 5.5% webpages with some levels of tracking activity and functionality breakage. These webpages contain method(s) that (1) implement both tracking and functionality or (2) are used by tracking and functional code for downstream activity (*e.g.,* initiating a network request). We foresee better separation at a finer granularity. In the future, we propose applying dynamic program slicing [75, 132, 195] to separate tracking statements from functional statements. For inseparable code, we propose dynamic invariant detection [110, 139] to construct program variable profiles for tracking and functional behaviors. Program in-variants for tracking can be used as an automated guard to prohibit tracking execution.

Dynamic nature of JS. We find that a number of scripts use dynamic features such as `eval()` and anonymous functions [151, 169]. A number of scripts also employ JS minification and obfuscation techniques that produce code that is uninterpretable manually [156, 176]. Such practices further motivate the use of advanced dynamic program analysis techniques for tracking code identification and removal.

JS dataflow analysis. In this work, we captured the stack trace of a tracking or functional network request and then annotate the script method at the top of the stack. By focusing

on request-initiator code units, we may miss opportunities to trace back to the source of the tracking behavior inside the nested JS codebase. Finding such a location may offer better opportunities to preserve functionality as the request-initiator method or script may simply be a "gateway" for all network requests. In addition to the call stack, we can also leverage the dataflow graph of the JS codebase to perform a richer analysis of a webpage's execution. For example, in Listing 4.3, the stack trace inside the method `B` does not contain the parameter `C`.

Since the method `B` depends on parameter `C`, the identification technique may not understand the entire context when `B()` is called. We recommend capturing such rich execution traces with calling contexts and a complete data flow graph to understand better the flow of information through the nested code and how it influences the execution behavior, tracking, or functional. We anticipate that such traces can help identify better locations (*e.g.,* non request-initiator methods) to alleviate tracking while preserving functionality.

```
1 function TrackingReq() {
2     C = getVal();
3     B(C)}; };
```

Listing 4.3: Call stack does not show complete dataflow.

Performance impact of JS blocking. Although we do not consider performance in our analysis, our focus is to minimize tracking without comprising functionality. Recent works [91, 92] show that the removal of non-critical components of JS code can significantly reduce page load times. Similarly, removing the tracking JS code may reduce the performance overhead along with functionality preservation.

Other future research directions. We plan to conduct an investigation into more meaningful and semantics-aware tracking code identification. Our key observation is that finding a tracking code unit in webpages has striking similarities with fault localization. Even a

simple faulty code localization method such as SBFL showed promising results towards functionality-preserving JS blocking. On the code refactoring front, our observation of 100K vanilla live websites reveals that today's webpages can withstand severe errors. Therefore, we expect that slightly unsafe code refactoring techniques to remove the tracking code may be promising in effectively preserving functionality while preventing tracking.

Future tracking code identification techniques can greatly benefit from recent advances in automated debugging and fault localization [125, 146]. For example, given filter list as a test oracle, we can adapt search-based debugging approaches to perform a systematic search on JS code and precisely isolate the tracking and functional code units [150]. Similarly, the completeness of static code dependency analysis (*e.g.,* reachability analysis) can complement the soundness of dynamic analysis (*e.g.,* call graph) to improve the precision of tracking code localization.

Code clone detection is an active area of research, with many advanced techniques available for traditional software [98]. Given annotated JS code units, code clone detection techniques can identify similar code on webpages to find the presence of tracking code. Once a JS code clone is correctly detected, we can leverage supervised learning [123, 172] to extract valuable features, both semantic and syntactic, for accurate tracking code localization. If such an accurate model is available, a JS blocker can detect tracking JS code units in real time and block them before loading the website.

Similar to training a classification model, one possible direction is to create a taxonomy of tracking code's signature, similar to the ones in malware detection [115, 124, 199], and find a match with a webpage's JS entity at page load. However, page load times are critical in the web domain, refraining from any computationally expensive operation. Using fingerprints to locate tracking code at page load is a lightweight process that can easily be performed at page load time without a noticeable slowdown.

Can publishers also benefit from the results of our JS blocking study? Our study is conducted from the perspective of privacy-enhancing content-blocking tools. If suitable, we suggest publishers adopt an approach such that either the website works reasonably without JS or at least employ a highly decoupled JS architecture that separates tracking and functionality, *i.e.,* separate JS scripts/methods. This architecture will retain functionality effectively when JS code level blocking reduces tracking. On the contrary, publishers who want to retain maximum tracking may leverage the current weakness of JS script-level content blocking by maximizing the overlap between tracking and functional code units.

4.6 Summary

In this chapter, we conduct a large-scale empirical investigation on the impact of different JS Code blocking methodologies on 100K websites, followed by a careful visual inspection of 383 websites to measure website breakage. Our results show that blanket JS blocking prevents tracking but incurs major functionality breakage on approximately two-thirds of the websites. We identify that 15% of the scripts on the web combine tracking and functionality, leading to website breakage if blocked. When we increase the granularity of JS blocking to target tracking methods inside mixed scripts, the functional breakage of websites reduces by 2X while providing the same level of tracking prevention. Our in-depth manual inspection of 383 websites validates that method-level JS blocking reduces major breakage by 3.8×. Through this study, we highlight the promise of fine-grained JS blocking and the subsequent open challenges towards adapting such a technique in practice.

Chapter 5

Blocking Tracking JavaScript at the Function Granularity

5.1 Introduction

Modern websites extensively rely on third-party JavaScript (JS) to implement tracking (*e.g.,* advertising, analytics) and non-tracking (*e.g.,* functional content) [19, 93, 147]. In fact, 81% of all tracking requests are triggered by JS [82]. Privacy-enhancing content blockers aim to block tracking JS without breaking the legitimate website functionality. As the arms race has escalated with trackers attempting to evade blocking, the state-of-the-art content blocking approaches face the following key challenges in blocking tracking JS [27, 95, 121, 170, 174].

First, the state-of-the-art content blocking approaches do not capture the context needed to effectively detect tracking behavior implemented in a script. While existing approaches (*e.g.,* [80, 95, 170]) capture the script that initiates a network request, they do not capture the full stack trace that is needed to understand the sequence of function calls that result in the network request. The lack of sufficient consideration of the execution context associated with tracking behavior fundamentally limits their effectiveness.

Second, the state-of-the-art content blocking approaches are challenged when tracking and non-tracking resources are lumped together. When both tracking and non-tracking resources

are served from the same network location, filter lists [21, 22] or even ML approaches [121, 170] struggle. This issue is further exacerbated by the use of URL encryption [24, 190]. This means that URL-based approaches are unable to discern between tracking and non-tracking resources. Similarly, JS signatures [95] are ineffective when tracking and non-tracking code is combined in the same script. This is fundamentally a granularity issue – *i.e.,* specific functions within mixed scripts are responsible for tracking [80].

Third, the state-of-the-art content blocking approaches rely on the laborious process of manually curated filter lists. Filter lists are used to block tracking network requests [21, 22] and stop the execution of tracking code [7, 61]. TrackerSift [80] relies on manually curated filter lists to detect mixed tracking scripts. uBlock Origin employs manually refactored replacements (aka scriptlets) to handle mixed tracking scripts [34, 65]. The reliance on manual curation fundamentally hinders scalability.

NoT.JS advances the state-of-the-art by addressing these three challenges. First, NoT.JS leverages browser instrumentation to capture dynamic execution context, including the call stack and the calling context of each function call in the call stack. While a JS function's static representation remains unchanged, the execution context around it may alter its semantics. Dynamic execution context enables NoT.JS to semantically reason about a JS function execution, which is essential in differentiating its participation in tracking and non-tracking activity. Second, NoT.JS leverages this dynamic execution context to encode fine-grained JS execution behavior in a rich graph representation that includes individual JS functions within each script. Third, NoT.JS trains a supervised machine learning classifier to detect tracking at the function-level granularity and automatically generate surrogate scripts to specifically block the execution of tracking functions while not impacting execution of non-tracking functions. NoT.JS is the first to fully automate the entire surrogate generation process end-to-end, which are currently painstakingly hand-crafted by experts.

We evaluate the effectiveness, robustness, and usability of NoT.JS on the top 10K websites from the top-million Tranco list [138]. Our evaluation shows that NoT.JS accurately detects tracking JS functions with 94% precision and 98% recall, outperforming the state-of-the-art by up to 40%. NoT.JS's contributions in incorporating dynamic execution context account for 29% improvement in F1-score. Against a number of JS obfuscation techniques, such as control flow flattening, dead code injection, functionality map, and bundling, NoT.JS remains fairly robust – its F1-score decreases by only 4%. NoT.JS's automatically generated surrogate scripts block 84% of the tracking JS function calls without causing any breakage on 92% of the websites.

We deploy NoT.JS to study the tracking functions in mixed scripts, discovering that 62% of the top 10K websites have at least one mixed script. We find that the tracking functions are served in the mixed scripts from more than eight thousand unique domains, including those belonging to tag management services, advertisers, and content delivery networks (CDNs). Notably, among these mixed scripts, a significant 70.6% are third-party scripts actively engaged in tracking activities like cookie ghost writing.

Our key contributions are summarized as follows:

1. We propose NoT.JS, a machine learning-based approach to detect and block tracking at the JS function-level granularity. We show that NoT.JS outperforms the state-of-the-art in terms of accuracy and is robust against evasion.

2. We implement NoT.JS as a browser extension and show that it can be used to automatically generate surrogate scripts by neutralizing tracking function calls. We show that these surrogate scripts can be injected into a website to reliably mitigate tracking at its origin without breaking website functionality.

3. We report to the filter list authors [21, 22] a sample of mixed scripts, detected by

NoT.JS as having both tracking and non-tracking functions, but missed by the filter lists. The filter list authors' review confirms that these are mixed scripts, known to implement tracking, cannot be blocked by filter lists without breaking functionality.

4. We deploy NoT.JS on the top 10K websites to measure the prevalence of tracking functions in the mixed scripts. We show that these mixed scripts are commonly served by tag management services, advertisers, and content delivery networks (CDNs). A majority of these mixed scripts are third-party, actively engaged in tracking activities such as cookie ghostwriting.

5.2 Threat-Model

In this section, we describe the threat model for mixed scripts – JS that combines both tracking and non-tracking functionality, making it challenging for privacy-enhancing content blockers to detect and block them.

Definitions. We use the term `initiator function` to describe a JS function that directly initiates a network request. This function is always at the top of the call stack when we analyze a network request. In Figure 5.1, both `getIdentifier` and `loadImage` are examples of initiator functions. Next, we use the term `gateway function` to describe a specific type of initiator function that only initiates network requests and performs no other task. A gateway function essentially initiates network requests on behalf of other functions. In Figure 5.2, `sendRequest` is an example of a gateway function. Finally, we use the term `"neutralize"` to remove tracking in a JS by replacing a tracking function call with a mock function call.

Below, we describe two specific techniques that trackers use to combine tracking and non-tracking JS code.

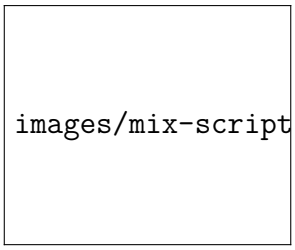


Figure 5.1: Marker ❶ denotes a tracking request sent to tracker.com, which in turn returns a user identifier indicated by ❷. This identifier is stored for monitoring user activity. On the other hand, ❸ marks a non-tracking request dispatched to a CDN, which fetches an image represented by ❹

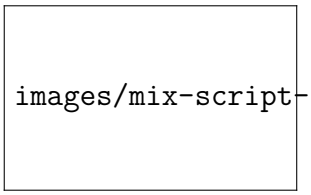


Figure 5.2: Marker ❶ marks a call from `getIdentifier` to `sendRequest`, initiating a tracking request to tracker ❷. The received identifier is captured at ❸ and relayed back to the originating function ❹. Concurrently, ❺ signifies a call from `loadImage` to `sendRequest`, resulting in a non-tracking request ❻ to a CDN. The fetched image is displayed at ❼ and returned to `loadImage` ❽.

Distinct tracking and non-tracking JS functions in the same script. Figure 5.1 illustrates a script containing both tracking and non-tracking JS functions, each with separate roles. In this example, `getIdentifier` gathers user data and sends a tracking request to `tracker.com` ❶, which returns a user identifier ❷. This identifier is used to track user activity on the webpage. The same script also includes functions like `displayContent` and `loadImage`, essential for the webpage's proper functioning. For instance, `loadImage` sends a non-tracking request ❸ to `CDN.com` to load an image ❹. Blocking the script to stop `getIdentifier` would also disable essential functions like `loadImage`, harming the webpage's functionality. Therefore, the ideal approach would neutralize tracking function calls to `getIdentifier` while leaving the rest of the script untouched.

Use of gateway functions to initiate tracking and non-tracking requests. Figure 5.2 illustrates how a gateway function, `sendRequest`, is used to handle both tracking and non-tracking network requests. The single script includes functions for both tracking and non-tracking tasks. Specifically, `getIdentifier` gathers user data for `tracker.com` to obtain a user identifier. Conversely, `loadImage` fetches images for the page from `CDN.com`. In contrast with the previous scenario, these functions do not initiate requests directly and delegate this

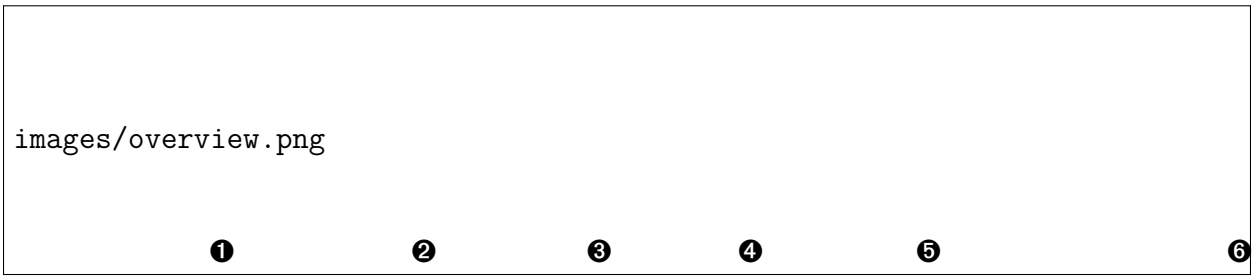


Figure 5.3: NoT.JS pipeline: ❶ Crawl sites, save data; ❷ Create JS function graph; ❸ Extract features and ❹ label it; ❺ Train classifier; ❻ Classify tracking/non-tracking JS functions, and create surrogates.

task to the gateway `sendRequest` function. For instance, `getIdentifier` calls `sendRequest` ❶ to send request to `tracker.com` ❷. In return, `tracker.com` provides a user ID ❸, which is sent back to `getIdentifier` ❹. Similarly, `loadImage` calls `sendRequest` to fetch images from `CDN.com`, which are then displayed on the webpage in ❺ - ❼.

The gateway functions further abstract tracking and non-tracking in JS code and necessitate the inclusion of the complete execution context of a JS function to determine the most effective strategy to block the execution of tracking JS code. Since the tracking function `getIdentifier` is not always at the top of the call stack (Figure 5.2), simply neutralizing the initiator function is not effective. Therefore, an ideal approach needs to incorporate a complete execution call stack and calling context to identify tracking JS functions. In this case, only the function calls to `sendRequest` invoked from `getIdentifier` should be neutralized while maintaining their functionality when `sendRequest` is invoked from `loadImage`.

5.3 Design And Implementation

In this section, we describe the design and implementation details of NoT.JS. Figure 5.3 provides an overview of NoT.JS's pipeline, which starts with automating website crawling using the Selenium web driver and collecting data using a Chrome browser instrumented

with a custom-built extension ❶. Using the collected data, NoT.JS generates a graph representation ❷ that leverages the JS dynamic execution context of a comprehensive list of webpage's activities like network requests, DOM modifications, storage access, and a subset of other Web APIs (listed in Table 5.1), that are commonly used by trackers [85]. Using this graph representation, NoT.JS extracts unique structural and contextual features of tracking activity ❸ and labels it ❹, which are then utilized for training a random forest classifier capable of accurately identifying tracking entities ❺. Finally, the classification results are utilized to generate surrogate scripts that neutralize tracking function calls which can be replaced at runtime by content blockers ❻.

5.3.1 NoT.JS's Chrome Instrumentation

NoT.JS first collects the training data to train a fine-grained, high-accuracy classifier. It automates the website crawling and data collection process using selenium [62] and a custom-built Chrome extension. We choose Chrome extension interface [49] to capture web activities due to its ease of use and lightweight nature compared to instrumenting the JS engine, such as V8 in Chrome [44]. NoT.JS's Chrome extension captures the JS dynamic execution context for each activity on the webpage, along with other relevant meta-data such as network requests and response payloads. NoT.JS's JS dynamic execution context for each web activity comprises of:

- The **call stack** [1] outlines the sequence of function calls, including details like the script URL, method name, and line and column numbers where each function is invoked.
- The **scope chain** [2] for each function call within the stack that includes the number of arguments and local and global variables.

[1] https://chromedevtools.github.io/devtools-protocol/tot/Runtime/#type-CallFrame
[2] https://chromedevtools.github.io/devtools-protocol/tot/Debugger/#type-Scope

Activity	Property
Network Request	Network.requestWillBeSent Network.responseReceived Network.requestWillBeSentExtraInfo Network.responseReceivedExtraInfo
DOM Modifications	DOM.attributeModified DOM.childNodeInserted DOM.childNodeRemoved
Storage Access	Document.cookie (get/set) Storage.setItem Storage.getItem
Web APIs	Navigator.sendBeacon Navigator.geolocation Navigator.userAgent
	BatteryManager.chargingTime BatteryManager.dischargingTime
	MouseEvent.movementX MouseEvent.movementY
	Element.copy Element.paste
	Document.visibilitychange
	Touch.force

Table 5.1: List of webpage's activities captured by NoT.JS.

The intuition behind capturing dynamic execution context is to gain a deeper understanding of the web activity. Figure 5.4 illustrates the two primary components of the Chrome extension, namely the background script and content script, which work together to facilitate data collection.

Background script. The background script [45] is an essential component of the Chrome extension that captures webpage activity using the Chrome DevTools Protocol APIs [50]. Specifically, the network API monitors traffic and provides valuable information about HTTP requests and responses, including headers, bodies, call stack, scope chain, and timestamps. Additionally, the DOM API captures changes in the Document Object Model and provides read and write operations along with the call stack, and the scope chain. However, the DevToolProtocol does not expose all Web APIs, cookies, or storage APIs, which are

available through the content script as discussed next.

Content script. The content script [51] runs within the context of the webpage and can interact with its functionality. It is responsible for collecting the JS dynamic execution context, *i.e.,* call stack and scope chain for the webpage's activity, and exposing Web APIs, cookies, and storage APIs that are not available in the background script, by overriding functions. Listing 5.1 shows the snippet from `content.js` that overrides the `sendBeacon` function of the `Navigator` object. The overridden function collects two types of information: the call stack of the JS execution and the scope chain of each function call in the call stack. The call stack is collected using the `console.trace()` [28] function, which logs the stack trace. The scope chain is collected using the Debugger API [29], specifically via the `Debugger.paused` event that provides the `callFrames.scopeChain` parameter.

```
1 // storing the original sendBeacon function
2 const sendBeac = Navigator.prototype.sendBeacon;
3 // overriding sendBeacon function
4 Navigator.prototype.sendBeacon = function(url, data) {
5     // collect stack trace and scope chain
6     console.trace();
7     Debugger.paused.callFrames.scopeChain;
8     // call back the original function
9     senBeac();}
```

Listing 5.1: Overriding `sendBeacon` function using content script.

Figure 5.4 illustrates the sequence of the data collection process with NoT.JS's chrome extension. First, `content script.js` sets up communication with `background.js`, as shown in step ❶. On a network activity or DOM modifications (❷), `background.js` triggers a message for `content.js` to capture the corresponding JS execution call stack and scope chain, as shown in ❸. Finally, the collected data is sent to `background.js` for storage, as shown in

❹ and ❺. Additionally, the `content.js` uses the same communication channel to log cookies, storage, and APIs data on a storage.

Data collection. We conduct an automated crawl of the landing pages of the top 10K websites in the Tranco top-million list [138] as of July 2023, using Selenium with Chrome 114.0.5735.133 and a purposely-built extension. We perform this crawl from the United States. On average, it took approximately 10 seconds for a webpage to fully load (until the onLoad event is fired) [60], and an additional 30 seconds before moving on to the next website. The crawling process is stateless, as we cleared all cookies and local browser states between consecutive crawls. This helps to ensure that the collected data is more reproducible and accurately reflects the current state of the webpage without being biased by previous webpages visits.

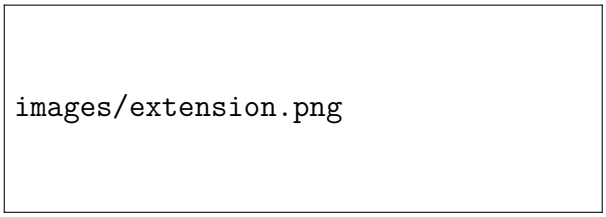


Figure 5.4: An illustration of NoT.JS's chrome-extension, showing the communication sequence between background and content scripts.

5.3.2 Graph Representation

NoT.JS constructs a graph representation from the collected data of each webpage. NoT.JS's graph representation leverages JS function-level features and JS execution context of each of the webpage's activity, as listed in the Table 5.1. The graph's unique structure offers several advantages over traditional techniques. NoT.JS allows for semantic reasoning with its dynamic execution context and enables traceability by providing a complete history of how a particular webpages activity is executed. This information is essential in differentiating the intention of the same graph node (*e.g.,* JS function) when participating in different execution scenarios. While their static representation remains unchanged, the JS execution context around them at runtime may alter their semantics. By building a graph representation

first, NoT.JS facilitates the extraction of structural and contextually rich features, which it utilizes to identify privacy-invasive JS functions that are otherwise tightly interleaved with non-tracking code. NoT.JS's graph representation is the first-of-its-kind [121, 170, 174] to incorporate the JS execution call stack and calling contexts (scope chain) fully.

Nodes. NoT.JS categorizes a webpage's activity into five types of nodes: JS functions, DOM elements, network, storage, and web APIs. The JS function node represents a function call that has attributes, including its parent script's URL, name (except for anonymous functions), scope chain, line, and column number. The scope chain refers to the variables, functions, objects, and closures that a JS function can access at the time of invocation. Closures are a special type of function that can access variables in its enclosing function's scope chain, even after the function has returned. One function can generate multiple nodes depending on its calling sequence (call stack) and context (scope chain).

This is particularly useful for gateway functions that participate in both tracking and non-tracking activity. For instance, in Figure 5.2, NoT.JS creates two separate nodes for the `sendRequest` function: one for its invocation within `getIdentifier` and another for `loadImage`. The DOM element node has attributes such as element name, class, or id and a value if available. The network node represents all the network requests that are sent by a webpage. The storage node represents all client-side data storing mechanisms, with attributes that differentiate be-

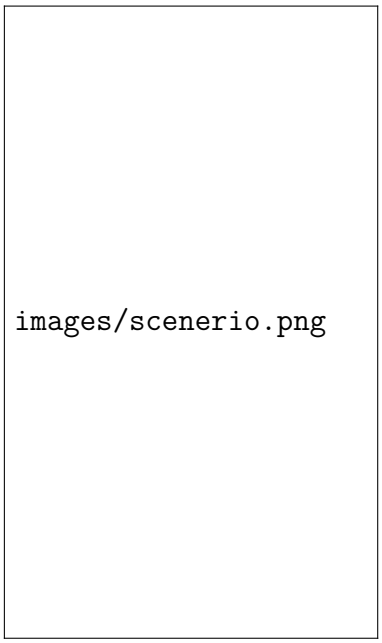


Figure 5.5: Sequence diagram: `getMouseMove()` collects data, calls `updateCookie(data)` to modify cookie, triggers `sendReq(data)` to send the request.

tween mechanisms such as cookies [30] and
local storage [35]. The web API node type represents Web APIs, as listed in Table 5.1, with attributes that differentiate between APIs, such as charging time and discharging time.

NoT.JS's graph representation is the first to include function and web API node types.

Edges. Edges in the graph generated by NoT.JS depict runtime dependencies between nodes. We extract two types of edges: call and behavioral edges. Call edges represent the sequence of function calls in a JS execution call stack. These edges connect function nodes to other function nodes to represent the dynamic caller-callee relationship. Whenever a function is called, a directed edge is created from the caller function node to the callee function node. These edges add valuable information about the sequence of function calls, which existing graph representations do not capture [121, 170]. Such edges enhance NoT.JS's capabilities in modeling the semantics of the JS script by making its graph context-aware. On the other hand, behavioral edges are used to represent interactions between the JS functions, DOM, network, storage, and API nodes. If a function initiates a network request, a behavioral edge is created from the corresponding JS function node to the network node. Similarly, if a function triggers a storage or Web API call, a behavioral edge is created between the JS function node and the corresponding storage or Web API node, respectively. The direction of the edge is dependent on the context of the API call. For instance, if the JS function is storing data, there is an edge from the JS function to the storage or API node, and vice versa.

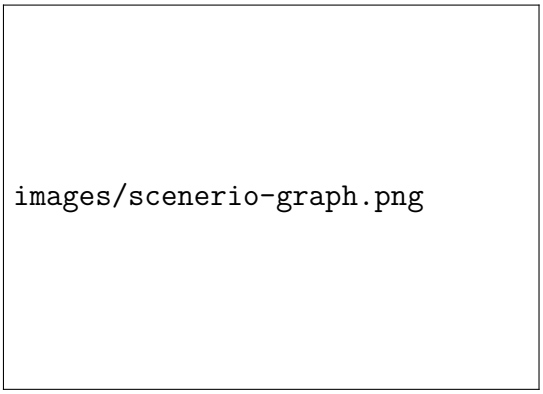


Figure 5.6: A simplified NoT.JS's graph on mouse movement tracking.

Graph construction. Figure 5.5 illustrates how NoT.JS constructs a graph representation

Features	`send Req()`	`update Cookie()`	`getMouse Move()`
Number of requests sent	1	1	1
Is gateway function	1	0	0
Number of cookie (setter)	0	1	0
Number of web API (getter)	0	0	1
Number of arguments	1	1	0
Number of callee functions	1	1	0
Number of caller functions	0	1	1
Ascendant has cookie accesses	1	0	0
Descendant has cookie accesses	0	0	1
Ascendant has web API accesses	1	1	0

Table 5.2: A small subset of features extracted from Figure 5.6.

for a webpage that tracks user movement on an HTML element, stores it in a cookie, and sends data to an external server. Each box in the illustration represents an entry of a JS execution call stack. The `getMouseMove` function is responsible for capturing the movement of the mouse pointer on the specified HTML element. This is achieved through the `MouseEvent` API, which provides access to the `movementX` property. This property represents the difference in the X coordinate of the mouse pointer between the current event and the previous mouse move event. Once the data is collected, it is passed on to the `updateCookie` function, which updates the cookie value using the `document.cookie` property. The data is then passed to the `sendReq` function, which sends an HTTP request to an external server with the data.

Figure 5.6 shows a NoT.JS's graph representation. The graph begins with a behavioral edge (dotted-line) that obtains mouse movement data associated with the specific HTML element, represented in ❶, and ❷. The call edge (solid-line) ❸ between `getMouseMove` and `updateCookie` represents a function call that also passes the mouse movement data (scope). The behavioral edge in ❹ represents the `document.cookie` call to update the storage node. The call edge ❺ between `updateCookie` and `sendReq` represents a function call that also passes the mouse movement data (scope). Finally, the behavioral edge ❻ between the function and the network node represents the HTTP request sent to an external server.

5.3.3 Feature Extraction

Once a fine-grained graph is generated for all observed web activities on a webpage, NoT.JS extracts two kinds of features from the graph to augment the current node's attributes: structural and contextual-based features.

Structural features. Structural features represent relationships between nodes in the graph, such as ancestry information and connectivity. For example, how many Web API nodes are present in the ancestor chain of a function node or whether a function directly or indirectly interacts with a storage node? Adding JS functions from the execution call stack improves the completeness of the structural features by providing additional context under which a function is called and filling in the missing pieces about the sequence of events prior to reaching the current node.

Contextual features. NoT.JS includes JS dynamic execution context in the generated graph using contextual-based features. We extract three types of contextual features. First, we count the number of local variables, global variables, and closure variables. Second, we count the number of arguments passed to the function. Third, we include the number of network requests, DOM modifications, and API (*e.g.,* cookies, storage APIs, etc.) calls made by the function. These features play a vital role in understanding JS function behavior, especially during the execution of tracking activity. A function's calling context has been used previously to construct the dynamic invariants of a program, which helps in verifying a given program behavior [149, 157, 194]. Such invariants are often implemented as assertions. Similarly, NoT.JS's contextual features can help the downstream training process learn invariants about JS function under which the JS function participates in tracking and the invariants under which it participates in non-tracking activity. Our graph representation is unique in its ability to extract contextual-based features that cannot be obtained by prior

approaches [121, 170, 174].

Table 5.2 presents a subset of the features, including structural and contextual attributes, obtained from the graph representation of NoT.JS shown in Figure 5.6. The `number of requests sent` feature is triggered for all functions, as the function calls appear in the call stack when tracking requests are initiated, with the `sendReq` function acting as the gateway. Contextual features encompass the number of storage (getter and setter) and web API (getter and setter) operations, as well as function arguments, providing insights into each function's contextual behavior. Furthermore, the inclusion of structural attributes, namely the ascendant and descendant relationships within the storage and web API nodes, enriches our understanding of hierarchical structures and code dependencies.

5.3.4 Labeling

We adopt a prior approach [82] to label NoT.JS's graph representation. We label network requests as either tracking or non-tracking based on whether their URL matches the rules in EasyList [21] and EasyPrivacy [22], which are widely employed by content blockers to identify tracking activity. Next, we analyze the call stack of the requests and label a JS function as tracking if it exclusively participates in the stack trace of tracking requests. Otherwise, if a JS function participates in non-tracking requests (or a combination of tracking and non-tracking requests), we label it as a non-tracking JS function. This approach establishes a conservative ground truth where the functions with mixed behavior are labeled as non-tracking. However, these mixed functions comprise only 3.9% of our ground truth, , as discussed in more detail in Section **??**. In addition, we exclude functions that trigger web, storage, or cookie API calls but are not present in either a tracking or non-tracking request call stack. This is because we lack evidence from filter lists to label them.

5.3.5 Classification

We use a random forest classifier, which is an ensemble learning method. It constructs multiple decision trees and combines their predictions to obtain a final prediction. Prior work [121, 170] also opt to use a random forest classifier since it outperformed other comparable models. To assess our classifier's performance, we divide our dataset into training, validation, and testing sets. Specifically, we use 60% of the dataset for training the classifier, 20% for evaluating its accuracy during hyperparameter tuning, and the remaining 20% for final evaluation of the model. To optimize hyperparameters, we use the validation set to configure the depth of each decision tree in the forest to 20 and the number of trees used in the forest to 1000. We partition our data to avoid any overlap between the JS functions used across training/validation and testing.

Data-split	**Tracking Functions**	**Non-tracking Functions**	**Total Functions**
Training	408,429	674,724	1,083,153
Validation	135,590	225,462	361,052
Testing	136,045	225,007	361,052

Table 5.3: The breakdown of the data employed to train, test, and validate the NoT.JS classifier.

5.3.6 Surrogate Generation and Replacement

Our classifier categorizes JS functions into tracking or non-tracking. For tracking JS, NoT.JS creates a replacement JS called a surrogate to replace the original JS in future page loads. NoT.JS's surrogate generation and replacement is website-specific, which helps address the variations in obfuscation and minification techniques deployed by different websites. Our approach for surrogate generation involves neutralizing tracking function calls identified by

the classifier by substituting them with a mock function call within the script. Once surrogate scripts are generated for a specific website using this approach, we create a filter rule that enables the substitution of the original script with the surrogate script at runtime.

Surrogate generation. Our surrogate script generation process relies on three key elements: (1) The classification labels assigned by NoT.JS. (2) The script source as it appeared when the response was received. (3) The line and column numbers corresponding to the function call at runtime.

```
1 function x() {
2     var e = [];
3     ...,
4     t.__satelliteLoadedCallback((function() {
5         var n, a, o;
6         for (n = 0, a = e.length; n < a; n++) o = e[n],
7         t._satellite.-track(o[0], o[1])
8         t._satellite.+mockTrack()
9     })), _satellite.track("pageload")}
```

Listing 5.2: The example demonstrating the neutralization of tracking function calls during surrogate generation.

Leveraging this information, we neutralize tracking function calls by substituting them with a mock call designed to consistently return an empty response. It is important to note that surrogate generation is an offline step. To illustrate this process, consider the function call `track(o[0], o[1])` on `adobe.com` in Listing 5.2 that operates in the context of Adobe Analytics and is classified as tracking by NoT.JS. NoT.JS records the exact line (line 7) and column where this function call begins. The column corresponds to the first element, `"t"` in this example. In the first step, we verify that the function call exists at the recorded line and column numbers based on its function name while skipping this step for anonymous

functions. We replace the original function call with the mock call that returns an empty response. As we discuss later in Section 5.4.5, this simple approach is able to effectively neutralize a vast majority of tracking function calls while upholding the structural integrity of the script.

Surrogate replacement. NoT.JS replaces the original mixed scripts with the generated surrogates at runtime during future page loads. NoT.JS first identifies the target scripts on a webpage using regular expressions (regex) of generated surrogate scripts. For example, we can create a regex rule to identify scripts associated with a domain like `adobe.com/*analytics.js`. Once identified, NoT.JS replaces the target script with the corresponding surrogate. This replacement mechanism is supported in different Chrome extension environments like manifest versions 2 (V2) and 3 (V3). In manifest V2, Chrome extensions employ the Fetch API [32], a conventional method for intercepting and modifying responses at runtime. When a network request is made, the NoT.JS's browser extension intercepts it, verifies that the response status is `OK` (status code 200), and subsequently modifies the response content. This approach allows us to replace a mixed script with the corresponding surrogate script. It is worth noting that this approach is similar to scriptlet replacement in content blocking tools such as uBlock Origin [65] and AdGuard [7]. Manifest V3 introduces the Declarative Net Request API [38], which presents distinct capabilities for response modification. This API does not provide direct response modification capabilities like V2. In the V3 environment, the original request is blocked and redirected to an alternate URL, effectively replacing the response content with the content retrieved from the redirected URL [36]. This allows successful replacement of surrogate scripts in a V3 extension environment.

5.4 Evaluation

This section evaluates NoT.JS's accuracy, feature contributions, robustness to JS obfuscation and enhanced code coverage, and compares it to existing countermeasures. It also evaluates its automated surrogate generation, replacement, and user-centric manual breakage inspection.

5.4.1 Accuracy Analysis

We train NoT.JS's random forest classifier on approximately 1.1 million JS functions in the training set, do hyper-parameter tuning on 361 thousand JS functions in the validation set, and then evaluate its accuracy on 361 thousand JS functions in the testing set. Table 5.3 presents a breakdown of the data split between tracking and non-tracking functions. NoT.JS is able to achieve a precision of 94.3% and a recall of 98.0%. The overall F1-score for NoT.JS is 96.2%, indicating its effectiveness in accurately distinguishing between tracking and non-tracking JS functions.

Model	**Section**	**Precision**	**Recall**	**F1 Score**
NoT.JS	Standard - 5.1	94.3%	98.0%	96.2%
NoT.JS	Obfuscation - 5.3	93.5%	90.4%	91.9%
NoT.JS	Coverage - 5.3	88.4%	95.7%	91.9%
WebGraph	Comparison - 5.4	49.3%	66.4%	56.5%

Table 5.4: Results showcase NoT.JS's precision, recall, and F1-score under standard setting, enhanced code coverage, post-JS obfuscation robustness, and comparison with WebGraph.

Error analysis. While NoT.JS has a relatively low false positive rate (3.5%) and false negative rate (1.9%), we investigate the reasons and contexts of its errors below. We conduct a

manual evaluation by randomly sampling 50 instances where NoT.JS incorrectly identifies JS functions as tracking, despite our ground-truth data missing them. The causes for these errors can be categorized into three primary categories: The first category includes functions in mixed scripts that cannot be blocked by filter lists to prevent website functionality breakage. For example, the function `t` in `js.cookie.min.js` on `kakaku.com` sets the tracking cookie and is classified as tracking by NoT.JS, while its other functions primarily serve the history feature on the website. Consequently, being a mixed script, it cannot be blocked by filter list authors (acknowledged in this GitHub issue [71]), yet it can be handled by NoT.JS. The second category includes functions that are actually involved in tracking but missed by filter lists [78]. For example, the function `b` in the script `htlbid-advertising.min.js` on `wkrn.com` manages ad slots and their configurations. We conduct an in-depth manual evaluation of the entire script, discovering that most of its functions are classified as tracking by NoT.JS. Following this, we report this issue [70] to filter list authors, leading to its inclusion in the filter rules. In total, NoT.JS identifies ten such cases out of a sample of fifty in the aforementioned two categories, that are either missed or cannot be handled by EasyList/EasyPrivacy. We report these cases to filter list authors, leading to four [69, 70, 71, 72] being recognized and six still pending review, highlighting NoT.JS's superior detection capabilities as compared to EasyList/EasyPrivacy. The last category includes functions, where forty out of fifty instances represent an actual error by NoT.JS, primarily due to reliance on features of dynamic execution context. We use the number of arguments, local and global variables in the scope chain, rather than examining the types or values passed to these variables. Therefore, functions with identical dynamic execution contexts—*i.e.,* the same call stack and scope chain—for both tracking and non-tracking activities are labeled as non-tracking in the ground truth but are classified as tracking by NoT.JS. These instances are not technically misclassified by NoT.JS, but rather point to the limitations intrinsic to dynamic execution context. Such multi-purpose functions can serve

both tracking and non-tracking roles. For instance, in Listing 5.3, `_setField` function in `visitor*.js` script served by `microsoft.com`. If parameter `f` is set to false, the function participates in the tracking activity on the website. However, if parameter `f` is set to true, the function participates in the non-tracking activity on the website. In ground truth, this function is labeled as non-tracking because of the same number of arguments. A deeper examination of parameter types and values could address these issues in the ground truth and the classification model trained on it in the future. In summary, 20% of the tracking functions in the sample represent errors in the filter list, while the remaining 80% constitute actual errors by NoT.JS.

```
1 a._setField = function(b, d, f) {
2         null == a._fields && (a._fields = {});
3         a._fields[b] = d;
4         f || a._writeVisitor()};
```

Listing 5.3: `_setField` function from the `Microsoft` script domain

False negatives predominantly stem from code coverage, as not all properties of tracking functions are captured. For instance, a more comprehensive list of Web APIs could be employed to capture additional characteristics, which would also assist in more precise profiling of tracking functions and, consequently, fewer false negatives. More comprehensive crawling can help address these issues, but it may lead to more noise and higher graph construction costs.

5.4.2 Feature Analysis

We study the most influential features in the classification of JS functions by NoT.JS using the concept of information gain. The number of storage accesses is one of the top 10 features in distinguishing tracking functions from non-tracking. Previous research recognizes that

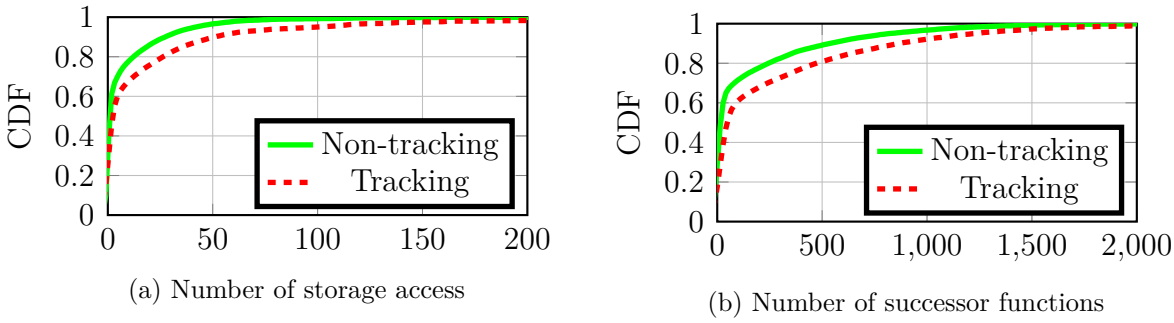
Initiator Functions	Non-initiator Functions	Context	Precision	Recall	F1 Score
✔	✗	✗	68.2%	65.1%	66.9%
✔	✔	✗	55.8%	99.7%	71.5%
✔	✔	✔	94.3%	98.0%	96.2%

Table 5.5: Ablation analysis results of NoT.JS's features in terms of precision, recall, and F1-score.

storage APIs are commonly employed by trackers [170]. Figure 5.7a shows that tracking JS functions have a higher frequency of storage accesses than non-tracking functions. The average storage access is 15 in tracking functions and 8 in non-tracking functions. Another top-10 is the number of successor functions, which offers insights into the calling context of the function. Successor functions show how connected a function is to other parts of the code. Figure 5.7b shows that tracking JS functions tend to have more successor functions than non-tracking functions. The average number of successor functions is 252 for tracking functions and 148 for non-tracking functions. The higher number of successor functions in tracking activities indicates more complex behavior, likely designed to gather, process, and transmit personal data.

Ablation analysis.

Next, we evaluate the impact of various graph configurations and features. We summarize our findings in Table 5.5. First, we find that solely incorporating initiator functions in the graph while excluding non-initiator functions lowers the F1-score of NoT.JS by 29.3%. This reduction is primarily due to the limitation that focusing solely on initiator functions omits the broader context essential for distinguishing between tracking and non-tracking activities. Next, we include both initiator and non-initiator functions but exclude the execution context features. Under this configuration, although precision decreases, there is a notable increase in the recall. This shows that the model now possesses a broader code coverage — it is identifying a greater number of tracking functions, thereby elevating recall. However,



(a) Number of storage access

(b) Number of successor functions

Figure 5.7: CDF highlights NoT.JS features: (a) storage accesses in tracking vs non-tracking functions, (b) successor functions in tracking vs non-tracking functions.

this leads to an increase in false positives. We observe that a call to the same function in tracking and non-tracking contexts has the same representation in the graph; thus, the extracted features are not distinctive enough to accurately segregate tracking functions from non-tracking ones. Finally, we include all functions and contextual features. This setting surpasses the performance of all prior configurations owing to the enhanced code coverage and execution context. The integration of calling context in NoT.JS results in the best precision and overall F1 score. This underscores the pivotal role of JS dynamic execution context with both initiator and non-initiator functions, emphasizing its importance in precisely identifying tracking JS functions.

5.4.3 Robustness

We evaluate the robustness of NoT.JS in detecting tracking functions amid manipulation attempts such as JS code obfuscation and code modifications, as well as in the context of enhanced code coverage. **Code obfuscation.** JS obfuscation is often employed to conceal the meaning of the code, making it harder to decipher for those who may try to reverse engineer or modify it [99, 196]. From our dataset of 10K websites, we randomly selected 10%, which includes 15,939 unique scripts. We then obfuscate these scripts using the `obfuscater.io`

[58] using the following configuration:

```
compact: true,
controlFlowFlattening: true,
controlFlowFlatteningThreshold: 1,
deadCodeInjection: true,
deadCodeInjectionThreshold: 1,
disableConsoleOutput: true,
identifierNamesGenerator: 'hexadecimal',
rotateStringArray: true,
selfDefending: true,
stringArray: true,
stringArrayThreshold: 1,
transformObjectKeys: true
```

Listing 5.4: Configuration used to obfuscate the scripts using obfuscator.io [58].

JS obfuscation poses several challenges for NoT.JS. First, it modifies the names of JS functions using hexadecimal notation, making it difficult for NoT.JS to identify and track the execution of these functions accurately. Second, it alters the JS execution call stack through the use of multiple techniques, such as control flow flattening [86, 191] and self-defending [165, 166]. Control flow flattening involves breaking down JS functions into smaller basic blocks and then rearranging these blocks to change the order of execution. This can make it difficult for NoT.JS to understand the flow of the code and track the execution of different functions. Self-defending adds protection code to the program that can detect if it is being debugged or modified and attempt to avoid execution in these cases. The protection code can prevent NoT.JS from gathering the necessary information and changing the execution call stack of JS activity.

Table 5.4 presents the classification results of NoT.JS on the obfuscated data using the

aforementioned configuration. Overall, obfuscating the JS code reduces the precision by only 0.8% and recall by 7.6%, leading to a 4.3% reduction in the overall F1 score. The marginal drop in recall can be attributed to the model's increased likelihood of missing additional tracking functions. This can be traced back to the non-operational "garbage" functions in the stack, which the model finds challenging to accurately identify. Given NoT.JS's reliance on the dynamic execution context over static features, its precision remains unaffected by obfuscation. However, to further enhance NoT.JS's accuracy on obfuscated scripts, future work could incorporate adversarial training by adding obfuscated scripts in the training set [104].

Enhanced code coverage. To assess the impact of enhanced code coverage on NoT.JS's performance, we conduct a more exhaustive crawl on a randomly selected subset, making up 10% of our original 10K website corpus. During this analysis, in addition to assessing the landing page, we explored five distinct internal pages of each website. We implement bot mitigation strategies that include simulating mouse movements at five unique offsets using the `move_by_offset(x, y)` function. Furthermore, we incrementally scroll through the webpage using the `window.scrollBy()` function. Upon completing the website crawl, we utilize the NoT.JS to classify the functions employed during the coverage analysis. As summarized in Table 5.4, our results indicate a 5.9% decline in precision, which led to a 4.2% reduction in the F1-score. Consequently, the data reveals only a minor decrease in both precision and the overall F1-score for NoT.JS, suggesting that it maintains a reliable level of accuracy even with expanded coverage.

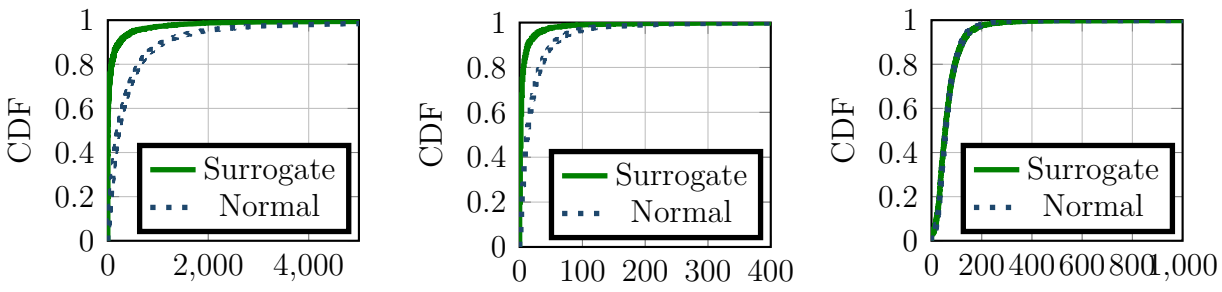
5.4.4 Comparison with Existing Countermeasures

We compare NoT.JS with the state-of-the-art baseline, WebGraph, that classifies tracking script URLs. We evaluate the performance of each. WebGraph predicts the JS script URLs, presuming all functions in the script will have the script label, whereas NoT.JS uniquely classifies JS functions. We measure the precision and recall, and later in the section 5.4.5, we assess the impact on website functionality, with a focus on website breakage

Accuracy analysis. Table 5.4 provides a summary of the results for NoT.JS and WebGraph. NoT.JS outperforms WebGraph in classifying tracking JS functions with 45.0% higher precision, 31.6% higher recall, and an overall 39.7% better F1-score. WebGraph is unable to fully capture the communication edges between the scripts present in the call stack during a tracking activity. As a result, the dynamic execution context surrounding data transfer and the initiator script, which sends the data to the server, is not captured and modeled. Furthermore, tracking at the script-level granularity obscures important features that are associated with fine-grained JS functions, making it difficult to distinguish tracking and non-tracking activity in mixed scripts [82]. Although WebGraph claims a 92% accuracy rate in its paper, this is largely because it is tailored to identify tracking script URLs. Its limitations become evident when dealing with mixed scripts, underscoring the need for function-level granularity.

5.4.5 Surrogate Generation and Replacement

We assess the feasibility of NoT.JS's real-time deployment through the evaluation of its surrogate generation and replacement strategy. For our evaluation, surrogates are generated for a randomly selected 50% of the 10K websites, after which we re-visit each site, substituting the original script with the surrogate prepared by NoT.JS. We utilize the surrogate



(a) Number of tracking functions (b) Number of tracking requests

(c) Number of non-tracking requests

Figure 5.8: CDF illustrates pre- and post-surrogate replacement metrics for 50% of websites: (a) tracking functions, (b) tracking requests, (c) non-tracking requests.

replacement strategy as outlined in Section 5.3, implemented as a Chrome extension.

Surrogate generation. As elaborated in Section 5.3, NoT.JS locates the tracking function call within the script source code and substitutes it with a mock function call, effectively neutralizing it. During the surrogate generation, we calculate four parameters to assess the efficacy of the technique. First, we compute the average number of tracking function calls per webpage that can be successfully neutralized, which is 122.3. Second, we determine the average number of tracking function calls per webpage that cannot be neutralized due to the limitations of our instrumentation, which fails to capture the script source code. The average number of such tracking functions per webpage is 3.1. Third, we calculate the average number of tracking function calls per webpage that cannot be neutralized as the script is inlined—*i.e.,* embedded within the HTML of the page. The average number of such tracking functions per webpage is 4.7. Finally, we measure the average number of tracking function calls per webpage that cannot be neutralized owing to the dynamism in the JS [192, 193], rendering us unable to confirm the line number and column number at run-time. The average number of such tracking functions per webpage is 14.8. In summary, our surrogate generation technique successfully neutralizes an average of 84.4% of classified tracking functions per webpage.

Surrogate replacement. After generating the surrogates, we evaluate both privacy and usability metrics across 50% of these websites, both before and after surrogate deployment. We calculate the average number of tracking functions, as well as tracking and non-tracking requests per website. Furthermore, we carry out a performance and user-centered manual breakage analysis to verify the ongoing usability of websites after surrogate deployment. Originally, the average number of tracking functions per website is 153.6, which drops to 38.5 after deploying the surrogates, representing an 80.1% reduction per website. Figure 5.8a displays the CDF that shows a significant decrease in tracking functions after surrogate deployment. Moreover, the average count of tracking requests per website is initially 28.0. This decreases to 8.4 post-deployment, representing a 76.9% reduction per website. Figure 5.8b displays the CDF that shows a significant decrease in tracking requests on the majority of websites following surrogate deployment. Finally, the average number of non-tracking requests per website is 74.3 and drops minimally to 71.5 after deploying the surrogates. Figure 5.8c displays the CDF, illustrating the number of non-tracking requests against the website distribution. The overlapping lines in the graph indicate negligible impact on website usability.

Performance analysis. We evaluate the performance overhead across 50% of these websites, both before and after surrogate deployment, utilizing Selenium [62] to extract standard page performance metrics for each visit. These metrics include JS memory usage and the timing of key page load events. Table 5.6 summarizes the results. Regarding JS memory usage, we observe a decrease of 32% in total and 25% in used JS heap memory. The total JS heap size represents the entire memory allocation for JS execution, while the used portion reflects the memory actively utilized by JS on the webpage. This decrease is primarily due to the neutralization of tracking function calls, which return an empty response instead of executing the original memory-consuming function. Furthermore, key events marking various stages

Metrics	Normal (Mean, Median)	Surrogate (Mean, Median)
JavaScript Memory Usage		
Heap Total Size	18.79 *MB*, 13.92 *MB*	12.86 *MB*, 9.18 *MB*
Heap Used Size	12.95 *MB*, 10.67 *MB*	9.74 *MB*, 7.39 *MB*
Performance Event Timing		
DOM Content Loaded	1,402 *ms*, 1,136 *ms*	1,290 *ms*, 1,112 *ms*
DOM Interactive	1,067 *ms*, 931 *ms*	1,162 *ms*, 1,005 *ms*
Load Event	2,641 *ms*, 1,888 *ms*	2,562 *ms*, 1,804 *ms*

Table 5.6: Performance analysis for the post-surrogate replacement.

in the browser's page load and rendering process, crucial benchmarks for user-perceived web page performance [43], are completed on average 32 milliseconds earlier. These improvements are seen across several metrics: DOM content load time, which indicates the time from page load start to complete HTML parsing and initial interactivity. DOM interactive time indicates the duration until the DOM is fully prepared for user interaction, and load event time, indicates the total time from page load to the full loading of all resources, such as images and CSS, signifying complete usability of the page. This overall improvement is expected, as our neutralized tracking function calls avoid invoking the original functions, performing less work by simply returning an empty response.

User-centric breakage analysis. We conduct a qualitative manual analysis of NoT.JS, following methodologies from previous studies [82, 155]. We select 50 webpages from the top-10K websites, specifically those hosting scripts mentioned in the exception rules of filter lists[3] and addressed with SugarCoat's six mock API implementations[4]. These are mainly mixed scripts; content blockers typically avoid blocking them to prevent website breakage, even though allowing them may compromise user privacy.

We recruit ten independent evaluators for breakage assessments. Each independent evaluator evaluates five distinct websites from our sample. Each webpage is evaluated by at least two

[3]The EasyList project tags git commit messages addressing compatibility fixes with "P:" - see https://github.com/easylist/easylist/commits.

[4]https://github.com/SugarCoatJS/sugarcoat-paper-dataset/tree/master/resources

different reviewers to ensure a comprehensive analysis. The webpages are evaluated in four different configurations: (1) Control, which displays the default webpage without any blocking; (2) WebGraph, highlighting the limitations of advanced script-level blocking in mixed script scenarios; (3) SugarCoat, highlighting its effectiveness, though it faces scalability challenges, especially with limited mock API implementations[5]; (4) NoT.JS, showing a reduced impact on website functionality compared to WebGraph, while being on par with SugarCoat in terms of handling mixed scripts. This ultimately highlights that NoT.JS significantly advances the handling of mixed scripts compared to current state-of-the-art approaches. We present WebGraph, SugarCoat, and NoT.JS in a randomized sequence to each evaluator, revealing only the control configuration.

We ask evaluators to classify breakage into four categories: navigation (moving between pages), SSO (initiating and maintaining login state), appearance (visual consistency), and miscellaneous (such as chats, search, shopping cart, etc.). Each evaluator labels breakage as either major or minor for each category: Minor breakage occurs when it is difficult but not impossible for the evaluator to use the functionality. Major breakage occurs when it is impossible to use the functionality on a webpage.

Table 5.7 summarizes the results in each category. The inter-evaluator agreement was 95.5%, suggesting substantial agreement and conflicts are resolved by one of the authors revisiting those websites. Conflicts mainly stem from ambiguity between appearance and miscellaneous categories, such as the interchangeable reporting of a missing ad overlay in both. Our analysis shows that WebGraph causes major breakage on 6% and minor breakage on 10% of the webpages. On the other hand, NoT.JS, only causes minor breakage on 8% of the webpages, without any major breakage. NoT.JS matches SugarCoat in terms of breakage but excels in mixed script handling, given SugarCoat's reliance on just six manually crafted

[5] https://github.com/SugarCoatJS/sugarcoat/tree/master/mocks

mock implementations. Specifically, NoT.JS significantly improves the major breakage in the navigation category. For example, on the webpage `bbc.com`, the search bar in the navigation menu disappears with WebGraph, resulting in major breakage. This major breakage stemming from the blocking of the mixed script `_app-*.js` hosted by `bbc.com`, is effectively mitigated when using NoT.JS, which employs a surrogate replacement for `_app-*.js`. Overall, NoT.JS significantly improves upon WebGraph's breakage and SugarCoat's scalability challenges by adeptly identifying and replacing tracking function calls in mixed scripts.

Category	WebGraph		SugarCoat		NoT.JS	
	Minor	Major	Minor	Major	Minor	Major
Navigation	0%	6%	0%	0%	0%	0%
SSO	2%	2%	2%	0%	2%	0%
Appearance	4%	0%	0%	0%	4%	0%
Miscellaneous	4%	4%	4%	0%	4%	0%

Table 5.7: Qualitative manual analysis for 50 webpages using NoT.JS, SugarCoat, and WebGraph, showing % of No, Minor, and Major breakages in navigation, SSO, appearance, and miscellaneous categories.

5.5 Deployment

We deploy NoT.JS to classify all JS functions in our crawl of the top 10K websites, including both known functions (previously labeled in our ground truth) and unknown functions (not previously labeled in our ground truth).

Prevalence of tracking functions. NoT.JS classifies 32.1% of the 2,088K JS functions in our dataset as tracking. We find that these tracking functions are most prevalent in the scripts served by 8,587 unique domains. Among these tracking function, 8.2% are anonymous functions, 18.3% are part of inline scripts, and 1.5% are part of eval scripts. For instance, the

function "Z.D" from the script analytics.js hosted by google-analytics.com appears on 56% of the websites. This function, on average, invokes cookie setter 3.3 times and cookie getter 20 times. Similarly, the "c" function from the script fbevents.js, hosted by connect.facebook.net, appears on 21.5% of the websites. Its typical calling context encompasses 3.9 closures and involves 6.6 get attribute (getAttribute) calls, along with 2.5 cookie accesses.

Characteristics of tracking functions in mixed scripts.

We find that 13.4% of all scripts are mixed, aligning with previous studies [80, 82], while a substantial 62.3% of websites incorporate at least one mixed script. On average, a website contains around 2.42 mixed scripts. Notably, a significant majority (70.6%) of the mixed scripts are served from third-party domain [6], whereas the remaining (29.4%) are served from first-party domains.

Our sampled analysis of the top-100 mixed scripts reveals widespread usage in a first-party context, enabling the setting of ghost first-party cookies [168]. These scripts set 14,867 ghost first-party cookies, out of which 150 ghost first-party cookies are found to be tracking after running CookieGraph [155], a tool designed to detect first-party tracking cookies. NoT.JS classified83% of JS functions as tracking functions that are either setting or getting these ghost first-party tracking cookies. For instance, NoT.JS detected the tracking function "a" within the script launch-*.min.js which is accessing the ghost first-party tracking cookie mbox on adobe.com. Similarly, NoT.JS detected the tracking function "o" within the script opus.js which is accessing the ghost first-party tracking cookie A1S on yahoo.com.

To illustrate how NoT.JS tackles our threat model, consider these two types of mixed scripts. The script named webpack-*.js is served by the domain cloudfront.net. NoT.JS detects tracking and non-tracking functions within this mixed script. Specifically, the function "t"

[6]The domain of the script's URL differs from the top-level URL of the page.

accesses local storage four times, sets it six times, and attaches event listeners to 244 different DOM elements, detected as tracking. In contrast, the `"a"` function in the same script avoids interactions with both local storage and the DOM, detected as non-tracking. Another scenario involves the script `app.js` from the domain `acsbapp.com`, which contains the function `"_e"`. The behavior of this function depends on its dynamic execution context, detected by NoT.JS. In a non-tracking calling context, the number of closures and local variables for the function are 3 and 1, respectively. However, in a tracking calling context, these values are 0, emphasizing the importance of context in distinguishing between tracking and non-tracking behaviors. In summary, NoT.JS adeptly handles the intricacies of mixed scripts by distinguishing between tracking and non-tracking functions, whether they are inherently mixed or influenced by dynamic execution contexts.

5.6 Summary

NoT.JS advances the state-of-the-art by detecting tracking at the JS function-level granularity. To this end, NoT.JS captures the dynamic execution context and then encodes this context to build a rich graph representation that captures individual JavaScript functions. Our evaluation showed that a machine learning classifier based on this graph representation achieves high precision (94%) and recall (98%) in detecting tracking JS functions, outperforming the state-of-the-art in terms of accuracy, robustness, and breakage. We also showed that NoT.JS is able to automatically generate privacy-safe surrogates of mixed scripts that combine tracking and functionality.

Chapter 6

Discussion and Future Work

Below, we discuss some opportunities for future work and limitations.

User interaction limitations. The level of interactions that can be captured by NoT.JS is dependent on the diversity and intensity of user activity, such as scrolling or clicking on internal pages. Consequently, NoT.JS may miss certain tracking functions due to limited user interactions. To mitigate this, we propose to use forced execution [131, 163, 182] in the future to improve the completeness of the page graph.

Browser-specific deployment. NoT.JS uses the Chrome browser and Chrome-based extensions to collect data due to its popularity. Extensions on other browsers (Firefox [33], Safari [42], Edge [37]) have different permissions and access to varying sets of information about a webpage's activity. Porting NoT.JS on other browsers may require additional engineering.

Expanding beyond main thread execution. NoT.JS presently captures only the dynamic execution context of the main thread, leaving out service workers that operate in a separate execution context. Enhancing NoT.JS to include these workers is a targeted area for future work, which will enable a more holistic capture of a webpage's activity.

Resource-constrained environments. While NoT.JS is adept at generating surrogate scripts at a large scale, its deployment on devices with limited storage presents a challenge. These devices cannot store the replacement surrogate scripts. To enhance NoT.JS

in resource-constrained environments, strategies like selectively pre-fetching popular site replacements and using private information retrieval for centralized repository access can be employed to balance efficiency and privacy.

Mixed function analysis. In our ground-truth only 3.9% of the functions are mixed, *i.e.,* involved in both tracking and non-tracking activities. Among these, a mere 0.8% are integral in contexts requiring the blocking of tracking activity, while the remainder can be left unblocked by targeting other tracking functions in the activity's execution chain. Future work will focus on a more detailed analysis of these mixed functions, examining individual statements within the functions rather than considering the entire function block.

Usage of bundlers. To reason out mixing, we identify bundled scripts within mixed scripts by adopting a high-precision heuristic from the Web Almanac [19], which involves searching for the keyword `webpackJsonp` in a script [1]. By default, the `JSONP` function begins with the keyword `webpackJsonp`, which asynchronously loads bundled scripts. However, the keyword `webpackJsonp` can be modified via Webpack's configuration settings, leading to the potential imperfect recall. Moreover, this heuristics only accounts for the scripts that are bundled using Webpack, excluding those bundled with other tools such as Parcel [40], Rollup [41], and Browserify [48]. Thus, the fraction of mixed scripts we identify as bundled is a lower bound. We find that 20.2% of the mixed scripts are bundled using Webpack [67], a rate comparable to that of functional scripts (20%) but higher than that of tracking scripts (10%). In the future, we can investigate whether the mixing of scripts is a result of using such bundlers.

[1] https://v4.webpack.js.org/configuration/output/#outputjsonpfunction

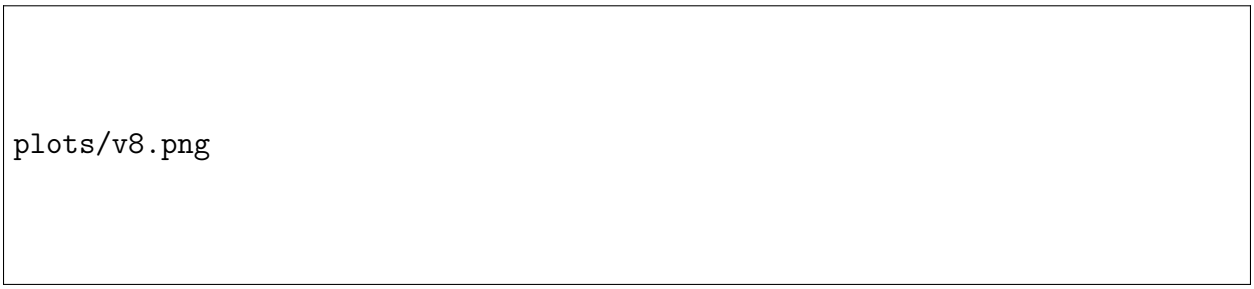


Figure 6.1: Gant chart for V8-based tracking function identification and blocking.

6.1 Chrome V8 Parser-Based Replacement for Tracking JavaScript Functions

We plan to explore an alternative solution to block mixed resources without disrupting legitimate functionality, while also addressing the key limitations of NoT.JS. NoT.JS has three notable shortcomings. Firstly, its surrogate script generation is an offline process, which is then used to create filter rules for replacing mixed scripts. This approach is vulnerable to evasion strategies like rotating domains. Secondly, while NoT.JS effectively generates surrogate scripts at scale, its deployment is challenging on devices with limited storage, which cannot accommodate these scripts. Thirdly, NoT.JS is currently limited to capturing the dynamic execution context of the main thread, excluding service workers operating in separate contexts. To remedy these issues, our focus shifts to identifying and replacing tracking functions at the V8 engine level, specifically at the AST (Abstract Syntax Tree). This strategy aims to address all three of NoT.JS's limitations. We have two main contributions: developing features for tracking detection at the AST level and implementing AST-level replacements to block calls to privacy-invading JavaScript APIs, thereby disrupting data collection at its source. This project is a collaborative effort with Brave [12], seeking to enhance web browsing privacy by improving NoT.JS's capabilities.

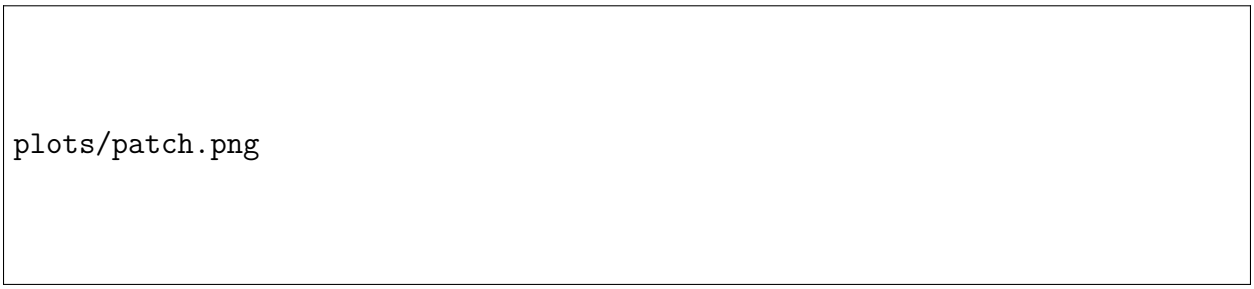


Figure 6.2: Gant chart for API-patching.

6.2 API Patching for Mixed Resources: Evaluating Viability and Challenges

In this thrust, we plan to investigate potential challenges associated with API patching, a common solution employed in current practices like scriptlets [65] and SuugarCoat [178]. We hypothesize that race conditions in API patching could pose significant issues. This is because any third-party script or even a Chrome extension has the potential to override these APIs, potentially leading to race conditions and subsequent disruption of intended functionalities. The primary objective of this project is twofold. Firstly, we aim to identify whether there are scripts or extensions that override privacy-invasive APIs and, if so, to understand the underlying reasons, whether they are performance-related or have other, possibly malicious, intentions. Secondly, we examine whether these race conditions could interfere with the existing solutions for handling mixed resources, potentially complicating their effectiveness. Ultimately, this research seeks to determine if there is a need to refine our techniques in the future to either address or avoid such race conditions, ensuring that current solutions continue to function effectively without unintended consequences.

Chapter 7

Conclusion

This thesis effectively confronts the widespread issue of advertising and tracking on the internet, particularly the recent strategy of mixing tracking with functional resources, which challenges the efficacy of content-blockers. Our approach, grounded in code-aware techniques like localization and refactoring strategies, presents a robust solution to this problem. The development of TrackerSift has been instrumental in revealing the extent of mixed resources and underscores the effectiveness of targeting JavaScript code for blocking. Our investigations have identified function-level granularity as the optimal level for JavaScript code blocking, successfully balancing the need to block intrusive trackers while maintaining essential website functionality. The introduction of NoT.JS, a machine learning-based classifier functioning at this granularity, marks a significant advancement. It skillfully identifies tracking functions and creates surrogate scripts that selectively eliminate tracking code while preserving functional code within mixed scripts. Overall, this thesis enhances the capabilities of content-blockers, significantly improving their ability to manage mixed scripts and, consequently, enhancing user privacy on the internet.

Bibliography

[1] A new way to control the ads you see on facebook, and an update on ad blocking, 2016. URL https://newsroom.fb.com/news/2016/08/a-new-way-to-control-the-ads-you-see-on-facebook-and-an-update-on-ad-blocking/.

[2] Ping pong with facebook. https://web.archive.org/web/20160818160457/https://adblockplus.org/blog/ping-pong-with-facebook, 2016.

[3] Ad blocker's successful assault on facebook enters its second month. https://web.archive.org/web/20221011151030/https://adage.com/article/digital/blockrace-adblock/311103, 2017.

[4] Ad network uses dga algorithm to bypass ad blockers and deploy in-browser miners. https://www.bleepingcomputer.com/news/security/ad-network-uses-dga-algorithm-to-bypass-ad-blockers-and-deploy-in\-browser-miners/, 2018.

[5] Who is stealing my power iii: An adnetwork company case study. https://web.archive.org/web/20230408021633/https://blog.netlab.360.com/who-is-stealing-my-power-iii-an-adnetwork-company-case-study-en/, 2018.

[6] ABP anti-circumvention filter list . https://github.com/abp-filters/abp-filters-anti-cv, 2019.

[7] AdGuard Scriptlets and Resources. https://github.com/AdguardTeam/Scriptlets, 2019.

[8] Webbundles harmful to content blocking, security tools, and the open web. https:

//web.archive.org/web/20230207160252/https://brave.com/web-standards-at-brave/3-web-bundles/, 2020.

[9] gorhill/ublock: ublock origin - an efficient blocker for chromium and firefox. fast and lean. https://github.com/gorhill/uBlock, July 2020.

[10] Adblock Plus. https://adblockplus.org/, 2021.

[11] AdGuard Scriptlets and Redirect resources. https://github.com/AdguardTeam/Scriptlets, 2021.

[12] Brave Browser. https://brave.com/, 2021.

[13] Extending devtools, 2021. URL https://developer.chrome.com/docs/extensions/mv3/devtools/.

[14] Adblock summit 2021. https://www.youtube.com/watch?v=V9CG0wdmfY4, 2021.

[15] Firefox 87 introduces smartblock for private browsing, Mar 2021. URL https://blog.mozilla.org/security/2021/03/23/introducing-smartblock/.

[16] Facebook pixel: Implementation, 2021. URL https://developers.facebook.com/docs/facebook-pixel/implementation/.

[17] Security/trackingprotectionbreakage, 2021. URL https://wiki.mozilla.org/Security/TrackingProtectionBreakage#Trivial_shim_needed_to_avoid_breakage.3B_no_yellowlisting_required.

[18] uBO-Scriptlets: A custom arsenal of scriptlets to be used for injecting userscripts via uBlock Origin. https://github.com/uBlock-user/uBO-Scriptlets,, 2021.

[19] Web almanac - javascript. https://web.archive.org/web/20230422093455/https://almanac.httparchive.org/en/2022/javascript, 2022.

[20] Adblock summit 2023. https://youtu.be/-KkMWUaajhU?t=27712, 2022.

[21] Easylist. https://easylist.to/easylist/easylist.txt, 2022.

[22] Easyprivacy. https://easylist.to/easylist/easyprivacy.txt, 2022.

[23] Tracking prevention in microsoft edge. https://docs.microsoft.com/en-us/microsoft-edge/web-platform/tracking-prevention, 2022.

[24] Facebook is now encrypting links to prevent url stripping. Website, 2022. URL https://web.archive.org/web/20230527091045/https://www.schneier.com/blog/archives/2022/07/facebook-is-now-encrypting-links-to-prevent-url-stripping.html.

[25] Enhanced tracking protection in firefox for desktop. https://support.mozilla.org/en-US/kb/enhanced-tracking-protection-firefox-desktop, 2022.

[26] Adblock summit 2022. https://youtu.be/yiNOaq4Py48?t=27247, 2022.

[27] ublock origin. https://github.com/gorhill/uBlock, 2022. URL https://github.com/gorhill/uBlock.

[28] Console trace. https://developer.mozilla.org/en-US/docs/Web/API/console/trace, 2023.

[29] Debugger api. https://chromedevtools.github.io/devtools-protocol/tot/Debugger/, 2023.

[30] Cookie access. https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie, 2023.

[31] EasyList. https://easylist.to/easylist/easylist.txt, 2023.

[32] Fetch api. https://chromedevtools.github.io/devtools-protocol/tot/Fetch/, 2023.

[33] Firefox browser. https://www.mozilla.org/en-US/firefox/, 2023.

[34] uBlock Origin Google Analytics Replacement. https://github.com/gorhill/uBlock/blob/master/src/web_accessible_resources/google-analytics_analytics.js, 2023.

[35] Local storage access. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage, 2023.

[36] Migrate from manifest v2 to manifest v3. https://developer.chrome.com/docs/extensions/migrating/blocking-web-requests/, 2023.

[37] Microsoft-edge browser. https://www.microsoft.com/en-us/edge?form=MA13FJ, 2023.

[38] Declarative net request api. https://developer.chrome.com/docs/extensions/reference/declarativeNetRequest/, 2023.

[39] NoScript. https://noscript.net/, 2023.

[40] Parcel. https://parceljs.org/, 2023.

[41] Rollup. https://rollupjs.org/, 2023.

[42] Safari browser. https://www.apple.com/safari/, 2023.

[43] Google. 2021. chrome user experience report. https://web.dev/articles/critical-rendering-path/measure-crp, 2023.

[44] V8. https://en.wikipedia.org/wiki/V8_(JavaScript_engine), 2023.

[45] Background script. https://developer.chrome.com/docs/extensions/mv2/background_pages/, 2023.

[46] Ad blockers usage and demographic statistics in 2022, Mar 2023. URL https://backlinko.com/ad-blockers-users.

[47] How to enable or disable JavaScript in a browser? https://www.computerhope.com/issues/ch000891.htm, 2023.

[48] Browserify. http://browserify.org/, 2023.

[49] chrome-extension. https://developer.chrome.com/docs/extensions/, 2023.

[50] Chrome DevTools Protocol — chromedevtools.github.io. https://chromedevtools.github.io/devtools-protocol/, 2023.

[51] Content script. https://developer.chrome.com/docs/extensions/mv3/content_scripts/, 2023.

[52] Disconnect Tracking protection lists. https://disconnect.me/trackerprotection, 2023.

[53] eplais. https://www.similarweb.com/website/elpais.com/, 2023.

[54] Filter-lists. https://filterlists.com/, 2023.

[55] gamestop. https://www.similarweb.com/website/gamestop.com/, 2023.

[56] livescore. https://www.similarweb.com/top-websites/category/sports/soccer/, 2023.

[57] HTML5 The noscript element. https://www.w3.org/TR/2011/WD-html5-author-20110809/the-noscript-element.html, 2023.

[58] obfuscater.io. https://obfuscator.io/, 2023.

[59] webalmnac:onloac. https://perma.cc/W9ZW-DP2D, 2023.

[60] onload event. https://httparchive.org/reports/loading-speed#ol, 2023.

[61] uBlock Origin web accessible resources. https://github.com/gorhill/uBlock/tree/master/src/web_accessible_resources, 2023.

[62] Selenium. https://www.selenium.dev/, 2023.

[63] Sample size calculator. https://www.surveymonkey.com/mp/sample-size-calculator/, 2023.

[64] tenki. https://www.similarweb.com/website/tenki.jp/, 2023.

[65] ublock origin scriptlets. https://github.com/uBlock-user/uBO-Scriptlets, 2023. URL https://github.com/uBlock-user/uBO-Scriptlets.

[66] washingtonpost. https://www.similarweb.com/website/washingtonpost.com/, 2023.

[67] Webpack. https://webpack.js.org, 2023.

[68] XMLHttpRequest. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest, 2023.

[69] Github issue-4. [redacted], 2024. Link redacted to comply with the double-blind review policy.

[70] Github issue-2. [redacted], 2024. Link redacted to comply with the double-blind review policy.

[71] Github issue-1. [redacted], 2024. Link redacted to comply with the double-blind review policy.

[72] Github issue-3. [redacted], 2024. Link redacted to comply with the double-blind review policy.

[73] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007.

[74] Pragya Agarwal and Arun Prakash Agrawal. Fault-localization techniques for software systems: A literature review. *SIGSOFT Softw. Eng. Notes*, 2014.

[75] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[76] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. Errors, misunderstandings, and attacks: Analyzing the crowdsourcing process of ad-blocking systems. In *Proceedings of the Internet Measurement Conference*, IMC '19, page 230–244, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369480. doi: 10.1145/3355369.3355588. URL https://doi.org/10.1145/3355369.3355588.

[77] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. Errors, Misunderstandings, and Attacks: Analyzing the Crowdsourcing Process of Ad-blocking Systems. In *ACM Internet Measurement Conference (IMC)*, 2019.

[78] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. Errors, misunderstandings, and attacks: Analyzing the crowdsourcing process of ad-blocking systems. In *Proceedings of the Internet Measurement Conference*, pages 230–244, 2019.

[79] Abdul Haddi Amjad, Danial Saleem, Muhammad Ali Gulzar, Zubair Shafiq, and Fareed Zaffar. Trackersift: untangling mixed tracking and functional web resources. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 569–576, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391290. doi: 10.1145/3487552.3487855. URL https://doi.org/10.1145/3487552.3487855.

[80] Abdul Haddi Amjad, Danial Saleem, Muhammad Ali Gulzar, Zubair Shafiq, and Fareed Zaffar. Trackersift: Untangling mixed tracking and functional web resources. In *Proceedings of the 21st ACM Internet Measurement Conference*. Association for Computing Machinery, 2021.

[81] Abdul Haddi Amjad, Danial Saleem, Muhammad Ali Gulzar, Zubair Shafiq, and Fareed Zaffar. Trackersift: Untangling mixed tracking and functional web resources. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 569–576, 2021.

[82] Abdul Haddi Amjad, Zubair Shafiq, and Muhammad Ali Gulzar. Blocking javascript without breaking the web: An empirical investigation. *arXiv preprint arXiv:2302.01182*, 2023.

[83] Abdul Haddi Amjad, Shaoor Munir, Zubair Shafiq, and Muhammad Ali Gulzar. Blocking tracking javascript at the function granularity. *arXiv preprint arXiv:2405.18385*, 2024.

[84] Pouneh Nikkhah Bahrami, Umar Iqbal, and Zubair Shafiq. Fp-radar: Longitudinal measurement and early detection of browser fingerprinting. *arXiv preprint arXiv:2112.01662*, 2021.

[85] Pouneh Nikkhah Bahrami, Umar Iqbal, and Zubair Shafiq. Fp-radar: Longitudinal

measurement and early detection of browser fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2022(2):557–577, 2022.

[86] Vivek Balachandran, Darell JJ Tan, Vrizlynn LL Thing, et al. Control flow obfuscation for android applications. *Computers & Security*, 61:72–93, 2016.

[87] Muhammad Ahmad Bashir, Sajjad Arshad, Engin Kirda, William Robertson, and Christo Wilson. How tracking companies circumvented ad blockers using websockets. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, page 471–477, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356190. doi: 10.1145/3278532.3278573. URL https://doi.org/10.1145/3278532.3278573.

[88] Muhammad Ahmad Bashir, Sajjad Arshad, Engin Kirda, William Robertson, and Christo Wilson. How tracking companies circumvented ad blockers using websockets. In *Proceedings of the Internet Measurement Conference (IMC)*, 2018.

[89] Brave. A long list of ways brave goes beyond other browsers to protect your privacy. https://brave.com/privacy-features/, 2022. URL https://brave.com/privacy-features/.

[90] Moira Burke, Anthony Hornof, Erik Nilsen, and Nicholas Gorman. High-cost banner blindness: Ads increase perceived workload, hinder visual search, and are forgotten. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 12(4):423–445, 2005.

[91] Moumena Chaqfeh, Yasir Zaki, Jacinta Hu, and Lakshmi Subramanian. Jscleaner: De-cluttering mobile webpages through javascript cleanup. In *Proceedings of The Web Conference 2020*, 2020.

[92] Moumena Chaqfeh, Russell Coke, Jacinta Hu, Waleed Hashmi, Lakshmi Subramanian,

Talal Rahwan, and Yasir Zaki. Jsanalyzer: A web developer tool for simplifying mobile web pages through non-critical javascript elimination. *ACM Transactions on the Web*, 2022.

[93] Quan Chen, Panagiotis Ilia, Michalis Polychronakis, and Alexandros Kapravelos. Cookie swap party: Abusing first-party cookies for web tracking. In *Proceedings of the Web Conference 2021*, pages 2117–2129, 2021.

[94] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. Detecting filter list evasion with event-loop-turn granularity javascript signatures. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1715–1729, 2021. doi: 10.1109/SP40001.2021.00007.

[95] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. Detecting filter list evasion with event-loop-turn granularity javascript signatures. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[96] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. Detecting Filter List Evasion With Event-Loop-Turn Granularity JavaScript Signatures. In *IEEE Symposium on Security and Privacy*, 2021.

[97] Yuyu Chen. Tough sell: Why publisher 'turn-off-your-ad-blocker' messages are so polite - digiday. https://digiday.com/media/tough-sell-publisher-turn-off-ad-blocker-messages-polite/, April 2016.

[98] Wai Ting Cheung, Sukyoung Ryu, and Sunghun Kim. Development nature matters: An empirical study of code clones in javascript applications. *Empirical Software Engineering*, 21(2):517–564, 2016.

[99] YoungHan Choi, TaeGhyoon Kim, SeokJin Choi, and Cheolwon Lee. Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis. In *Future Generation Information Technology: First International Conference, FGIT 2009, Jeju Island, Korea, December 10-12, 2009. Proceedings 1*, pages 160–172. Springer, 2009.

[100] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *USENIX Security Symposium*, 2011.

[101] Ha Dao and Kensuke Fukuda. Characterizing cname cloaking-based tracking on the web. In *Traffic Monitoring and Analysis*, 2020. URL https://api.semanticscholar.org/CorpusID:219957591.

[102] Ha Dao, Johan Mazel, and Kensuke Fukuda. Characterizing cname cloaking-based tracking on the web. *IEEE/IFIP TMA'20*, pages 1–9, 2020.

[103] Ha Dao, Johan Mazel, and Kensuke Fukuda. Cname cloaking-based tracking on the web: Characterization, detection, and protection. *IEEE Transactions on Network and Service Management*, 2021.

[104] Sean M Devine and Nathaniel D Bastian. An adversarial training based machine learning approach to malware classification under adversarial conditions. In *HICSS*, pages 1–10, 2021.

[105] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem. The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion. *PETS*, 2021.

[106] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem.

The cname of the game: Large-scale analysis of dns-based tracking evasion. *Proceedings on Privacy Enhancing Technologies*, 2021(3):394–412, 2021.

[107] Marwa El-Wahab, Amal Aboutabl, and Wessam El-Behaidy. Graph mining for software fault localization: An edge ranking based approach. *Journal of Communications Software and Systems*, 13:178–188, 01 2018. doi: 10.24138/jcomss.v13i4.402.

[108] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016.

[109] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, page 213–224, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581130740. doi: 10.1145/302405.302467. URL https://doi.org/10.1145/302405.302467.

[110] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[111] Aurore Fass, Michael Backes, and Ben Stock. Jstap: A static pre-filter for malicious javascript detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019.

[112] Kiran Garimella, Orestis Kostakis, and Michael Mathioudakis. Ad-blocking: A study on performance, privacy and counter-measures. In *Proceedings of the 2017 ACM on Web Science Conference*, 2017.

[113] Liang Gong, Hongyu Zhang, Hyunmin Seo, and Sunghun Kim. Locating crashing faults based on crash stack traces. In *arXiv:1404.4100*, 2014.

[114] Le Hieu, Markopoulou Athina, and Shafiq Zubair. Cv-inspector: Towards automating detection of adblock circumvention. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[115] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. MutantX-S: Scalable Malware Clustering Based on Static Features. In *2013 USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[116] Muhammad Ikram, Hassan Jameel Asghar, Mohamed Ali Kaafar, Balachander Krishnamurthy, and Anirban Mahanti. Towards seamless tracking-free web: Improved detection of trackers via one-class learning, 2016.

[117] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. The ad wars: retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, page 171–183, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351188. doi: 10.1145/3131365.3131387. URL https://doi.org/10.1145/3131365.3131387.

[118] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. The ad wars: Retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the 2017 Internet Measurement Conference*. Association for Computing Machinery, 2017.

[119] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *IMC*, 2017.

[120] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. The ad wars: Retrospective measurement

and analysis of anti-adblock filter lists. In *ACM Internet Measurement Conference (IMC)*, 2017.

[121] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. Adgraph: A graph-based approach to ad and tracker blocking. IEEE, 2020.

[122] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. Adgraph: A graph-based approach to ad and tracker blocking. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 763–776. IEEE, 2020.

[123] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *To appear in the Proceedings of the IEEE Symposium on Security & Privacy*, 2021.

[124] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320, 2011.

[125] Tom Janssen, Rui Abreu, and Arjan JC Van Gemund. Zoltar: A toolset for automatic fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664. IEEE, 2009.

[126] Shujuan Jiang, Wei Li, Haiyang Li, Yanmei Zhang, Hongchang Zhang, and Yingqi Liu. Fault localization for null pointer exception based on stack trace and program slicing. In *2012 12th International Conference on Quality Software*, 2012.

[127] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 273–282, New

York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139934. doi: 10.1145/1101908.1101949. URL https://doi.org/10.1145/1101908.1101949.

[128] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[129] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 467–477, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 158113472X. doi: 10.1145/581339.581397. URL https://doi.org/10.1145/581339.581397.

[130] Jordan Jueckstock and Alexandros Kapravelos. Visiblev8: In-browser monitoring of javascript in the wild. In *Proceedings of the Internet Measurement Conference*, pages 393–405, 2019.

[131] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. J-force: Forced execution on javascript. In *Proceedings of the 26th international conference on World Wide Web*, pages 897–906, 2017.

[132] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information processing letters*, 29(3):155–163, 1988.

[133] Jesutofunmi Kupoluyi, Moumena Chaqfeh, Matteo Varvello, Russell Coke, Waleed Hashmi, Lakshmi Subramanian, and Yasir Zaki. Muzeel: Assessing the impact of javascript dead code elimination on mobile web performance. In *Proceedings of the 22nd ACM Internet Measurement Conference*, 2022.

[134] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. Localising faults in test

execution traces. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*, IWPSE 2015, page 1–8, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338165. doi: 10.1145/2804360.2804361. URL https://doi.org/10.1145/2804360.2804361.

[135] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[136] Hieu Le, Athina Markopoulou, and Zubair Shafiq. Cv-inspector: Towards automating detection of adblock circumvention. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[137] Hieu Le, Salma Elmalaki, Athina Markopoulou, and Zubair Shafiq. Autofr: Automated filter rule generation for adblocking. *USENIX Security Symposium*, 2023.

[138] 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. doi: 10.14722/ndss.2019.23386.

[139] K Rustan M Leino and Peter Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming*, pages 491–515. Springer, 2004.

[140] Su-Chin Lin, Kai-Hsiang Chou, Yen Chen, Hsu-Chun Hsiao, Darion Cassel, Lujo Bauer, and Limin Jia. Investigating advertisers' domain-changing behaviors and their impacts on ad-blocker filter lists. In *Proceedings of the ACM Web Conference 2022*, pages 576–587, 2022.

[141] Matthew Malloy, Mark McNamara, Aaron Cahn, and Paul Barford. Ad blockers: Global prevalence and impact. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, 2016.

[142] Matthew Malloy, Mark McNamara, Aaron Cahn, and Paul Barford. Ad Blockers: Global Prevalence and Impact. In *ACM Internet Measurement Conference (IMC)*, 2016.

[143] Matthew Malloy, Mark McNamara, Aaron Cahn, and Paul Barford. Ad blockers: Global prevalence and impact. In *Proceedings of the 2016 internet measurement conference*, pages 119–125, 2016.

[144] Jian Mao, Jingdong Bian, Guangdong Bai, Ruilong Wang, Yue Chen, Yinhao Xiao, and Zhenkai Liang. Detecting malicious behaviors in javascript applications. *IEEE Access*, 6, 2018.

[145] Giorgio Maone. Surrogate Scripts vs Google Analytics. https://hackademix.net/2009/01/25/surrogate-scripts-vs-google-analytics/.

[146] Wes Masri. Automated fault localization: Advances and challenges. *Advances in Computers*, 99:103–156, 2015.

[147] Jonathan R Mayer and John C Mitchell. Third-party web tracking: Policy and technology. In *2012 IEEE symposium on security and privacy*, pages 413–427. IEEE, 2012.

[148] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar R. Weippl. Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools. In *IEEE European Symposium on Security and Privacy*, 2017.

[149] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering*, 41(5):429–444, 2014.

[150] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151, 2006.

[151] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. Statically detecting javascript obfuscation and minification techniques in the wild. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 569–580. IEEE, 2021.

[152] Muhammad Haris Mughees, Zhiyun Qian, Zubair Shafiq, Karishma Dash, and Pan Hui. A first look at ad-block detection: A new arms race on the web. *arXiv preprint arXiv:1605.05841*, 2016.

[153] Muhammad Haris Mughees, Zhiyun Qian, and Zubair Shafiq. Detecting Anti Ad-blockers in the Wild . In *Privacy Enhancing Technologies Symposium (PETS)*, 2017.

[154] Shaoor Munir, Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. Cookiegraph: Measuring and countering first-party tracking cookies. *arXiv:2208.12370*, 2022.

[155] Shaoor Munir, Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. Cookiegraph: Understanding and detecting first-party tracking cookies, 2023.

[156] Ray Ngan, Surya Konkimalla, and Zubair Shafiq. Nowhere to hide: Detecting obfuscated fingerprinting scripts. *arXiv preprint arXiv:2206.13599*, 2022.

[157] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to generate disjunctive invariants. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 608–619, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568275. URL https://doi.org/10.1145/2568225.2568275.

[158] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[159] Rishab Nithyanand, Sheharbano Khattak, Mobin Javed, Narseo Vallina-Rodriguez, Marjan Falahrastegar, Julia E. Powles, Emiliano De Cristofaro, Hamed Haddadi, and Steven J. Murdoch. Adblocking and Counter-Blocking: A Slice of the Arms Race. In *USENIX Workshop on Free and Open Communications on the Internet*, 2016.

[160] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. *SIGPLAN Not.*, 51(6):42–56, June 2016. ISSN 0362-1340. doi: 10.1145/2980983.2908099. URL https://doi.org/10.1145/2980983.2908099.

[161] Page Fair. The State of the Blocked Web. https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf, 2017.

[162] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620, 2017. doi: 10.1109/ICSE.2017.62.

[163] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong

Su. {X-Force}:{Force-Executing} binary programs for security applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 829–844, 2014.

[164] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*, 2018.

[165] Yan Qin, Weiping Wang, Zixian Chen, Hong Song, and Shigeng Zhang. Transast: A machine translation-based approach for obfuscated malicious javascript detection. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 327–338. IEEE, 2023.

[166] Kunlun Ren, Weizhong Qiang, Yueming Wu, Yi Zhou, Deqing Zou, and Hai Jin. An empirical study on the effects of obfuscation on static machine learning-based malicious javascript detectors. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1420–1432, 2023.

[167] Kaleigh Rogers. Why doesn't my ad blocker block 'please turn off your ad blocker' popups? - vice. https://www.vice.com/en_us/article/j5zk8y/why-your-ad-blocker-doesnt-block-those-please-turn-off-your-ad-blocker-popups, December 2018.

[168] Iskander Sanchez-Rola, Matteo Dell'Amico, , Davide Balzarotti, Pierre-Antoine Vervier, and Leyla Bilge. Journey to the center of the cookie ecosystem: Unraveling actors'; roles and relationships. In *S&P 2021, 42nd IEEE Symposium on Security & Privacy, 23-27 May 2021, San Francisco, CA, USA*, 2021.

[169] Shaown Sarker, Jordan Jueckstock, and Alexandros Kapravelos. Hiding in plain site: Detecting javascript obfuscation through concealed browser api usage. In *Proceedings of the ACM Internet Measurement Conference*, 2020.

[170] Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. Webgraph: Capturing advertising and tracking information flows for robust blocking. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022.

[171] Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. {WebGraph}: Capturing advertising and tracking information flows for robust blocking. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2875–2892, 2022.

[172] Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. WebGraph: Capturing advertising and tracking information flows for robust blocking. In *USENIX Security Symposium*, 2022.

[173] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. Filter list generation for underserved regions. *CoRR*, abs/1910.07303, 2019. URL http://arxiv.org/abs/1910.07303.

[174] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. Filter list generation for underserved regions. WWW '20. Association for Computing Machinery, 2020.

[175] Alexander Sjosten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. Filter List Generation for Underserved Regions. In *The Web Conference*, 2020.

[176] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *World Wide Web (WWW) Conference*, 2019.

[177] Michael Smith, Pete Snyder, Benjamin Livshits, and Deian Stefan. Sugarcoat: Programmatically generating privacy-preserving, web-compatible resource replacements for content blocking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2844–2857, 2021.

[178] Michael Smith, Pete Snyder, Benjamin Livshits, and Deian Stefan. Sugarcoat: Programmatically generating privacy-preserving, web-compatible resource replacements for content blocking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2844–2857, 2021.

[179] Peter Snyder, Antoine Vastel, and Ben Livshits. Who filters the filters: Understanding the growth, usefulness and efficiency of crowdsourced ad blocking. *Proc. ACM Meas. Anal. Comput. Syst.*, 2020.

[180] Peter Snyder, Antoine Vastel, and Ben Livshits. Who filters the filters: Understanding the growth, usefulness and efficiency of crowdsourced ad blocking. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(2):1–24, 2020.

[181] H. A. D. Souza, M. L. Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *ArXiv*, abs/1607.04347, 2016.

[182] Zhenhao Tang, Juan Zhai, Minxue Pan, Yousra Aafer, Shiqing Ma, Xiangyu Zhang, and Jianhua Zhao. Dual-force: Understanding webview malware via cross-language forced execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 714–725, 2018.

[183] Phani Vadrevu and Roberto Perdisci. What you see is not what you get: Discovering and tracking social engineering attack campaigns. In *Proceedings of the Internet Measurement Conference*, pages 308–321, 2019.

[184] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Arpád Beszédes. Call frequency-based fault localization.

[185] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Call frequency-based fault localization. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 365–376. IEEE, 2021.

[186] Antoine Vastel, Peter Snyder, and Benjamin Livshits. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced AdBlocking. In *ACM SIGMETRICS/Performance*, 2020.

[187] Hernán Ceferino Vázquez, Alexandre Bergel, Santiago Vidal, JA Díaz Pace, and Claudia Marcos. Slimming javascript applications: An approach for removing unused functions from javascript libraries. *Information and software technology*, 2019.

[188] Jeremy Wagner. Javascript: 2022: Web almanac by http archive, 2022. URL https://almanac.httparchive.org/en/2022/javascript.

[189] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. Webranz: Web page randomization for better advertisement delivery and web-bot prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, 2016.

[190] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. Webranz: web page randomization for better advertisement delivery and web-bot prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 205–216, 2016.

[191] Zhi Yue Wang and Wei Min Wu. Technique of javascript code obfuscation based on control flow tansformations. *Applied Mechanics and Materials*, 519:391–394, 2014.

[192] Shiyi Wei and Barbara G Ryder. State-sensitive points-to analysis for the dynamic behavior of javascript objects. In *ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28*, pages 1–26. Springer, 2014.

[193] Shiyi Wei, Franceska Xhakaj, and Barbara G Ryder. Empirical study of the dynamic behavior of javascript objects. *Software: Practice and Experience*, 46(7):867–889, 2016.

[194] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 61–72, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950340. URL https://doi.org/10.1145/2950290.2950340.

[195] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[196] Wei Xu, Fangfang Zhang, and Sencun Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 9–16. IEEE, 2012.

[197] Wei Xu, Fangfang Zhang, and Sencun Zhu. Jstill: mostly static detection of obfuscated malicious javascript code. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 117–128, 2013.

[198] Eric Zeng, Tadayoshi Kohno, and Franziska Roesner. What makes a "bad" ad? user

perceptions of problematic online advertising. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–24, 2021.

[199] Zhaoqi Zhang, Panpan Qi, and Wei Wang. Dynamic malware analysis with feature engineering and feature learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1210–1217, 2020.